The Dissertation Committee for Ramakrishna Rao Kotla
certifies that this is the approved version of the following dissertation:

xBFT: Byzantine Fault Tolerance with High Performance, Low Cost, and Aggressive Fault Isolation

Committee:

Michael D. Dahlin, Supervisor

Lorenzo Alvisi

Vijay Garg

Adnan Aziz

Thomas W. Keller

xBFT: Byzantine Fault Tolerance with High Performance, Low Cost, and Aggressive Fault Isolation

by

Ramakrishna Rao Kotla, B.Tech., M.S.E

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2008

To my beloved grand father (Thathayya), Bhujanga Rao Boinpalli.

Acknowledgments

My journey as a graduate student started with an end goal of earning a doctoral degree. But, now, it is the journey that I cherish more than the achieved goal.

First, I must thank my advisor Mike Dahlin the most. He not only helped me in setting my research direction but also played a major role in shaping me as a researcher. I am really glad that he has given me ample freedom to work on problems that interested me while ensuring that I produced the best work by critiquing dispassionately about various aspects of the problems that I worked on. His sermons on clear presentation, implementation, and evaluation of ideas in experimental software systems has influenced the way I approach, think about, and present problems in systems research. It was a great experience working with Mike who understands and appreciates the importance of building practical systems.

I am really fortunate to work closely with Lorenzo Alvisi in the LASR lab. I am impressed as well as influenced by his research style with emphasis on attention for details which adds scientific rigor to system research. He is a great mentor who can boost your morale when chips are down and makes the LASR lab a fun place to work at.

I would like to thank Harrick Vin for introducing me to the LASR lab and for giving me an opportunity to work on interesting problems. Although we did not closely collaborate over the years, we discussed on the problems that I was working on when possible. I must thank Adnan Aziz who served as my advisor in the ECE department when I started my graduate study. It was my pleasure to write a paper with him based on a project we have done in his course. I would like to thank Vijay garg who served as a co-advisor for my Master's thesis as well as doctoral dissertation committee. I would like to thank Tom Keller for

serving as my external dissertation committee member and also for providing me an opportunity to work on power-aware systems research as an intern at IBM Research.

My friendship with Ravi, Amol, Prem, Joseph, Aniket, Jayaram, Sugat, and Praveen made my life easier even during the toughest of times. I had fun time with Tanmoy, Ravi, ACP, and Arun, who shared apartments with me at various times during my grad life. I have enjoyed my interactions with other grad students in the LASR lab: Jiandan, Jeff, JP, Allen, Ed, Harry, Amit, Prince, Navendu, Nalini, and Taylor. I am thankful to them for patiently listening to my practice talks. I have enjoyed collaborating with Allen and Ed on Zyzzyva work. Allen extended and improved the quality of proofs for Zyzzyva protocol. Ed helped in running Zyzzyva experiments.

This dissertation would not have been possible without the unconditional love and support from my parents, wife, and other family members. I had a great teacher in my grandfather who taught me at a very young age that the real learning starts from questioning. My mom taught me the value of working hard without ever saying a word. I am lucky to be married to priya whose unflinching love and support helped me safely get through some of the worse testing times. It is no coincidence that my best research work came after I married her. I will always cherish the wonderful time I had with my sister's family, especially the time I spent with my niece and nephew who made sure that I smiled even on bad days. I am also fortunate to have support from other extended family members.

xBFT: Byzantine Fault Tolerance with High Performance, Low Cost, and Aggressive Fault Isolation

Publication No. _____

Ramakrishna Rao Kotla, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Michael D. Dahlin

We are increasingly relying on online services to store, access, share, and disseminate critical information from anywhere and at all times. Such services include email, digital storage, photos, video, health and financial services, etc. With increasing evidence of non-fail-stop failures in practical systems, Byzantine fault tolerant state machine replication technique is becoming increasingly attractive for building highly-reliable services in order to tolerate such failures. However, existing Byzantine fault tolerant techniques fall short of providing high availability, high performance, and long-term data durability guarantees with competitive replication cost.

In this dissertation, we present BFT replication techniques that facilitate the design and implementation of such highly-reliable services by providing high availability, high performance and high durability with competitive replication cost (hardware, software, network, management).

First, we propose CBASE, a BFT state machine replication architecture that leverages application-level parallelism to improve throughput of the replicated system by identifying and executing independent requests concurrently. Traditional state machine replication based Byzantine fault tolerant (BFT) techniques

provide high availability and security but fail to provide high throughput. This limitation stems from the fundamental assumption of generalized state machine replication techniques that all replicas execute requests sequentially in the same total order to ensure consistency across replicas. Our architecture thus provides a general way to exploit application parallelism in order to provide high throughput without compromising correctness.

Second, we present Zyzzyva, an efficient BFT agreement protocol that uses speculation to significantly reduce the performance overhead and replication cost of BFT state machine replication. In Zyzzyva, replicas respond to a client's request without first running an expensive three-phase commit protocol to reach agreement on the order in which the request must be processed. Instead, they optimistically adopt the order proposed by the primary and respond immediately to the client. Replicas can thus become temporarily inconsistent with one another, but clients detect inconsistencies, help correct replicas converge on a single total ordering of requests, and only rely on responses that are consistent with this total order. This approach allows Zyzzyva to reduce replication overheads to near their theoretical minima.

Third, we design and implement SafeStore, a distributed storage system designed to maintain long-term data durability despite conventional hardware and software faults, environmental disruptions, and administrative failures caused by human error or malice. The architecture of SafeStore is based on *fault isolation*, which SafeStore applies aggressively along administrative, physical, and temporal dimensions by spreading data across autonomous storage service providers (SSPs). SafeStore also performs an efficient end-to-end audit of SSPs to detect data loss quickly and improve data durability by reducing MTTR. SafeStore offers durable storage with cost, performance, and availability competitive with traditional storage systems.

We evaluate these techniques by implementing BFT replication libraries and further demonstrate the practicality of these approaches by implementing an NFS based replicated file system(CBASE-FS) and a durable storage system (SafeStore-FS).

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are increasingly relying on online services to store, access, share, and disseminate critical information from anywhere and at all times. Such services include email [10, 19, 39], digital storage [1, 21], photos [20, 28, 40], video [11, 41], health [12], financial [32], etc.

These services have two important requirements. First, we would like these services to be highly-available to provide correct service or data without interruption, and highly-durable to store data correctly for long durations spanning many years or even decades. Second, these services have to provide high performance –throughput [71], and latency [65]–to meet service level performance guarantees [1] in order to support applications using these services such as Amazon's S3 storage [1], Google's GMail [10], Microsoft's SkyDrive [21].

We need to overcome several challenges to meet these requirements. First, such a highly-reliable (highly-available and highly-durable) service has to be robust to broad range of failures such as media failures [105, 120, 133], software bugs [27, 106, 135], user errors [104, 117], administrator errors [6, 43, 73, 101], insider attacks [37, 51], malware threats [27, 125], geographic failures [9, 15], and organizational failures [34, 38]. Second, we have to provide better reliability with performance and cost (software, hardware, management, storage, network) comparable to that of existing commercial practice [2].

[1] At Amazon.com [65], an example SLA guarantees a service that provides a response time within 300ms for peak load of 99.9% of requests for a peak client load of 500 requests per second.

[2] For example, Google file system [71] uses three-way replication to protect data from failures.

The goal of our research is to facilitate the design and implementation of highly-reliable replicated systems to support these services. In this dissertation, as a step towards realizing this goal, we develop Byzantine fault tolerant replication techniques that can tolerate arbitrary failures while meeting the following practicality requirements: (1) high throughput [71], (2) low latency [65], (3) low system cost (hardware, software, network, management) [75, 102, 136].

1.1 Problem 1: High Availability

There is a large body of research that uses state machine replication [119] technique to improve availability of systems in the presence of failures. State machine replication technique replicates application state onto multiple servers (replicas) instead of single server to tolerate replica failures. Replicas co-ordinate to provide the same abstraction of centralized service of an unreplicated system to the end application. Some of these techniques [87, 90] can only tolerate a weaker set of failures that are caused by *fail-stop* or *benign* faults where faulty components fail by only stopping or by omitting some steps. Byzantine Fault Tolerant (BFT) state machine replication techniques [44, 45, 55, 63, 77, 111] can tolerate a stronger set of failures caused by *Byzantine faults* where faulty components can deviate from their specifications in arbitrarily bad ways. *Byzantine* faults subsume *benign* as well as *malicious* faults.

Three trends make Byzantine Fault Tolerant (BFT) replication increasingly attractive for building reliable and practical systems.

1. The mounting evidence of non-fail-stop behavior in real systems [37, 50, 51, 98, 101, 106, 123, 134, 135] suggest that BFT may yield significant benefits even without resorting to *n*-version programming [60, 81, 111].

2. The growing value of data [7, 12, 32, 114] and falling costs of hardware [8, 79] make it advantageous for service providers to trade increasingly inexpensive hardware for the peace of mind potentially

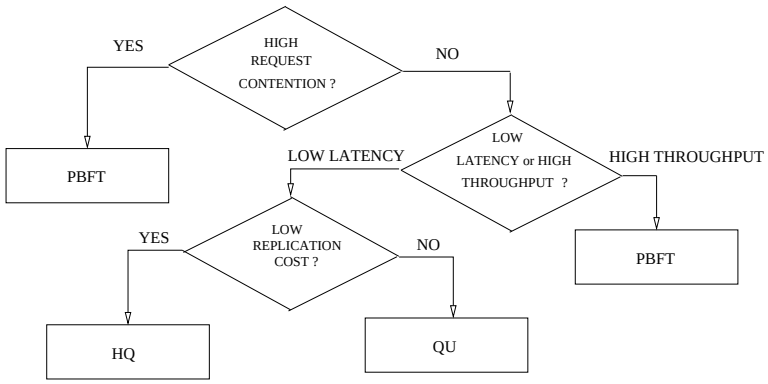


Figure 1.1: Complexity: Design space complexity of BFT replication technique with existing BFT protocols (PBFT [55],QU [45],HQ [63]).

provided by BFT replication.

3. The improvements to the state of the art in BFT replication techniques [45, 55, 63, 85, 111, 136] make BFT replication increasingly practical by narrowing the gap between BFT replication costs and costs already being paid for non-BFT replication. For example, by default, the Google file system uses 3-way replication of storage, which is roughly the cost of BFT replication for $f = 1$ failures with 4 agreement nodes and 3 execution nodes [136].

Challenges We have to address the following drawbacks of existing approaches using BFT state machine replication for using this technique to build practical systems with high availability.

- **BFT is complex:** Figure 1.1 (based on the analysis provided by cowling et al. [**?**]) captures the complexity of the state-of-the-art in BFT protocols where the system designers have to choose a protocol based on predicted workload and application characteristics. Such complexity represents a barrier to adoption of BFT techniques because it requires a system designer to choose the right technique for a workload and then for the workload not to deviate from expectations.

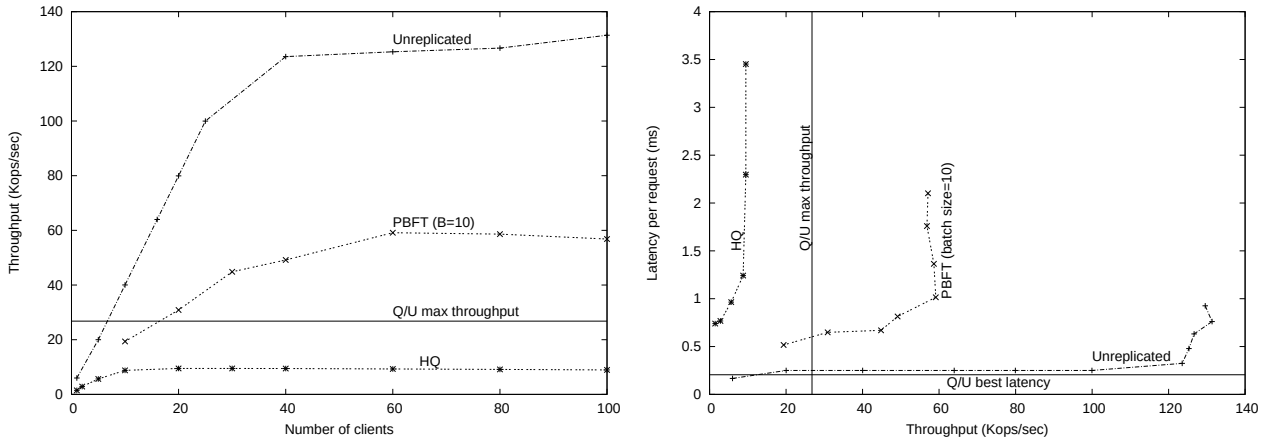


Figure 1.2: BFT protocol overhead: Performance comparison of state-of-the-art BFT protocols (PBFT [55], Q/U [45], HQ [63]) with unreplicated service. (a) Throughput versus clients: Peak throughput of unreplicated service is at least 2x better than PBFT, 4x better than Q/U, and 10x better than HQ (b) Throughput versus latency: With increasing load on the system, unreplicated service sustains lower latency for significantly higher throughput than existing BFT protocols.

- **BFT protocol overheads are significant:** Figure 1.2 suggests that BFT protocols impose significant overhead in peak throughput and latency compared to unreplicated service.

- **BFT limits application throughput:** The traditional BFT state machine replication technique [55] require non-faulty replicas to execute requests sequentially in the same order, completing execution of one request before beginning the execution of next one, to ensure that all non-faulty replicas are in a consistent state. This sequential execution of requests can severely limit the throughput of the applications–such as databases, file systems, and web servers-that are designed to achieve high throughput via concurrency. Figure 1.3(a) shows that such a limitation of traditional BFT replicated systems results in a significant loss of performance by not allowing replicas to execute requests concurrently.

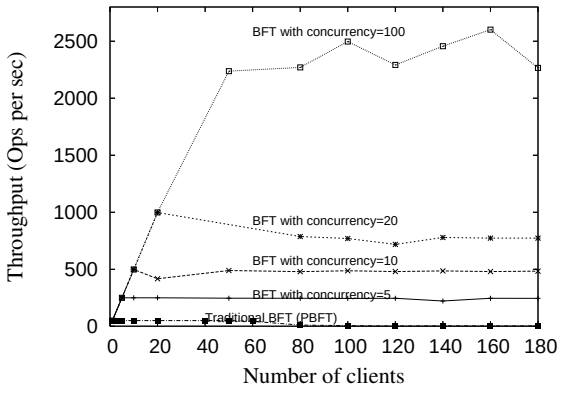


Figure 1.3: Application throughput: The traditional BFT state machine replication limits the throughput of replicated systems by its inability to execute application requests concurrently. We plot the measured throughput of a traditional BFT replicated system (PBFT [55]) that executes requests sequentially and compare it with the measured throughput of a hypothetical BFT system that can execute requests concurrently. We vary available concurrency (number of requests that can be executed concurrently) of the application using the sleep micro-benchmark [85]. The hypothetical BFT system provides significantly higher throughput than PBFT.

1.2 Problem 2: High Durability

The design of storage systems that provide data durability on the time scale of decades is an increasingly important challenge as more valuable information is stored digitally [14, 49, 114]. For example, data from the National Archives and Records Administration indicate that 93% of companies go bankrupt within a year if they lose their data center in some disaster [7], and a growing number of government laws [12, 32] mandate multi-year periods of data retention for many types of information [16, 104].

Against a backdrop in which over 34% of companies fail to test their tape backups [6] and over 40% of individuals do not back up their data at all [43], multi-decade scale durable storage raises two technical challenges. First, there exist a broad range of threats to data durability including media failures [105, 120, 133], software bugs [106, 135], malware [27, 125], user error [104, 117], administrator error [73, 101], organizational failures [34, 38], malicious insiders [37, 51], and natural disasters on the scale of buildings [9]

or geographic regions [15]. Requiring robustness on the scale of decades magnifies them all: threats that could otherwise be considered negligible must now be addressed. Second, such a system has to be practical with cost, performance, and availability competitive with traditional systems.

Storage outsourcing is emerging as a popular approach to address some of these challenges [75]. By entrusting storage management to a Storage Service Provider (SSP), where "economies of scale" can minimize hardware and administrative costs, individual users and small to medium-sized businesses seek cost-effective professional system management and peace of mind vis-a-vis both conventional media failures and catastrophic events.

Challenges Unfortunately, relying on an SSP is no panacea for long-term data integrity. SSPs face the same list of hard problems outlined above and as a result even brand-name ones can still lose data [13, 18]. To make matters worse, clients often become aware of such losses only after it is too late. This opaqueness is a symptom of a fundamental problem: SSPs are separate administrative entities and the internal details of their operation may not be known by data owners. While most SSPs may be highly competent and follow best practices punctiliously, some may not. By entrusting their data to back-box SSPs, data owners may free themselves from the daily worries of storage management, but they also relinquish ultimate control over the fate of their data. In short, while SSPs are an economically attractive response to the costs and complexity of long-term data storage, they do not offer their clients any end-to-end guarantees on data durability, which we define as the probability that a specific data object will not be lost or corrupted over a given time period.

1.3 Contributions

We design a reliable system that addresses the above problems by separating [77, 109] the two concerns of short-term availability (ability to provide service when desired) and long-term durability (ability to store data for longer durations). We fundamentally rethink BFT state machine replication techniques to provide high

availability while reducing complexity, reducing replication protocol overhead, and improving application performance by exploiting application concurrency. We use the principle of aggressive fault isolation to ensure high data durability despite conventional hardware and software faults, environmental disruptions, and administrative failures. In particular, this dissertation makes following key contributions in building highly available and durable systems.

1. **CBASE: High Execution Throughput Byzantine Fault Tolerance.** We propose a simple change to Byzantine Fault Tolerant state machine replication libraries in order to provide high throughput. Traditional state machine replication based Byzantine fault tolerant (BFT) techniques provide high availability and security but fail to provide high execution throughput. This limitation stems from the fundamental assumption of generalized state machine replication techniques that all replicas execute requests sequentially in the same total order to ensure consistency across replicas. CBASE is a high execution throughput Byzantine fault tolerant architecture that uses application-specific information to identify and concurrently execute independent requests. Our architecture thus provides a general way to exploit application parallelism in order to provide high throughput without compromising correctness.

   Although this approach is extremely simple, it yields dramatic practical benefits. When sufficient application concurrency and hardware resources exist, CBASE provides orders of magnitude improvements in throughput over BASE, a traditional BFT architecture. CBASE-FS, a Byzantine fault tolerant file system that uses CBASE, achieves twice the throughput of BASE-FS for the IOZone micro-benchmarks even in a configuration with modest available hardware parallelism.

2. **Zyzzyva: Speculative Byzantine Fault Tolerance.** We propose Zyzzyva, a BFT state machine replication protocol, that uses speculation to reduce replication overhead and simplify the design of BFT

state machine replication. In Zyzzyva, unlike in traditional BFT protocols [55, 85, 111], replicas speculatively execute requests without running an expensive agreement protocol to definitively establish the order. As a result, correct replicas' states may diverge, and replicas may send different responses to client libraries. Nonetheless, client's applications observe the traditional and powerful abstraction of a replicated state machine that executes requests in a linearizable [55] order because replies carry with them sufficient history information for client libraries to determine if the replies and history are stable and guaranteed to be eventually committed. If a speculative reply and history are stable, then client library passes the reply to the client application. Otherwise, the client waits until the system converges on a stable reply and history.

This approach allows Zyzzyva to reduce replication overheads to near their theoretical minima and significantly improve performance–throughput and latency–of the system. We implemented Zyzzyva replication library that provides a peak throughput that is within 35% of unreplicated service.

3. **SafeStore: A Durable and Practical Storage System.** We implement SafeStore, a distributed storage system, that is designed to provide long-term data durability despite conventional hardware and software faults, geographical catastrophes, and administrative failures caused by human error or malice. The architecture of SafeStore is based on fault isolation, which SafeStore applies aggressively along administrative, physical, and temporal dimensions by spreading data across autonomous storage service providers (SSPs). However, current storage interfaces provided by SSPs are not designed for high end-to-end durability.

   Safestore uses a new storage system architecture that (1) spreads data efficiently across autonomous SSPs using informed hierarchical erasure coding that, for a given replication cost, provides several additional 9's of durability over what can be achieved with existing black-box SSP interfaces, (2) performs an efficient end-to-end audit of SSPs to detect data loss that, for a 20% cost increase, improves

data durability by two 9's by reducing MTTR(mean time to recovery), and (3) offers durable storage with cost, performance, and availability competitive with traditional storage systems. We instantiate and evaluate these ideas by building a SafeStore-based file system with an NFS-like interface.

In conclusion, in this dissertation, we design and implement BFT replication techniques to support highly-reliable services by providing (1) high-availability with costs, latency, and throughput competitive with existing commercial practice, and (2) high-durability by tolerating failures due to broad range of threats over long durations.

1.4 Organization

In chapter 2, we present the system model and architecture. In chapter 3, we present CBASE, a high-throughput BFT architecture, that provides a general way to exploit application parallelism in order to provide high application throughput in BFT replicated systems. In chapter 4, we present Zyzzyva, a speculative BFT state machine protocol, that uses speculation to reduce replication protocol overheads and simplify the design of BFT replicated systems. In chapter 5, we describe SafeStore, a highly durable distributed storage system, that uses aggressive fault isolation to ensure long-term data durability. We present related work in chapter 6, and chapter 7 summarizes this dissertation.

Chapter 2

Byzantine Fault Tolerant State Machine Replication

This chapter provides a brief overview of Byzantine fault tolerant(BFT) state machine replication based approach to build reliable systems. Here we explain the system model, service properties, and architecture of existing BFT state machine protocols. We use the same system model throughout this dissertation unless otherwise stated.

2.1 System Model

State machine replication [119] is a general technique that can be used to replicate any service that can be modeled as a deterministic state machine replication. These services can have operations that perform arbitrary computations provided they are deterministic: the result and new state produced when an operation is executed must be completely determined by the current state and the operation arguments. However, some common forms of non-determinism in practical systems can be handled by transforming [57, 111] non-deterministic state machines into deterministic state machines by abstracting non-deterministic operations in a way that is not visible to the external world.

Such a replicated state machine provides the same service as unreplicated state machine but improves reliability by tolerating some number of faulty replicas. BFT state machine replication is a form of state machine replication that can tolerate *Byzantine* faulty replicas (described below).

The replicated service is implemented by n replicas. Client issues requests to the replicated service to invoke operations and wait for replies. Client and replicas are correct if they follow the BFT state machine

replication algorithm (PBFT [55], HQ [63]) used by the replicated service. The clients and replicas run on different nodes in an asynchronous distributed system where nodes are connected by unreliable network links. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order. We do not assume any bound on the relative processing speeds of the nodes.

BFT replication protocols [55, 63, 136] use public-key digital signatures (PK) to authenticate messages. These protocols implement a non-PK variant of the protocol that replaces [56] expensive public-key digital signatures with MAC(message authentication codes). In the public-key version of a protocol, any node can authenticate message by signing the message it sent. We denote a message X signed by principal (node or replica) Y's public key as $\langle X \rangle_{\sigma_Y}$. These protocols use cryptographic hash function D to compute message digests.

These protocols assume a Byzantine failure model where faulty nodes (clients and replicas) can deviate from the protocol specification arbitrarily. They can stop functioning, corrupt their internal replica state, send arbitrary messages, etc. These protocols also assume a strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. But they do however assume that the adversary cannot break cryptographic techniques like collision-resistant hashes, encryption, and signatures.

2.2 Service properties

BFT replication protocols [55, 63, 136], provide *safety* and *liveness* properties [93] assuming that no more than $\lfloor n-1/3 \rfloor$ replicas are faulty over the lifetime of the system.

The safety property [57] of BFT protocols ensure correct behavior of the replicated service in an asynchronous distributed system. BFT protocols provide a modified form of linearizability [76] (takes into consideration Byzantine faulty clients [57]) where the replicated service behaves like a centralized service

that executes requests atomically one at a time. In fail-stop model, safety can be guaranteed even when all replicas fail whereas in a Byzantine fault model safety requires a bound on the number of faulty replicas as they can behave arbitrarily bad. However, the traditional BFT protocols [55, 63, 136] tolerate optimal number of faults as it is impossible [57] to tolerate more than a third of faulty replicas. Safety is ensured regardless of the number of faulty clients (even when they collude among themselves or with faulty replicas) by ensuring that operations performed by faulty clients are seen in a consistent way by all non-faulty replicas. The damage that can be done by faulty clients is controlled using access control and authentication mechanisms before operations are invoked on the state machine.

The liveness property ensures that clients eventually receive replies from the replicated service and complete their operations. BFT protocols cannot guarantee liveness in an asynchronous distributed system as it is impossible [69] to implement consensus in such a system model. BFT protocols guarantee liveness during the intervals when the assumption of weak synchrony (such as bounded fair link [136]) holds where the messages are processed by the receiver within some fixed (but potentially unknown) worst case delay when they are sent (and retransmitted until the replies are recieved).

BFT replicated systems fail to provide correct service if some of these assumptions fail. For example, more than a third of replicas may fail due to correlated fault events such as administrative error [73, 101](if all these replicas belong to the same administrative domain), natural calamities [15](if all these replicas are co-located), software bug [106, 135] (if they use same version of the application code), media failures [105, 120, 133] (if they use same batch of disks from a single vendor) etc. Also, a recent study on malicious insider attacks [37, 51] suggest that a faulty client that has access to data shared by correct clients can delete or corrupt the data resulting in significant financial and other losses. The chances of such correlated faults happening is higher when such a replicated service is used to store data durably over long durations spanning many years or even decades. Such a durable storage service is required for applications that store digital

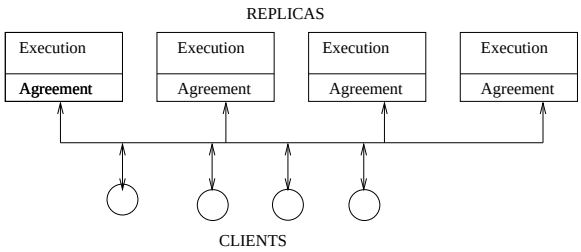


Figure 2.1: BFT State Machine Replication Architecture

archives, photos, health information, etc. In chapter 5, we present the design and implementation of a back-end durable storage system that can be used by BFT replicated systems to provide high data durability in the presence of such correlated replica failures or client failures.

2.3 BFT State Machine Replication Architecture

Figure 2.1 illustrates a typical BFT state machine replication architecture. Clients issue requests to the replicated service. Conceptually, replicas consist of two stages, an agreement stage and an execution stage. In reality, these two stages may be either tightly integrated on a single machine [55, 111] or implemented on different machines [136].

The agreement stage runs a distributed agreement protocol (such as such as three phase multicast protocol in PBFT [55]) to ensure that all non-faulty replicas eventually receive all the client requests and also agree on the request order in which the requests are delivered to the execution stage. Such an agreement protocol ensures that all non-faulty replicas agree on the same order despite upto $\lfloor n-1/3 \rfloor$ replicas can be Byzantine faulty. In chapter 4, we present the design and implementation of a new BFT agreement protocol that reduces replication overhead-performance and cost- and simplifies the design of BFT replicated systems.

The execution stage implements the application state machine and executes client requests in the order delivered by the agreement protocol. In chapter 3, we present the design and implementation of new BFT state machine replication architecture that improves the throughput of the execution stage.

Such a BFT architecture ensures the *safety* property because all non-faulty replicas start from the same initial state produce the same set of outputs and reach the same final state after executing the client requests in the same order sequentially.

Chapter 3

CBASE: High Execution Throughput Byzantine Fault Tolerance

In this chapter, we propose a high execution throughput Byzantine fault tolerant (BFT) architecture that uses application-specific information to identify and concurrently execute independent requests. Our architecture overcomes the limitation of existing BFT techniques by proposing a simple change to the BFT replication architecture that provides a general way to exploit application parallelism in order to provide high application throughput in the execution stage. Although this approach is extremely simple, it yields dramatic practical benefits.

We begin by providing some background in section 3.2, and then explain our approach in section 3.3. We explain the design and implementation of CBASE prototype based on this architecture and the BFT replicated network file system (CBASE-FS) in section 3.5. In section 3.6, we present the evaluation of our CBASE prototype as well as CBASE-FS file system to demonstrate the practical benefits.

3.1 Introduction

Recent work on Byzantine fault tolerant (BFT) state machine systems has demonstrated that generalized state machine replication can be used to improve robustness [45, 55, 63] and confidentiality [136] of a service in the presence of Byzantine failures due to hardware crashes [105, 120, 133], software bugs [27, 106, 135], operator errors [6, 43, 73, 101], and malicious attacks [27, 37, 51, 125]. Furthermore, this work suggests that this approach can be used to build practical systems as it adds low latency overhead [55, 111, 136], can recover proactively from faults [56] and make use of multiple existing off-the-shelf implementations [111] to avoid correlated failures, and can minimize replication of the application-specific parts of the system [136].

However, current BFT state machine systems can fail to provide high throughput. They use generalized state machine replication techniques that require all non-faulty replicas to execute all requests sequentially in the same order, completing execution of each request before beginning execution of the next one. This sequential execution of requests can severely limit the throughput of systems designed to achieve high throughput via concurrency [130]. Unfortunately, this concurrency-dependent approach lies at the core of many (if not most) large-scale network services such as file systems [47], web servers [130], mail servers [115], and databases [111, 127]. Furthermore, technology trends generally make it easier for hardware architectures to scale throughput by increasing the number of hardware resources (e.g., processors, hardware threads, or disks) rather than increasing the speed of individual hardware elements. Although current BFT systems like PBFT [55] and BASE [111] implement optimizations such as request batching in order to amortize their replication overheads due to agreement overheads, sequential execution of requests still imposes a fundamental limitation on application-level concurrency.

We address this problem by introducing a simple addition to the existing BFT state machine replication architectures that allows throughput of the system to scale with application parallelism and available hardware resources. Our architecture separates agreement from execution [136] and inserts a general parallelizer module between them. The parallelizer uses application-supplied rules to identify and issue concurrent requests that can be executed in parallel without compromising the correctness of the replicated service. Hence, the throughput of the replicated system scales with the parallelism exposed by the application and with available hardware resources. More broadly, in our architecture replicas execute requests according to a partial order that allows for concurrency as opposed to the total order enforced by traditional BFT architectures.

We demonstrate the benefits of our architecture by building and evaluating a prototype library for constructing Byzantine fault-tolerant replicated services called CBASE (Concurrent BASE). CBASE extends

the BASE system [111] which uses the traditional BFT state machine replication architecture. We use a set of micro-benchmarks to stress test our system and find that when sufficient application concurrency and hardware resources exist, CBASE provides orders of magnitude improvements in throughput over the traditional BFT architecture. We also find that for applications or hardware configurations that can not take advantage of concurrency, CBASE adds little overhead compared to the optimized BASE system. As a case study, we implement CBASE-FS, a replicated BFT file system, to quantify the benefits for a real application. CBASE-FS achieves twice the throughput of BASE-FS for the IOZone micro-benchmarks even in a configuration with modest available hardware parallelism. When we artificially simulate more hardware resources, CBASE's maximum write throughput scales by over an order of magnitude compared to the traditional BFT architecture.

The main contribution of this study is a case for changing the standard architecture for BFT state machine replication to include a parallelizer module that can expose potentially concurrent requests to enable parallel execution. Based on this study, we conclude that this idea is appealing for two reasons. First, it is simple. It requires only a small change to the existing standard BFT replication architecture. Second, it can provide large practical benefits. In particular, this simple change can improve the throughput of some services by orders of magnitude, making it practical to use BFT state machine replication for modern commercial services that rely on concurrency for high throughput.

The main limitation of this approach is that safely executing multiple requests in parallel fundamentally requires application-specific knowledge of inter-request dependencies. But, we do not believe this limitation undermines the argument for adding a parallelizer model to BFT state machine replication libraries. In particular, our prototype parallelizer implements a set of default rules that assume that all requests depend on all other requests. Applications that are satisfied with sequential execution can simply leave these default rules in place, and applications that desire increased throughput can override these rules to expose their

concurrency to the replication library. Furthermore, designers of such applications can take an iterative approach, first developing simple rules that expose some application concurrency and later developing more sophisticated rules that expose more concurrency if required for performance.

The rest of this chapter proceeds as follows. Sections 4.2 and 3.2 outline our system model and review the standard architecture for existing BFT state machine replication systems. Then Section 5.2 describes our proposed architecture and Section 3.5 describes our prototype replication library, CBASE. Section 5.6 discusses our experimental evaluation, Section 4.6 discusses related work, and Section 5.8 summarizes our conclusions.

3.2 Background: BFT systems

BFT state machine replication [55, 56, 111] based systems provide high availability by replicating the server and use a distributed algorithm to coordinate the replicas. Such a system provides *safety* and *liveness* guarantees while tolerating no more than a third of faulty replicas ($\lfloor (n-1)/3 \rfloor$ faulty replicas where n is the total number of replicas). Safety requires that the replicated service provides linearizability(modified to account Byzantine-faulty clients [57]) where the service behaves like a centralized implementation that executes requests atomically one at a time. Liveness requires that the correct clients eventually receive replies to their requests.

Figure 3.1 illustrates a typical BFT state machine replication architecture. Clients issue requests to the replicated service. Conceptually, replicas consist of two stages, an agreement stage and an execution stage. In reality, these two stages may be either tightly integrated on a single machine [55, 111] or implemented on different machines [136]. The agreement stage runs a distributed agreement protocol to agree on the order of client requests and the execution stage executes all of the requests in the same order.

Each execution node maintains a state machine that implements the desired service. A state machine

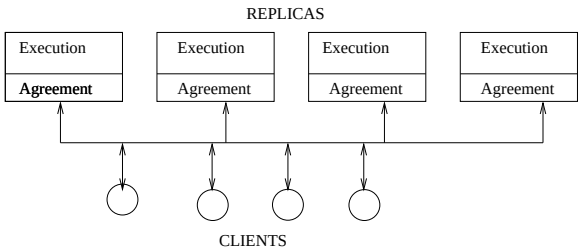


Figure 3.1: Traditional BFT Architecture

consists of a set of state variables that encode the machine's state and a set of commands that transform its state. A state machine takes one or more of the following actions to execute a command:

1. Read a subset of the state variables, called the read-set R.
2. Modify a subset of the state variables, called the write-set W.
3. Produce some output O to the environment.

A command is non-deterministic if its write-set values or output are not uniquely determined by its input and read-set values; otherwise it is deterministic. A state machine is called a deterministic state machine if all commands are deterministic. For safety, all non-faulty replicas starting from the same state should produce the same set of outputs and reach the same final state after executing the same set of requests from clients. Traditional state machine replications assume deterministic state machines or use deterministic methods to work around [55] non-determinism in the applications. The following requirements [119] ensures safety of a replicated system:

Schnieder's classical technique [119] for constructing deterministic replicated state machines ensure safety by enforcing:

1. **Agreement:** *Every non-faulty state machine replica receives every request*

2. **Order:** *Every non-faulty state machine replica processes the requests it receives in the same relative order*.

Traditional BFT state machine replication approaches [55] provide safety in an asynchronous system model where network may fail to deliver messages, delay them, duplicate them, or deliver them out of order, and there is no bound on the difference in computational speeds of nodes on which state machines are replicated. However, it is impossible [69] to guarantee liveness in a truly asynchronous system. Hence, these systems guarantee liveness during the intervals when the assumption of weak synchrony (formally defined as bounded fair links [136]) holds where the messages are processed by the reciever within some fixed (but potentially unknown) worst case delay when they are sent and potentially retransmitted until they are recieved.

Although this approach can provide high-availability by tolerating faults, it can fail to provide high throughput because the Order requirement does not, in general, allow replicas to execute requests concurrently. In particular, unless strong assumptions are made about the state machine's internal implementation, execution nodes must finish executing request i before executing request $i+1$. Otherwise, concurrency within a state machine could introduce non-determinism into the system, which could cause different replicas' state to diverge.

3.3 High Throughput BFT State Machine Replication

Figure 3.2 illustrates our high throughput state machine replication architecture, where we maintain the separation between the agreement and execution stages and introduce a *parallelizer* between them. The parallelizer takes a totally ordered set of requests from the agreement stage and uses application-supplied rules to first identify independent requests and then issue them concurrently to the execution stage. A thread pool or event based architecture [130] in the execution stage can then execute the requests in parallel to improve system throughput.

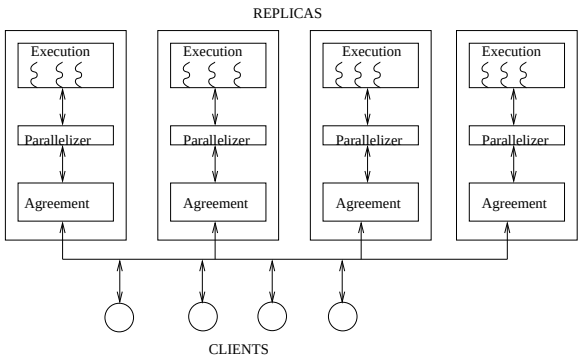


Figure 3.2: CBASE: High execution throughput BFT state machine replication architecture

3.3.1 Relaxed Order and Parallelizer

The key idea of high throughput state machine replication is to relax Schneider's *Order* [119] requirement on state machine replication (defined above) to allow concurrent execution of independent requests without compromising safety.

We say that two requests are *dependent* if the write-set of one has at least one state variable in common with the read-set or write-set of the other. More formally, we define dependence as follows: Request r_i, with read-set R_i and write-set W_i and request r_j, with read-set is R_j and write-set W_j, are *dependent requests* if any of the following conditions is true (1) $W_i \cap W_j \neq \phi$, (2) $W_i \cap R_j \neq \phi$, or (3) $R_i \cap W_j \neq \phi$.

Given this notion of dependence, we refine Schneider's *Order* requirement for replicated state machine to ensure high throughput. The modified requirements that ensures safety while providing high throughput are as follows:

1. **Agreement:** *Every non-faulty state machine replica receives every request*

2. **Relaxed order:** *Every non-faulty state machine replica processes any pair of dependent requests it receives sequentially and in the same relative order.*

Relaxed order provides the same safety property of traditional state machine replication for two reasons. First, like traditional state machine replication, *Relaxed Order* ensures that *dependent requests* are executed strictly in the same order at all non-faulty replicas by following the *total order* provided by the agreement stage. Second, unlike traditional state machine replication, *relaxed order* allows concurrent execution of *concurrent requests* without affecting safety because these requests can be commuted safely as they modify disjoint sets of state variables. The result of executing concurrent requests in any order places the system in the same final state with the same output visible to the external world[1].

Notice that under the Relaxed Order requirement, concurrent requests can be processed in parallel. Thus, with the Relaxed Order requirement, all non-faulty replicas execute requests in the same *partial order* as opposed to the traditional architecture where all correct replicas execute requests in the same *total order*.

In the CBASE architecture, the parallelizer uses application-specific information to take advantage of the Relaxed Order requirement. The parallelizer transforms a totally ordered schedule of requests provided by the agreement protocol into a partially ordered schedule based on application semantics.

A *sound parallelizer* with following properties meets the safety requirements of state machine replication:

1. **Partial order:** *For any two requests r_i and r_j such that r_i and r_j are dependent and r_i precedes r_j in the total order established by the agreement stage, parallelizer completes executing request r_i before it begins to execute request r_j.*

2. **Non-blocking:** *The parallelizer eventually executes a pending request that is not dependent on any other preceeding request.*

[1] Like traditional state machine replication technique, we assume that replicas are non-deterministic in nature or handle non-determinism [55, 111] in a way that is not visible to the external world.

The *Partial order* property ensures that the non-faulty replicas meet the *relaxed order* requirement. The *Non-blocking* property of the parallelizer ensures liveness of the system.

Notice that there are two properties that are *not* required of a parallelizer. First, we do not require *precision*: a sound parallelizer may enforce additional ordering constraints on requests beyond those required by the partial order property. This non-requirement is important because it allows us to simplify the design of parallelizers for complex applications by building *conservative* parallelizers that can introduce false dependencies between requests. For example, in Section 3.5.3 we describe a simple NFS implementation that uses a conservative analysis to identify some, but not all, concurrent requests. Second, we do not require *equality*: different correct parallelizers may enforce different partial orders as long as all correct parallelizers' partial orders are consistent with the order required by the partial order property. One could, for example, implement multiple versions of the parallelizer for an application to prevent any one implementation from being a single point of failure [128].

3.4 Safety and Liveness properties

Theorem 1. *The properties of existing BFT agreement protocol [55] and of a sound parallelizer ensure the safety and liveness properties of a replicated service.*

Safety proof: The existing BFT agreement protocol [55] used in the agreement stage guarantees that the client requests are ordered in the same *total order* at all non-faulty replicas while tolerating upto f replica failures in the system. From the *partial order* property of a sound parallelizer, all non-faulty replicas order the *dependent requests* in the same order that follows the total order decided by the agreement protocol. Faulty replicas cannot affect the order of *dependent requests* at non-faulty replicas. Hence, all non-faulty replicas execute the dependent requests in the same order and satisfy the *Relaxed Order* property which ensures safety property of the replicated system.

Liveness proof: If the traditional BFT system comprising of the agreement and execution stages is live, then the high-throughput BFT system comprising of the agreement, execution, and parallelizer is also live. The liveness property of the agreement stage ensures that all client requests will be eventually delivered in the same *total order* to all non-faulty replicas. From the *non-blocking* property of the parallelizer, non-faulty replicas execute the first request eventually as there is no request in the total order that the first request is dependent on. By applying *non-blocking* and *partial order* properties of a *sound parallelizer* recursively on subsequent requests after the execution of preceeding requests, starting from the first request, we can prove that all clients requests are executed eventually at all non-faulty replicas and replies are sent back to the client eventually.

3.4.1 Advantages and Limitations

The high throughput state machine replication architecture has two potential advantages. First, it can support high-throughput applications. If the workload contains independent requests and the system has enough hardware resources, then independent requests can be executed concurrently by the execution stage to improve the throughput of a system. Second, it is simple and flexible. In particular, to achieve high throughput, we do not change any of the other components in the system like client behavior, the agreement protocol, or the application. These components can therefore be changed to suit the requirements of the replicated system. For example, one can change the agreement protocol and client side behavior to build a system that either tolerates Byzantine failures or fail-stop failures while achieving high throughput without modifying the parallelizer.

The main limitation of a system using this architecture is that the rules used by the parallelizer to identify dependent requests require knowledge of the inner workings of each application. In many ways, this knowledge is similar to that required to build the abstraction layer used in BASE to mask differences in different

implementations of the same underlying application [111]. However, it may in general be difficult to know what internal state a given request affects or to determine with certainty whether any given pair of requests are dependent.

Fortunately, it is not necessary to completely understand the inner workings of an application in order to define a parallelizer for it. In particular, it is always permissible to define *conservative* rules that include all true dependencies but also include some false dependencies. System designers may choose to follow an incremental approach by first defining a set of simple but conservative rules to identify "obvious" concurrent requests and then progressively refine the rules if more parallelism is needed to meet performance goals.

3.5 CBASE Prototype

The goal of our prototype is to demonstrate a general way to extend state machine replication systems in order to allow concurrent execution of requests for applications that can identify dependencies among requests.

Our prototype, CBASE (Concurrent BASE) system extends the BASE [111] system, which is based on traditional state machine replication, to use the high throughput state machine replication architecture described in the previous section.

CBASE modifies BASE to cleanly separate [136] the agreement and execution stages[2] and introduces a parallelizer between these stages as shown in Figure 3.2. CBASE's single threaded agreement module uses BASE's 3-phase atomic multicast protocol to establish a total order on requests. The CBASE parallelizer uses an application-specific set of rules to extract *concurrent* requests and execute them in parallel. The CBASE parallelizer guarantees the safety and liveness properties by ensuring *partial order* and *non-blocking* properties of a sound parallelizer as defined in section 3.3.1.

[2]Note, however, that our prototype implementation does not allow the agreement and execution modules to run on different sets of machines.

The CBASE parallelizer uses a *dependency graph* to ensure the *partial order* property. The requests are populated in the *dependency graph* using the total order in which requests are delivered by the agreement stage and the application-specific rules to establish *dependence* relation (as defined in section **??**) among requests. Every request is assigned a vertex in the *dependency graph* as soon as it is delivered by the agreement stage with a totally ordered sequence number. A directed edge from a request r_i to a request r_j exists in the *dependence graph* iff it satisfies the following two conditions: (1) $i > j$ (that is r_j preceeds r_i in the total order assigned by the agreement stage) and (2) r_i and r_j are *dependent requests* as defined in section 3.3.1. Such a *dependency graph* forms a DAG (directed acyclic graph) because the edges are directed and the first condition that uses the total ensures that there can be no cycles. The *dependency graph* represents the partial order schedule for the requests.

The CBASE parallelizer uses a *non-blocking scheduler* to execute the requests in the partial order schedule defined by the *dependency graph* and there by ensures the *partial order* property of a sound parallelizer. A request is said to be not blocking on any preceeding request if its vertex has no outgoing edges to other vertices (requests) in the *dependence graph*. The *non-blocking scheduler* of the parallelizer executes a request if it is not blocking. The *non-blocking scheduler* executes a request by assigning it a thread among the worker thread pool in the execution stage. Hence, the scheduler can execute *concurrent* requests in parallel by assigning different threads in the pool. The worker thread updates the *dependency graph* after a executing a request by removing the corresponding vertex and all the directed edges incident onto this vertex. Thus, it unblocks all the requests that are blocked on this request in the partial order and allows the *non-blocking scheduler* to execute new set of requests that are not blocked by other preceeding requests in the *dependency graph*. The *non-blocking scheduler* guarantees the *non-blocking* property of a sound parallelizer by executing outstanding requests (that are not blocked on preceeding requests) in the order they are delivered by the agreement stage. Such a scheduling scheme guarantees liveness by ensuring that a request that is not

dependent on any preceeding request will be executed eventually.

The default behavior of the parallelizer is to treat all the requests as dependent, in which case it behaves like the existing BASE system where the requests are executed sequentially. This default behavior can be used when the state machine is treated as a black box or where dependencies across requests cannot easily be inferred. The rules in the parallelizer can be incrementally refined by taking a conservative approach where the requests known to touch different states can be treated as independent and all the other requests can be treated as dependent. Similarly, for backwards compatibility with existing state machines, if a state machine is not thread safe we can just have a single worker thread or implement a mutual exclusion lock around the state machine.

3.5.1 Parallelizer interface

The parallelizer appears to the agreement and execution threads as a variation of a producer/consumer queue. When a consumer thread asks for a request, the parallelizer searches for a request that is not dependent of all incomplete preceding requests and returns one if found; otherwise it blocks the consumer thread until a request becomes independent. The detailed description of parallelizer interface used by agreement and execution stages is described in [84] and we just list them here for brevity.

- Parallelizer.insert(): Called by the agreement stage to enqueue a request when the request is committed in the agreement stage.
- Parallelizer.next_request(): Called by the execution stage to fetch an independent request.
- Parallelizer.remove_request(): Called by the execution stage after the execution of a request is completed to delete request state in the parallelizer.

- Parallelizer.sync(): This interface supports replica state checkpointing required by the BASE system [111]. The agreement stage updates the next checkpoint sequence number by calling this function as soon as the current checkpoint is complete.

3.5.2 Dependence Analysis

The parallelizer's goal is to determine if a new request is dependent on any pending request using application-specific rules. The parallelizer design must balance three conflicting goals: (1) Generality – the parallelizer should provide an interface that allows a broad range of applications to encode rules for detecting dependencies among their requests; (2) Simplicity – the interface for specifying these rules should be simple to reduce the effort and likelihood of error in dependency-rule specification; and (3) Flexibility – the interface should allow specification of simple conservative dependency rules and progressive refinement to more precise dependency rules that expose more concurrency. Notice that our design is a compromise among these design goals and that other algorithms for identifying dependencies among requests could be explored in future work.

In the CBASE prototype, conflict detection between a pair of requests depends on the *functions* they invoke and the *arguments* they pass. An application that has F distinct function entry points provides the parallelizer with following four application-specific functions and rules for conflict detection:

1. A *request parser* that takes an application request and produces a function ID and an argument object.

2. An *operator concurrency matrix OCM* that identifies pairs of functions that are considered to be in conflict *independent* of the arguments to the functions. *OCM* is an FxF matrix, where $OCM[i, j]$ is true if a request invoking function i and a request invoking function j are always considered to be dependent. This dependency may be because these functions always access common state with one

of them updating that state, or this dependency may be because these functions sometimes access common state and a conservative design assumes they always do for simplicity or because more careful analysis of arguments is impractical or unnecessary for the application.

3. An *argument analysis function AAF* that takes two argument objects and returns true if an analysis of the arguments indicates that functions that are not flagged by the *OCM* may access common state when supplied with these arguments. More precisely, $AAF(a_1, a_2)$ must return true if there exists any pair of functions f_1, f_2 such that $OCM[f_1, f_2] = false$ but $f_1(a_1)$ and $f_2(a_2)$ access common state and either modifies that common state.

4. An *operator+argument concurrency matrix OACM* that identifies pairs of functions that are considered to be in conflict only when an analysis of the arguments indicates that they may access common state.

When a new request r_j calling function f_j with arguments a_j arrives, the parallelizer compares it to each pending request r_i calling function f_i with arguments a_i as follows. First, it checks for argument-independent dependencies using an application-specific operator concurrency matrix (*OCM*): if $OCM[f_i, f_j]$ is true, the requests are dependent. If not, then it checks to see if the arguments indicate that there may be additional risk of dependencies using an argument analysis funtion (*AAF*) : if $AAF(a_i, a_j)$ is true, then it also checks for argument-dependent dependencies and identifies a dependency between r_i and r_j if $OACM[f_i, f_j]$ (operator+argument concurrency matrix) is true. Finally, if $OCM[f_i, f_j]$ is false and either $AAF(a_i, a_j)$ is false or $OACM[f_i, f_j]$ is false, then no dependency between r_i and r_j exists. Please refer to [84] for a detailed description of dependence analysis.

This structure facilitates a 2-level analysis in which the operator concurrency matrix *OCM* defines broad rules where no argument analysis is attempted or needed and in which the operator+argument concurrency

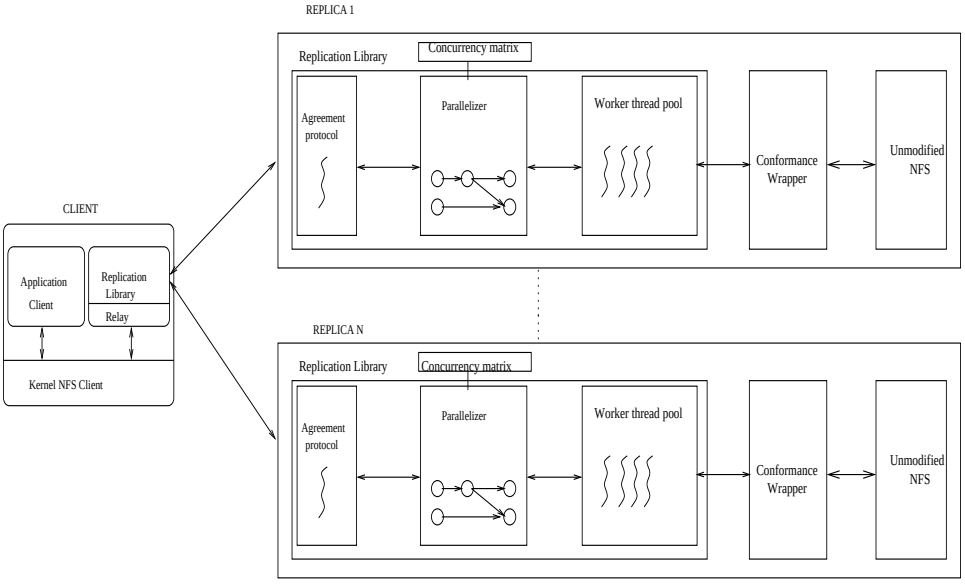


Figure 3.3: CBASE-FS: High throughput Byzantine fault tolerant NFS

matrix *OACM* defines more precise rules that are invoked after an analysis of the arguments indicates that two calls that sometimes are independent may be in conflict due to their arguments. The next subsection describes our NFS file system prototype where we use the *OACM* to encode rules for functions if the state they affect is easily identified from file handles in their arguments and where we use the *OCM* to handle other functions.

3.5.3 Example Service: NFS

We have implemented CBASE-FS, a Byzantine fault tolerant NFS [25] using CBASE as shown in Figure 3.3. Our implementation builds on BASE-FS [111], which uses existing implementations of NFS to implement each instance of the replicated state machine. In particular, a client in CBASE-FS mounts the replicated file system exported by the replicas as a local NFS file system [25]. Unmodified applications access the file system using standard file system calls. The local kernel sends NFS calls to the local user-level NFS server,

which acts as a wrapper for CBASE-FS by calling the *invoke* procedure of the BASE replication library to relay the request to the replicas. This procedure returns when the wrapper receives $f+1$ matching replies from different replicas.

The agreement stage in CBASE establishes a total order on requests and then sends each ordered request to the parallelizer. The parallelizer updates the dependency graph using NFS's concurrency matrix as defined in section 5.3.1 whenever a request is enqueued. The worker threads in the execution stage dequeue independent requests and execute the requests.

CBASE-FS uses BASE's [111] abstraction layer (conformance wrapper) to resolve non-determinism in NFS such as file handle assignment or timestamp generation. Additionally, CBASE introduces a new source of non-determinism due to concurrent execution of NFS *create* operations to different files. The existing BASE conformance wrapper at different replicas could return different file handles based on the order of execution of these requests. We fix this problem by having a rule in the concurrency matrix to treat the requests with create/delete operations as always dependent.[3]

Concurrency Matrix for NFS

For NFS, we keep the classification simple by just looking at file handles, and thus must have conservative rules for some of the operations. Our argument analysis function (AAF) defines two arguments as related if they include a common file handle. We present the key rules that are used in defining NFS's argument-independent operator concurrency matrix (OCM) and argument-dependent operator+argument concurrency matrix (OACM) below. We have the complete description of concurrency matrices in appendix A.

[3]We speculate that additional concurrency could be exposed by including constraints based on a request's total-order sequence number to the conformance wrapper's file handle generation logic and the parallelizer's dependency logic.

- getattr and null requests are read only requests and hence are independent for both related and unrelated arguments.
- Reads to different files are independent whereas reads to the same files are dependent. Reads modify the last-accessed-time attribute of a file, so we do not concurrently execute read requests to the same file.
- Writes to different files are independent and writes to the same file are dependent. Reads are dependent on writes to the same file and vice-versa.
- All create and remove operations to the same file or different files are dependent as they introduce non-determinism if executed concurrently.
- Create/Rename/Remove operations are always treated as dependent on Read or Write operations. Read/Write operations carry the file handle of the file whereas create/rename/remove requests carry the file handle of the directory in which file is present and the filename of the file to be deleted. As we just look at the file handle to decide if two arguments are related or not, we cannot execute the requests with create/rename/remove concurrently with read/write requests.

We give up some potential concurrency across requests with these conservative rules. Looking at other fields in the request apart from file handle and keeping additional state about file handles could allow for more sophisticated and accurate classification. There is a tradeoff between on one hand the simplicity of the design and the time spent to classify requests versus on the other hand the amount of concurrency realized by the parallelizer. This trade-off should be explored in more detail in the future.

3.5.4 Additional Optimizations

In order to improve throughput CBASE supports some of the optimizations introduced by PBFT [55] such as reduced communication, request batching, read-only optimization . However, CBASE does not support tentative execution as it is shown in [57] that this optimization has little impact on throughput when used along with request batching and that it adds complexity to the code to keep uncommitted state in the system.

3.6 Evaluation

A high throughput BFT system should achieve two goals: (1) when there is application parallelism and hardware concurrency it should provide high throughput compared to traditional BFT system, and (2) when there is no parallelism in the application or when there are limited resources it must have low overhead.

All experiments run with 4 replicas and the system tolerates one Byzantine fault. Replicas run on single processor machines with 933 MHZ PIII processor and connected by a 100 Mbit ethernet hub. All the machines have 256MB of memory except for one that has 512MB of memory. The experiments run on an isolated network. We use 5 client machines to load the system. Client machines are connected to the network through the same ethernet hub as the replicas. Two of the client machines have 933 MHZ PIII processor with 512MB of memory and the other three machines have 450 MHZ PIII processor with 128KB of memory. All machines run Redhat Linux 7.2.

3.6.1 Micro-Benchmark

The micro-benchmark compares the performance of BASE and CBASE executing a simple, stateless service where clients send null requests to which the server reply with null results. We show that for our microbenchmark CBASE imposes little additional latency or overhead compared to BASE and that CBASE's throughput scales linearly with application parallelism and available hardware resources.

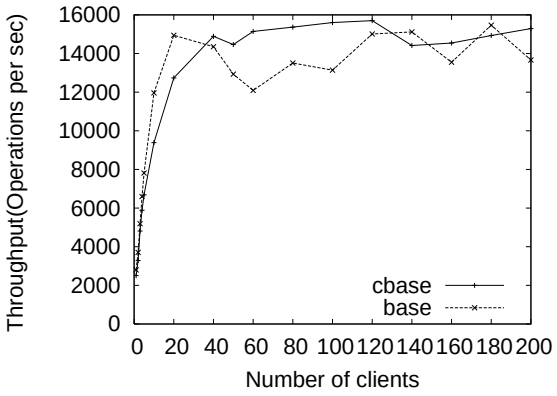
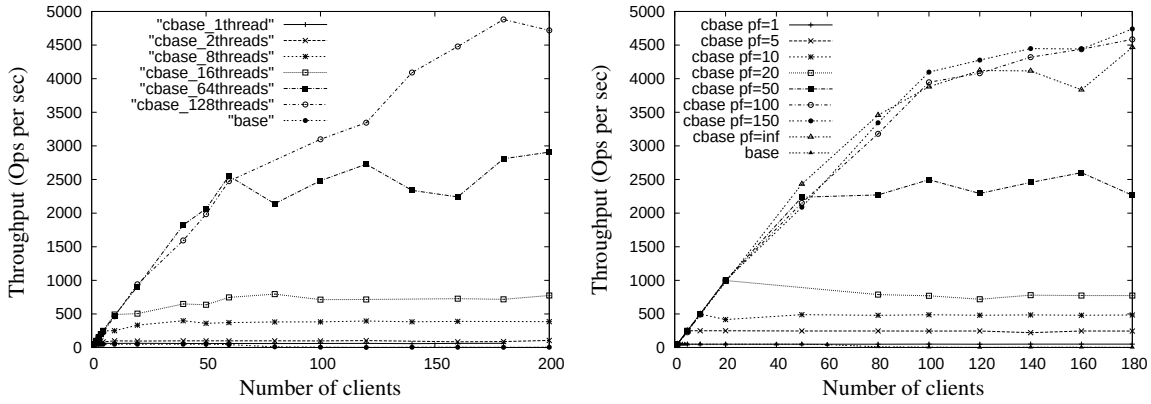


Figure 3.4: Overhead of CBASE versus BASE

Overhead

Figure 3.4 compares the overhead of BASE and CBASE by running the baseline benchmark configured with infinite application concurrency (no shared state across requests) and minimal hardware demand per request (each application request at the server simply returns immediately). BASE is CPU-limited—a small number of clients saturate the CPU, but BASE allows throughput to reach a peak of about 15,000 requests per second by employing agreement-stage batching [56], yielding a CPU overhead of less than 100 μs per request. CBASE runs with 16 execution threads and BASE runs with 1 thread. All points in the graph are averages of 3 runs with variance of less than 15%. The CBASE parallelizer treats all requests as independent, but limited hardware resources limit the benefits gained by concurrency—requests run on a uniprocessor and return immediately. Figure 3.4 shows that the lines representing CBASE and BASE closely follow each other illustrating that CBASE introduces little overhead when there is no scope for concurrent execution of requests.



(a) (b)

Figure 3.5: Scalability of throughput: (a) With varying hardware resources (b) With varying levels of application parallelism where *parallelism factor* is varied from minimum(pf=1) to infinity(pf=inf).

Scalability of throughput with application parallelism and resources

The throughput of a service depends both on the parallelism present in the application and on the hardware resources (e.g., processors, disks, bandwidth) available to the system. In this set of experiments, we evaluate the scalability of throughput with varying application parallelism and hardware resources.

First, we evaluate the ability to scale throughput with resources. We simulate accesses to a varying array of parallel disks by running the benchmark with the modification that the code to process each request sleeps for 20ms before returning a reply. The CBASE parallelizer assumes infinite parallelism in the application and considers all requests to be independent. We simulate varying "disk" resources by configuring CBASE to run with varying numbers of execution threads. We note that BASE still runs with a single thread since it never attempts to issue more than one request to the execution stage at a time. Figure 3.5(a) shows that the throughput of BASE saturates at 50 ops/sec (as expected with 20ms service time for each operation) which matches the throughput of CBASE running with 1 thread. The throughput of CBASE increases with

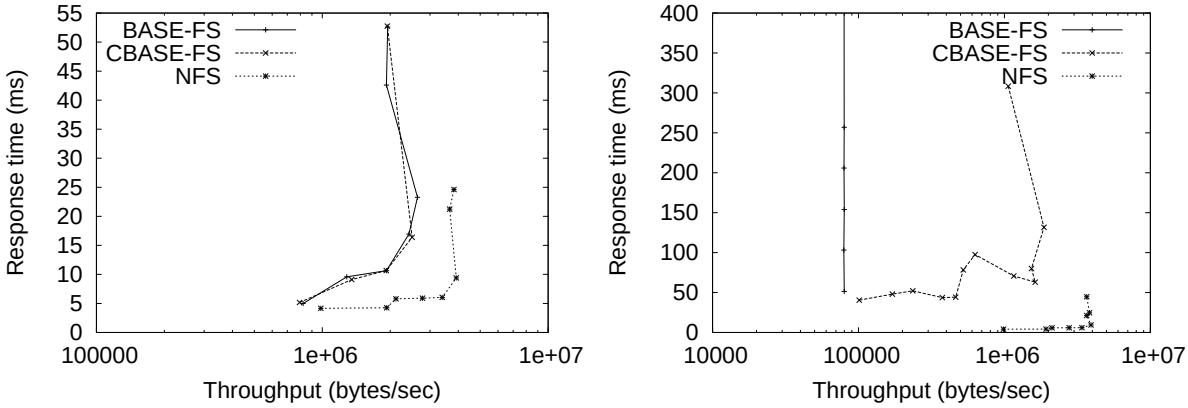
the number of clients but eventually saturates because increasing the number of clients improves concurrency only if throughput is limited by the available hardware resources. As the number of "disks" (threads) increases, the throughput of CBASE increases nearly linearly—128 "disks" reach a throughput of 4700 requests/second.

Next, we evaluate the scalability of throughput with parallelism in the application. We run the same experiment as above except that we fix the number of resources in this experiment and vary parallelism in the application. We emulate 100 resources by fixing the number of CBASE execution threads to 100. We define the *parallelism factor* as the number of requests that we allow to be executed concurrently, and simulate varying application parallelism by varying this parameter. The parallelizer randomly assigns each incoming requests to one of *parallelism factor* buckets and creates dependencies among all requests to the same bucket, allowing only a fixed number of requests to be independent at any point of time. Figure 3.5(b) shows that the throughput of BASE saturates at 50 ops/sec and that CBASE matches this performance when the application *parallelism factor* is 1. CBASE's maximum throughput increases almost linearly with increasing *parallelism factor* up to 100. The throughput of CBASE does not improve beyond a *parallelism factor* of 100 because it is limited by the 100 simulated hardware resources.

Notice that when application parallelism and hardware resources are available, CBASE's throughput can exceed BASE's by orders of magnitude.

3.6.2 NFS Micro-Benchmarks

In this subsection, we evaluate the performance of CBASE-FS, a replicated NFS that uses CBASE. We compare the performance of CBASE-FS with BASE-FS and unreplicated NFS.



(a) (b)

Figure 3.6: Throughput versus response time (a) With 4KB writes to evaluate CBASE protocol overhead (b) With 4KB writes and artifical delay to evaluate benefits of pipelining in CBASE

Local disk

In this benchmark, each client writes 4KB of data to a different file in a directory exported by the file system. We vary the number of concurrent clients and measure the response time and throughput of the system. As described in Section 5.2, requests to different files are generally treated as independent requests by the CBASE parallelizer. CBASE-FS runs with 16 threads and unreplicated NFS runs with 16 daemon processes. In all file system instances, NFS servers write asynchronously to the disk.

Overhead Figure 3.6(a) plots the response time versus the throughput of CBASE-FS, BASE-FS, and unreplicated NFS. CBASE-FS and BASE-FS closely follow each other and their throughput saturates around 2.5MB/sec, whereas the throughput of NFS saturates around 4MB/sec. In this experiment, because servers run on a uniprocessor system and write asynchronously to the local disk there is little or no benefit for concurrently executing the requests because the threads write in the file buffer cache in memory and rarely

block. Hence we show that CBASE-FS performs as well as BASE-FS and adds little or no overhead when there is no scope in concurrency. The maximum throughput of BASE and CBASE is within a factor of 2 compared to unreplicated NFS; the difference stems from the extra overhead of processing protocol messages, additional cryptographic computations, and extra kernel crossings. For similar reasons, NFS also yields less latency than BASE and CBASE.

Benefits of Pipelining with artificial delay In this experiment we evaluate the performance when there is scope for concurrent execution of requests. We simulate this scenario by making BASE and CBASE servers sleep for 20 ms after writing to a file and before sending a reply to the client. Figure 3.6(b) shows the response time plotted against the throughput of BASE, CBASE and NFS. The throughput of BASE saturates at about 90 KB/sec since it cannot execute more than 1 request at a time. However, CBASE achieves its maximum throughput of about 2MB/sec when there is sufficient load on the system to run enough concurrent requests to achieve this throughput, which is almost 20 times more than that of the throughput of BASE. We did not modify the NFS implementation to sleep for 20 ms so its performance remains the same. This experiment shows that CBASE-FS does orders of magnitude better than BASE-FS when there is scope for concurrent execution of requests.

Benefits of Pipelining with multiple disks In this experiment we evaluate the performance benefits in the presence of real hardware concurrency. We run the same benchmark as above but with 3 server replicas running on machines with two disks (IBM-PSG and Quantum Viking II). The single disk experiment is run with single client which writes 4KB of data to a different file on the same disk (IBM-PSG) for 1000 times. Experiment with 2 disks is run with two clients each of which write 4KB of data to files on different disks. All the servers are configured to write data synchronously to disk. Figure 3.7 shows the throughput of BASE-FS, CBASE-FS and unreplicated NFS, when run with single disk and two disks. CBASE-FS and

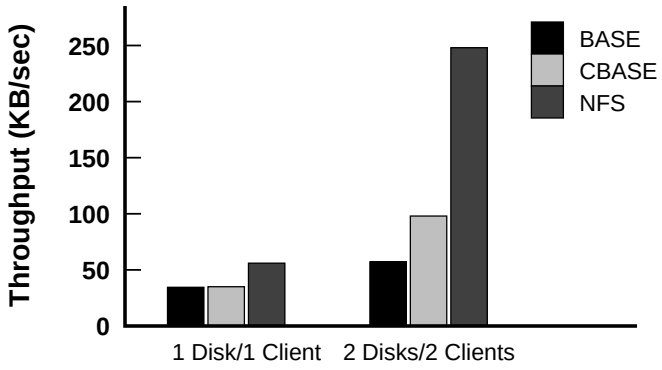
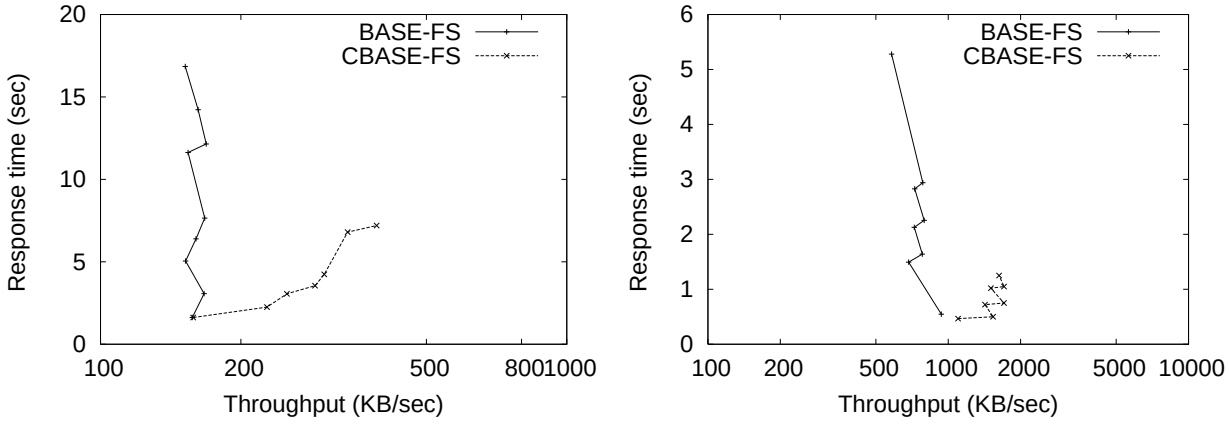


Figure 3.7: Throughput with multiple disks

BASE-FS have similar performance with a single disk but with two disks CBASE-FS outperforms BASE by 72% by concurrently writing to both disks. Unreplicated NFS outperforms both CBASE-FS by a factor of 1.5 with a single disk and 2.5 with two disks which is consistent with earlier results and for similar reasons.

Iozone micro-benchmark

Iozone [17] provides various microbenchmarks to test the performance of commercial file systems. We run the *write* and *random mix* micro-benchmarks to test CBASE-FS and compare its performance with BASE-FS. Rather than introduce artificial delays as above, we introduce the opportunity for hardware parallelism by configuring our system so that each file server accesses data on a *remote disk* that it mounts via NFS from a separate machine. Each IO request may thus access the local CPU, network, remote CPU, and remote disk, which affords the system an opportunity to benefit from pipelining. We use the remote disk setup to evaluate the performance in these experiments . We run the Iozone micro-benchmarks in cluster-mode,



(a) (b)

Figure 3.8: IOZONE: Throughput versus response time for (a) Write microbenchmark (b) Random microbenchmark

where clients are equally divided among 5 client machines and each client accesses a different file.

The write microbenchmark measures the performance of writing 256KB of data to a new file. We configure the test to have each client write to a different file to provide parallelism across the requests to the file systems. We vary the number of clients to vary the load on the system. As shown in the figure 3.8(a), BASE saturates at about 160 KB/sec where as CBASE saturates at about 320 KB/sec resulting in 100% improvement in performance as we vary load. CBASE-FS could not achieve more than a 2x improvement in performance despite having more available application-level parallelism because the system is limited by the remote disk bandwidth. Unreplicated NFS achieves a maximum throughput of 500KB/sec when the NFS server is running on the remote disk machine.

The random mix microbenchmark measures the performance of writing and reading files of size 256KB with accesses being made to random locations within each file. We configured the test to have clients write to different files to provide parallelism across requests, and we vary the number of clients to vary the load

on the system. As shown in the figure 3.8(b), BASE's throughput saturates at about 1MB/sec and CBASE's at about 2MB/sec. File caching at clients improves the throughput of both systems compared to the previous experiment. Overall, CBASE-FS's maximum throughput is 100% better than that of BASE-FS.

3.6.3 Macro-benchmarks

We evaluate the performance of CBASE-FS and BASE-FS with two file system macro-benchmarks: Andrew [55] and Postmark [29].

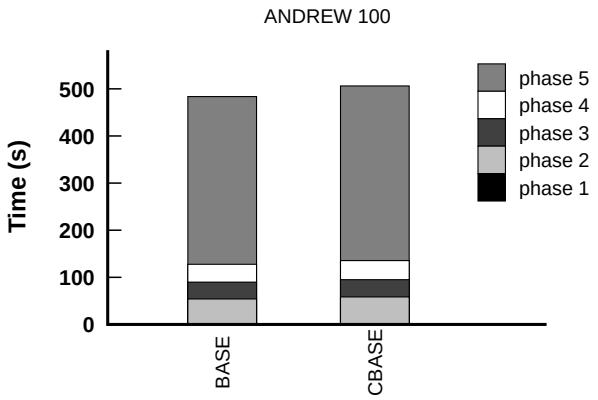


Figure 3.9: Andrew 100 benchmark

The Andrew-100 benchmark sequentially runs 100 copies of the Andrew benchmark which provides little concurrency. CBASE-FS and BASE-FS have essentially identical performance with BASE outperforming CBASE by 4% as there is no scope for concurrency.

PostMark [29] is a benchmark to measure performance of the Internet applications such as email, net news, e-commerce, etc. It initially creates a pool of files and then performs a specified number of transactions consisting of creating or deleting a file and reading or appending a file. We set file sizes to be between

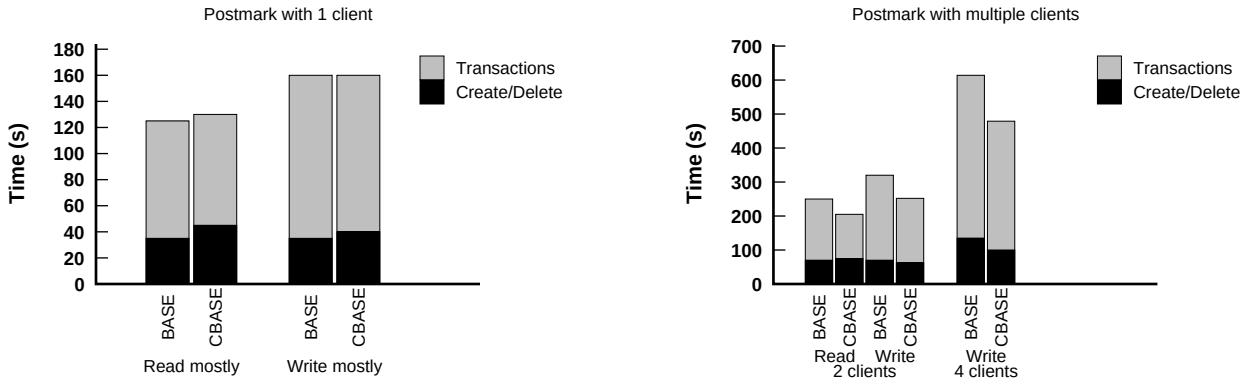


Figure 3.10: Postmark benchmark

1KB and 100KB. We run the benchmark with 100 files for 500 transactions. In our *read-mostly* experiment, we set the read bias at 9 so that transactions are dominated by reads over appends. In our *write-mostly* experiment, we set the read bias at 1 so that transactions are dominated by writes compared to reads. CBASE-FS and BASE-FS replicas write to the remote disk to evaluate the benefits of concurrent execution when run with multiple postmark clients.

Figure 3.10 shows the performance of BASE-FS and CBASE-FS when the experiment is run with varied number of Postmark clients. The performance of CBASE-FS and BASE-FS are nearly identical when run with 1 client. We also ran experiment with 2 and 4 postmark clients where each client operates on a different set of files. CBASE-FS is 20-25% faster than BASE-FS when run with multiple clients. CBASE-FS could not realise as much improvement in performance as in microbenchmarks because it is limited by the single available hardware disk.

3.7 Related Work

There is a large body of research on replication techniques to implement highly-available systems that tolerate failures. To the best of our knowledge, this was the first study that tries to improve application

throughput of a Byzantine fault tolerant system by providing a general way to use application semantics to execute requests concurrently.

Schneider [119] introduces the idea of using application semantics to reorder commutative requests in the state machine replication technique. Reordering requests can improve average response time of a system but will not improve throughput. We generalize this idea to use application semantics to identify independent requests and concurrently execute these requests to improve throughput of a system.

Byzantine fault tolerant state machine replication has been extensively studied [54, 70] and recent work has shown that BFT systems can be implemented in practical systems [55, 56, 111]. Although optimizations from these systems like request batching, reduced communication, and symmetric encryption improve throughput by reducing computation and network overhead, the throughput of these optimizations does not overcome the fundamental limits of sequential execution of requests. Some of these systems do support a tentative execution optimization to concurrently execute read requests, but such a solution cannot handle other type of requests. We provide a general strategy for exploiting application-level and hardware-level parallelism that can be applied to any of these systems.

Farsite [44] and Oceanstore [77] use PBFT [55] to provide Byzantine fault tolerant services. These systems provide scalability by partitioning application state where each partition can potentially be served by a different replica group (directory group /primary replica group). However, requests to a given group are sequentially executed which can limit the throughput of the system.

A recent work by Vandiver et al. [127] uses commit barrier scheduling to concurrently execute requests in BFT replicated transaction processing systems where it is hard to specify static application-specific rules to detect concurrent requests.

3.8 Conclusion

In this chapter, we proposed a high throughput BFT state machine replication architecture by making a simple change to existing BFT state machine replication architectures to introduce an application-specific parallelizer layer that allows concurrent execution of independent requests. We impelmented a system prototype called CBASE using this technique and demonstrated that it provides orders of magnitude improvement in performance over existing systems provided there is enough parallelism in the application and there are sufficient hardware resources. Although our work is motivated by and focussed on BFT state machine replication, the partial order property can be exploited in the context of traditional state machine replication based systems that tolerate fail-stop failures to improve throughput.

Chapter 4

Zyzzyva: Speculative Byzantine Fault Tolerance

In the previous chapter, we presented a BFT state machine replication architecture that provide high application throughput in the execution stage. In this chapter, we present the design and implementation of Zyzzyva[1], an efficient replication protocol used in the agreement stage. Zyzzyva uses speculation to reduce the performance overheads and replication cost of Byzantine fault tolerant state machine replication technique. The throughput, latency, and replication cost overheads of Zyzzyvamatch or approach the lower bounds.

4.1 Introduction

Three trends make Byzantine Fault Tolerant (BFT) replication increasingly attractive for practical deployment. First, the growing value of data and and falling costs of hardware make it advantageous for service providers to trade increasingly inexpensive hardware for the peace of mind potentially provided by BFT replication. Second, mounting evidence of non-fail-stop behavior in real systems [37, 50, 51, 98, 101, 106, 123, 134, 135] suggest that BFT may yield significant benefits even without resorting to *n*-version programming [48, 81, 111]. Third, improvements to the state of the art in BFT replication techniques [45, 58, 63, 86, 111, 136] make BFT replication increasingly practical by narrowing the gap between BFT replication costs and costs already being paid for non-BFT replication. For example, by default, the Google file system

[1] Zyzzyva (ZIZ-uh-vuh) is the last word in the dictionary.
According to dictionary.com, a zyzzyva is "any of various tropical American weevils of the genus Zyzzyva, often destructive to plants."

		PBFT	Q/U	HQ	Zyzzyva	State Machine Repl. Lower Bound
Cost	Total replicas	**3f+1**	5f+1	**3f+1**	**3f+1**	3f+1 [103]
	Replicas with application state	**2f+1** [136]	5f+1	3f+1	**2f+1**	2f+1
Throughput	MAC ops at bottleneck server	2+(8f+1)/b	2+8f	4+4f	**2+3f/b**	$2^{\dagger}$
Latency	Critical path NW 1-way latencies	4	**2**	4	**3**	$2/3^{\ddagger}$

Table 4.1: Properties of state-of-the-art and optimal Byzantine fault tolerant service replication systems tolerating f faults, using MACs for authentication [58], and using a batch size of b [58]. **Bold** entries denote protocols that match known lower bounds or those with the lowest known cost. †It is not clear that this trivial lower bound is achievable. ‡The distributed systems literature typically considers 3 one-way latencies to be the lower bound for agreement on client requests [66, 88, 96]; 2 one-way latencies is achievable if no concurrency is assumed. This table is explained in Appendix F.

uses 3-way replication of storage, which is roughly the cost of BFT replication for $f = 1$ failures with 4 agreement nodes and 3 execution nodes [136].

This chapter presents Zyzzyva, a new agreement protocol that uses *speculation* to reduce the cost and simplify the design of BFT state machine replication [89, 118]. Like traditional state machine replication protocols [58, 111, 136], a primary proposes an order on client requests to the other replicas. In Zyzzyva, unlike in traditional protocols, replicas speculatively execute requests without running an expensive agreement protocol to definitively establish the order. As a result, correct replicas' states may diverge, and replicas may send different responses to clients. Nonetheless, applications at clients observe the traditional and powerful abstraction of a replicated state machine that executes requests in a linearizable [76] order because replies carry with them sufficient history information for clients to determine if the replies and history are *stable* and guaranteed to be eventually committed. If a speculative reply and history are stable, the client uses the reply. Otherwise, the client waits until the system converges on a stable reply and history.

The challenge in Zyzzyva is ensuring that responses to correct clients become stable. Ultimately, replicas are responsible for ensuring that all requests from a correct client eventually complete, but a client waiting for a reply and history to become stable can speed the process by supplying information that will either cause

the request to become stable rapidly within the current view or trigger a view change. Note that because clients do not require requests to commit but only to become stable, clients act on requests in one or two phases rather than the customary three [58, 111, 136].

Essentially, Zyzzyva rethinks the sync [99] for BFT: instead of pessimistically ensuring that replicas establish a final order on requests before communicating with a client, we move the output commit to the client. Leveraging the client in this way offers significant practical advantages. Compared to state of the art protocols including PBFT [58, 111, 136], Q/U [45], and HQ [63], Zyzzyva reduces cryptographic overheads and increases peak throughput by a factor of two to an order of magnitude for demanding workloads. In fact, Zyzzyva's replication costs, processing overheads, and communication latencies approach their theoretical lower bounds.

4.1.1 Why another BFT protocol?

The state of the art for BFT state machine replication is distressingly complex. In a November 2006 paper describing Hybrid-Quorum replication (HQ replication) [63], Cowling et al. draw the following conclusions comparing three state-of-the-art protocols (Practical Byzantine Fault Tolerance (PBFT) [58, 86, 111, 136], Query/Update (Q/U) [45], and HQ replication [63]):

- "In the regions we studied (up to $f = 5$), if contention is low and low latency is the main issue, then if it is acceptable to use $5f + 1$ replicas, Q/U is the best choice, else HQ is the best since it outperforms [P]BFT with a batch size of 1." [63]
- "Otherwise, [P]BFT is the best choice in this region: it can handle high contention workloads, and it can beat the throughput of both HQ and Q/U through its use of batching." [63]

- "Outside of this region, we expect HQ will scale best: HQ's throughput decreases more slowly than Q/U's (because of the latter's larger message and processing costs) and [P]BFT's (where eventually batching cannot compensate for the quadratic number of messages)." [63]

Such complexity represents a barrier to adoption of BFT techniques because it requires a system designer to choose the right technique for a workload and then for the workload not to deviate from expectations.

As Table 4.1 indicates, Zyzzyva simplifies the design space of BFT replicated services by approaching the lower bounds in almost every key metric.

With respect to replication cost, Zyzzyva and PBFT match the lower bound both with respect to the total number of replicas that participate in the protocol and the number of replicas that must hold copies of application state and execute application requests. Both protocols hold cost advantages of 1.5–2.5 over Q/U and 1.0–1.5 over HQ depending on the number of faults to be tolerated and the relative cost of application vs. agreement node replication.

With respect to throughput, both Zyzzyva and PBFT use batching when load is high and thereby approach the lower bound on the number of authentication operations performed at the bottleneck node, and Zyzzyva approaches this bound more rapidly than PBFT. Q/U and HQ's inability to support batching increases the work done at the bottleneck node by factors approaching 5 and 4, respectively, when one fault is tolerated and by higher factors in systems that tolerate more faults.

With respect to latency, Zyzzyva executes requests in three one-way message delays, which matches the accepted lower bound in the distributed systems literature for agreeing on a client request [66, 88, 96] and improves upon both PBFT and HQ. Q/U sidesteps this lower bound by providing a service that is slightly weaker than state machine replication (i.e., it does not put a total order on all requests) and by optimizing for cases without concurrent access to any state. This difference presents a chink in Zyzzyva's armor, which

Zyzzyva minimizes by matching the lower bound on message delays for full consensus. We believe that Zyzzyva's other advantages over Q/U—fewer replicas, improved throughput via batching, simpler state machine replication semantics, ability to support high-contention workloads—justify this "extra" latency.

With respect to fault scalability [45], the metrics that depend on f grow as slowly or more slowly in Zyzzyva as in any other protocol.

Note that as is customary [45, 58, 63, 111, 136], Table 4.1 compares the protocols' performance during the expected common case of fault-free, timeout-free execution. All of the protocols are guaranteed to operate correctly in the presence of up to f faults and arbitrary delays, but all of these protocols can pay significantly higher overheads and latencies in such scenarios. In §4.5.5, we consider the susceptibility of these protocols to faults and argue that Zyzzyva remains the most attractive choice.

4.2 System Model

Zyzzyva is a BFT state machine replication protocol that cane be used to build replicated services like other BFT protocols [55, 63, 111, 136] as explained in chapter 2.

We assume the Byzantine failure model where faulty nodes (replicas or clients) may behave arbitrarily. We assume a strong adversary that can coordinate faulty nodes to compromise the replicated service. We do, however, assume the adversary cannot break cryptographic techniques like collision-resistant hashes, encryption, and signatures. In the public-key version of our protocol, we denote a message X signed by principal Y's public key as $\langle X \rangle_{\sigma_Y}$. Our system ensures its safety and liveness properties if at most f replicas are faulty. We assume a finite client population, any number of which may be faulty.

Our system's safety properties hold in any asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, corrupt them, delay them, or deliver them out of order. Liveness, however, is ensured only during intervals in which messages sent to correct nodes are processed within some fixed (but potentially unknown) worst case delay from when they are sent.

Our system implements a BFT service using state machine replication [58, 86, 118]. Traditional state machine replication techniques can be applied only to deterministic services. We cope with the non-determinism present in many real-word applications (such as file systems [25] and databases [131]) by abstracting the observable application state at the replicas and using the agreement stage to resolve non-deterministic choices [111, 127].

Services limit the damage done by Byzantine clients by authenticating clients, enforcing access control to deny clients access to objects they do not have a right to, and (optionally) by maintaining multiple versions of shared data (e.g., snapshots in a file system [82, 117]) so that data can be recovered from older versions if a faulty client destroys data [80].

4.3 Protocol

Zyzzyva is a BFT state machine replication protocol based on three sub-protocols: (1) agreement, (2) view change, and (3) checkpoint.

The *agreement* sub-protocol orders requests for execution by the replicas. The *view change* sub-protocol coordinates the election of a new primary when the current primary is faulty or the system is running slowly. The *checkpoint* sub-protocol limits the state that must be stored by replicas and reduces the cost of performing view changes.

Principles and Challenges Zyzzyva focuses on safety properties *as they are observed by the client*. In Zyzzyva, replicas can become temporarily inconsistent with one another, but clients detect inconsistencies, drive replicas to converge on a single total ordering of requests, and only rely on responses that are consistent with this total order.

Given the duties BFT replication protocols already place on clients [45, 58, 63, 91, 111, 136], it is not a large step to fully move the output commit to the client, but this small step pays big dividends. First,

Zyzzyva leverages speculative execution—replicas execute a request *before* its order is fully established. Second, Zyzzyva leverages fast agreement protocols [66, 88, 96] to establish a request ordering in as few as three message delays. Third, the agreement sub-protocol stops working on a request once a client knows the request's order, thereby avoiding work that would otherwise be needed to establish this knowledge at the replicas.

These choices lead to two key challenges in designing Zyzzyva. First, we must specify carefully the conditions under which a request *completes* at a client and define agreement, checkpoint, and view change sub-protocols to retain the abstraction that requests execute on a single, correct state machine. Intuitively, a request completes when a correct client may safely act on the reply to that request. To help a client determine when it is appropriate to act on a reply, Zyzzyva appends history information to the replies received by a client so that the client can judge whether the replies are based on the same ordering of requests. Zyzzyva ensures the following safety condition:

Safety: If a request with sequence number n and history h_n completes, then any request that completes with a higher sequence number $n' \geq n$ has a history $h_{n'}$ that includes h_n as a prefix.

Second, the view change sub-protocol must ensure liveness despite an agreement sub-protocol that never requires more than two phases to complete during a view. We shift work from the agreement sub-protocol to the view change sub-protocol by introducing a new "I hate the primary" phase that guarantees that a correct replica only abandons the current view if it can ensure that all other correct replicas will join the mutiny. Zyzzyva ensures the following liveness condition under eventual synchrony[2] [67]:

Liveness: Any request issued by a correct client eventually completes.

[2] In practice eventual synchrony can be achieved by using exponentially increasing timeouts [58].

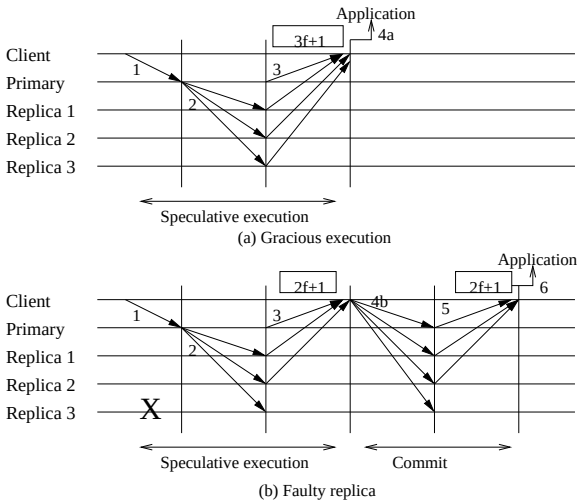


Figure 4.1: Protocol communication pattern within a view for (a) gracious execution and (b) faulty replica cases. The numbers refer to the main steps of the protocol numbered in the text.

Protocol Overview Zyzzyva is executed by $3f+1$ replicas, and execution is organized into a sequence of views. Within a view, a single replica is designated as the primary responsible for leading the agreement sub-protocol.

Figure 4.1 shows the communication pattern for a single instance of our client-centric fast agreement sub-protocol. A client sends a request to the primary, the primary forwards the request to the replicas, and the replicas execute the request and send their responses to the client. A request *completes* at a client in one of two ways. First, if the client receives $3f+1$ mutually-consistent *responses* (including an application-level *reply* and the *history* on which it depends), then the client considers the request complete and acts on it. Second, if the client receives between $2f+1$ and $3f$ mutually-consistent responses, then the client gathers $2f+1$ responses and distributes this *commit certificate* to the replicas. Once $2f+1$ replicas acknowledge receiving a commit certificate, the client considers the request *complete* and acts on the corresponding reply.

If a sufficient number of replicas suspect that the current primary is faulty, then a view change occurs and

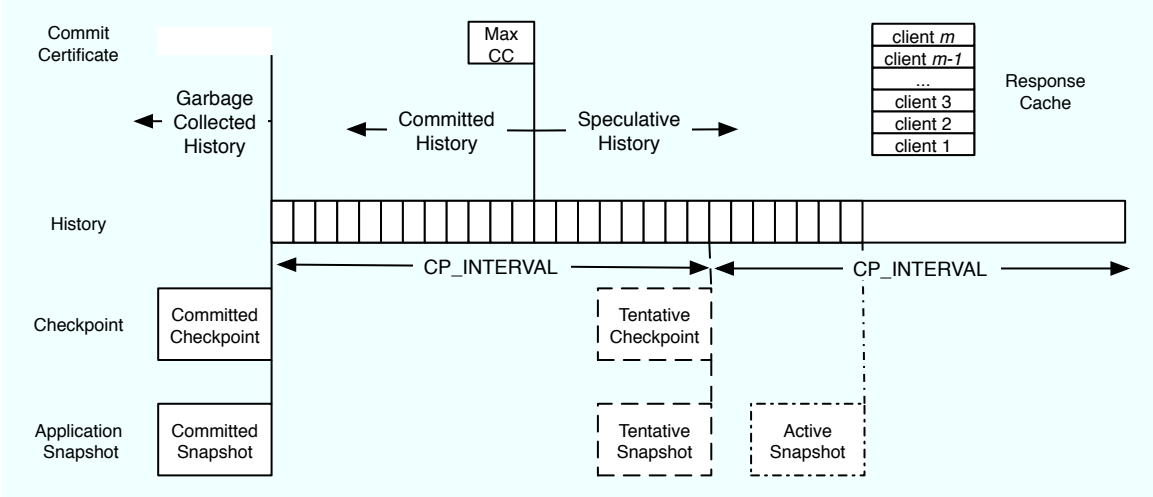


Figure 4.2: State maintained at each replica.

a new primary is elected.

In the rest of this section, we describe the basic protocol and outline the proof of its correctness and the details are in Appendix G. In §4.4 we describe a number of optimizations, all implemented in our prototype, that reduce encryption costs by replacing public key signatures with message authentication codes (MACs), improve throughput by batching requests, reduce the impact of lost messages by caching out-of-order messages, improve read performance by optimizing read-only requests, reduce bandwidth by having most replicas send hashes rather than full replies, reduce overheads by including MACs only for a preferred quorum, and improve performance in the presence of faulty nodes by including additional witness replicas.

In §4.4.1 we discuss Zyzzyva5, a variation of the protocol that requires $5f+1$ agreement replicas but that completes in three one-way message exchanges as in Figure 4.2(a) even when up to f non-primary replicas are faulty.

4.3.1 Node State and Checkpoint Protocol

To ground our discussion in definite terms, we begin by discussing the state maintained by each replica as summarized by Figure 4.2. Each replica *i* maintains an ordered *history* of the requests it has executed and a copy of the *max commit certificate*, the commit certificate (defined below) seen by *i* that covers the largest prefix of *i*'s stored history. The history up to and including the request with the highest sequence number covered by this commit certificate is the *committed history*, and the history that follows is the *speculative history*. We say that a commit certificate has sequence number *n* if *n* is the highest sequence number of any request in the committed history.

A replica constructs a checkpoint every *CP_INTERVAL* requests. A replica maintains one *stable checkpoint* and a corresponding *stable application state snapshot*, and it may store up to one *tentative checkpoint* and corresponding *tentative application state snapshot.* It commits the history before taking a tentative checkpoint. The process by which a tentative checkpoint and application state become committed is similar to the one used by earlier BFT protocols [58, 63, 86, 111, 136], so we defer a detailed discussion to Appendix G. However, to summarize briefly: when a correct replica generates a tentative checkpoint, it sends a signed CHECKPOINT message to all replicas. The message includes the highest sequence number of any request included in the checkpoint and a digest of the corresponding tentative checkpoint and application snapshot. A correct Zyzzyva replica considers the checkpoint and corresponding application snapshot stable when it collects $f + 1$ matching CHECKPOINT messages signed by different replicas.

To bound the size of the history, a replica (1) truncates the history before the committed checkpoint and (2) blocks processing of new requests after processing $2 \times$ *CP_INTERVAL* requests since the last committed checkpoint.

Finally, each replica maintains a *response cache* containing a copy of the latest ordered request from, and corresponding response to, each client.

4.3.2 Agreement Protocol

Figure 4.1 illustrates the basic flow of the agreement sub-protocol during a view. Because replicas execute requests speculatively in the order proposed by the primary without communicating with other replicas, the key challenge is ensuring that clients only act upon replies that correspond to stable requests executed in a total order that is guaranteed to eventually *commit* at all correct servers. The protocol is constructed so that a request *completes* at a client when the client receives $3f+1$ matching responses or acknowledgements from $2f+1$ replicas that they have received a *commit certificate* comprising a *local commit* from $2f+1$ replicas. Either of these conditions serves to prove that the request will eventually be *committed* at all correct replicas with the same sequence number and history of preceding requests observed by the client.

To describe how the system deals with this and other challenging, but standard, issues—lost messages, faulty primary, faulty clients, etc.—we follow a request through the system, defining the rules a server uses to process each message. The numbers in Figure 4.1 correspond to numbers in the text identifying major steps in the protocol and Table 4.2 summarizes the labels we give fields in messages. Most readers will be happier if on their first reading they skip the text marked Additional Pedantic Details.

1. Client sends request to the primary.

A client c requests an operation o be performed by the replicated service by sending a message $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ to the replica it believes to be the primary (i.e., the primary for the last response the client received).

Additional Pedantic Details: If the client guesses the wrong primary, the retransmission mechanisms discussed in step 4c below forwards the request to the current primary. The client's timestamp t is included to ensure exactly-once semantics of execution of requests.

2. Primary receives request, assigns sequence number, and forwards ordered request to replicas.

Label	Meaning
c	Client ID
CC	Commit certificate
d	Digest of client request message $d = H(m)$
i, j	Server IDs
h_n	History through sequence number n $h_n = H(h_{n-1}, d)$
m	Message containing client request
max_n	Max sequence number accepted by replica
n	Sequence number
o	Operation requested by client
OR	Order Request message
POM	Proof Of Misbehavior
r	Application reply to a client operation
t	Timestamp assigned to an operation by a client
v	View number

Table 4.2: Labels given to fields in messages.

When the primary p receives message $m = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ from client c, the primary assigns a sequence number n in view v to the request and relays a message $\langle \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_p}, m \rangle$ to the backup replicas where v indicates the view in which the message is being sent, n is the proposed sequence number for m, $d = H(m)$ is the digest of m, $h_n = H(h_{n-1}, d)$ is a digest summarizing the history, and ND is a set of values for non-deterministic application variables (time in file systems, locks in databases, etc.) required for execution.

Additional Pedantic Details: The primary only takes the above actions if $t > t_c$ where t_c is the highest timestamp previously received from c.

3. Replica receives ordered request, speculatively executes it, and responds to the client.

Upon receipt of a message $\langle \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_p}, m \rangle$ from the primary p, replica i accepts the ordered request if m is a well-formed REQUEST message, d is a correct digest of m, $n = max_n + 1$ where max_n is the largest sequence number in i's history, and $h_n = H(h_{n-1}, d)$. Upon accepting the message, i appends

the ordered request to its history, executes the request using the current application state to produce a reply r, and sends to c a message $\langle\langle$SPEC-RESPONSE, $v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$ where $OR = \langle$ORDER-REQ, $v, n, h_n, d, ND\rangle_{\sigma_p}$.

Additional Pedantic Details: A replica may only accept and speculatively execute requests in sequence-number order, but message loss or a faulty primary can introduce holes in the sequence number space. Replica i discards the ORDER-REQ message if $n \leq max_n$. If $n > max_n + 1$, then i discards the message, sends a message $\langle$FILL-HOLE, $v, max_n + 1, n, i\rangle_{\sigma_i}$ to the primary, and starts a timer. Upon receipt of a message $\langle$FILL-HOLE, $v, k, n, i\rangle_{\sigma_i}$ from replica i, the primary p sends a $\langle\langle$ORDER-REQ, $v, n', h_{n'}, d, ND\rangle_{\sigma_p}, m'\rangle$ to i for each request m' that p ordered in $k \leq n' \leq n$ during the current view; the primary ignores fill-hole requests from other views. If i receives the valid ORDER-REQ messages needed to fill the holes, it cancels the timer. Otherwise the replica broadcasts the FILL-HOLE message to all other replicas and initiates a view change when the timer fires. Any replica j that receives a FILL-HOLE message from i sends the corresponding ORDER-REQ message, if it has received one. If, in the process of filling in holes in the replica sequence, replica i receives conflicting ORDER-REQ messages then the conflicting messages form a proof of misbehavior as described in protocol step 4d.

4. Client gathers speculative responses.

The client receives messages $\langle\langle$SPEC-RESPONSE, $v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$, where i identifies the replica issuing the response, from the replicas. SPEC-RESPONSE messages from distinct replicas *match* if they have identical v, n, h_n, $H(r)$, c, t, and r fields. There are four cases to consider. The first three handle varying numbers of matching speculative replies without considering the *OR* field, while the last considers only the *OR* field.

4a. Client receives $3f + 1$ matching responses and completes the request.

In the absence of faults, the client receives matching SPEC-RESPONSE messages from all $3f+1$ replicas. The client then considers the request and its history to be *complete* and delivers the reply *r* to the application. Zyzzyva guarantees that even if there is a view change, all correct replicas will always execute this request at this point in their history to produce this response. Notice that although the client has a proof that the request's place in history is irrevocably set, no server has such a proof. Indeed, a server at this point cannot determine whether a request has completed in its final order or a roll-back of the server's state will be necessary because a faulty primary ordered the request inconsistently across replicas.

4b. Client receives between $2f+1$ and $3f$ matching responses, assembles a commit certificate, and transmits the commit certificate to the replicas.

If the network, primary, or some replicas are faulty, the client *c* may never receive responses from all $3f+1$ replicas. The client therefore sets a timer when it first issues a request: when this timer expires, if *c* has received matching speculative responses from between $2f+1$ and $3f$ replicas, then *c* sends a message $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$ where *CC* is a commit certificate consisting of a list of $2f+1$ replicas, the replica-signed portions of the $2f+1$ matching SPEC-RESPONSE messages from those replicas, and the corresponding $2f+1$ replica signatures.

Additional Pedantic Details: *CC* contains $2f+1$ signatures on the SPEC-RESPONSE message and a list of $2f+1$ nodes, but, since all the responses received by *c* from replicas are identical, *c* only needs to include *one* replica-signed portion of the SPEC-RESPONSE message. Also note that, for efficiency, *CC* does not include the body *r* of the reply but only the hash $H(r)$.

4b.1. Replica receives a COMMIT message from a client containing a commit certificate and acknowledges with a LOCAL-COMMIT message.

When a replica *i* receives a message $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$ containing a valid commit certificate *CC* proving that a request should be executed with a specified sequence number and history in the current view, the

replica first ensures that its local history is consistent with the one certified by *CC*. If so, replica *i* (1) updates its *max commit certificate* state if this certificate's sequence number is higher than the stored certificate's sequence number and (2) sends a message $\langle \text{LOCAL-COMMIT}, v, d, h, i, c \rangle_{\sigma_i}$ to *c*.

Additional Pedantic Details: If the local history simply has holes encompassed by *CC*'s history, then *i* fills them as described in 3. If, however, the two histories contain different requests for the same sequence number, then *i* initiates the view change protocol.

4b.2. Client receives a LOCAL-COMMIT messages from $2f+1$ replicas and completes the request.

The client resends the COMMIT message until it receives corresponding LOCAL-COMMIT messages from $2f+1$ distinct replicas. The client then considers the request and its history to be *complete* and delivers the reply *r* to the application. The system guarantees that even if there is a view change, all correct replicas will always execute this request at this point in their history to produce this response.

Additional Pedantic Details: When the client first sends the COMMIT message to the replicas it starts a timer. If this timer expires before the client receives $2f+1$ LOCAL-COMMIT messages then the client moves on to protocol step 4c described below.

4c. Client receives fewer than $2f+1$ matching SPEC-RESPONSE messages and resends its request to all replicas, which forward the request to the primary in order to ensure the request is assigned a sequence number and eventually executed.

Client. If the network or primary is faulty, the client *c* may never receive matching SPEC-RESPONSE messages from $2f+1$ replicas. The client therefore sets a second timer when it first issues a request and resends the REQUEST message to all replicas when the second timer expires. It then resets its timers and continues gathering speculative responses.

Replica. When non-primary replica *i* receives a message $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ from client *c* there are two possible actions for *i* to take. If the request matches or has a lower client-supplied timestamp than the

currently cached request for client c, then i resends the cached response to c. If instead the request has a higher timestamp than the currently cached response, then i sends a message $\langle$CONFIRM-REQ, v, m, $i\rangle_{\sigma_i}$ where $m = \langle \text{REQUEST}, o, t, c\rangle_{\sigma_c}$ to the primary p and starts a timer. If the replica accepts an ORDER-REQ message for this request before the timeout, it processes the ORDER-REQ message as described above. If the timer fires before the primary orders the request, the replica initiates a view change.

Primary. Upon receiving the confirm request message $\langle \text{CONFIRM-REQ}, v, m, i\rangle_{\sigma_i}$ from replica i, the primary p checks the client's timestamp for the request. If the request is new, p sends a new ORDER-REQ message using the next sequence number to order as described in step 2; otherwise, p sends to i the cached ORDER-REQ message for the most recent request from c.

Additional Pedantic Details: If replica i has received a commit certificate or stable checkpoint for a subsequent request, then the replica sends a LOCAL-COMMIT to the client even if the client has not received a commit certificate for the retransmitted request. Additionally, if replica i does not receive the ORDER-REQ message from the primary, the replica sends the CONFIRM-REQ message to all other replicas. Upon receipt of a CONFIRM-REQ message from another replica j, replica i sends the ORDER-REQ message it received from the primary to j; if i did not receive the request from the client, i acts as if the request came from the client itself.

4d. Client receives responses indicating inconsistent ordering by the primary and sends a proof of misbehavior to the replicas, which initiate a view change to oust the faulty primary.

If client c receives a pair of SPEC-RESPONSE messages containing valid messages $OR = \langle$ORDER-REQ, v, n, h_n, d, $ND\rangle_{\sigma_j}$ for the same request ($d = H(m)$) in the same view v with differing sequence number n or history h_n, then the pair of ORDER-REQ messages constitutes a proof of misbehavior (*POM*) against the primary. Upon receipt of a *POM*, c sends a message $\langle \text{POM}, v, POM\rangle_{\sigma_c}$ to all replicas. Upon receipt of a valid POM message, a replica initiates a view change and forwards the POM message to all other replicas.

Note that cases 4b and 4c are not exclusive of 4d; a client may receive messages sufficient to complete a request or form a commit certificate and also a proof of misbehavior against the primary.

4.3.3 View Changes

Fast agreement and speculative execution have profound effects on Zyzzyva's view change sub-protocol. In this section we highlight the differences between the Zyzzyva view change sub-protocol and that of previous systems. For completeness we include the full view change sub-protocol in Appendix G.1.1.

The view change sub-protocol must elect a new primary and guarantee that it will not introduce any changes in a history that has already completed at a correct client. To maintain this safety property, traditional view change sub-protocols [58, 63, 86, 111, 136] require a correct replica that commits to a view change to stop accepting messages other than CHECKPOINT, VIEW-CHANGE, and NEW-VIEW messages. Also, to prevent faulty replicas from disrupting the system, a view change sub-protocol should never remove a primary unless at least one correct replica commits to the view change. Hence, a correct replica traditionally commits to a view change if either (a) it observes the primary to be faulty or (b) it has a proof that $f+1$ replicas have committed to a view change. On committing to a view change a correct replica sends a signed VIEW-CHANGE message that includes the new view, the sequence number of the replica's latest stable checkpoint (together with a proof of its stability), and the set of prepare certificates—the equivalent of commit certificates in Zyzzyva—collected by the replica.

The traditional view change completes when the new primary, using $2f+1$ VIEW-CHANGE messages from distinct replicas, computes the history of requests that all correct replicas must adopt to enter the new view. The new primary includes this history, with a proof of validity, in a signed NEW-VIEW message that it broadcasts to all replicas.

Zyzzyva maintains the overall structure of the traditional protocol, but it departs in two significant ways

that together allow clients to accept a response before any replicas know that the request has been committed and allow the replicas to commit to a response after two phases instead of the traditional three.

1. First, to ensure liveness, Zyzzyva strengthens the condition under which a correct replica commits to a view change by adding a new "I hate the primary" phase to the view change sub-protocol. We explain the need for and details of this addition below by considering *The Case of the Missing Phase.*

2. Second, to guarantee safety, Zyzzyva weakens the condition under which a request appears in the history included in the NEW-VIEW message. We explain the need for and details of this change below by considering *The Case of the Uncommitted Request.*

The Case of the Missing Phase

As Figure 1 shows, Zyzzyva's agreement protocol guarantees that every request that completes within a view does so after at most two phases. This property may appear surprising to the reader familiar with PBFT. If we view a correct client that executes step **4b** of Zyzzyva as implementing a broadcast channel between replicas, then Zyzzyva's communication pattern maps to only two of PBFT's three phases, one where communication is primary-to-replicas *(pre-prepare)* and the second involving all-to-all exchanges (either *prepare* or *commit)*. Where did the third phase go? And why is it there in the first place?

The answer to the second question lies in the subtle dependencies between the agreement and view change sub-protocols. No replicated service that uses the traditional view change protocol can be live without an agreement protocol that includes both the *prepare* and *commit* full exchanges.[3] To see how this constraint applies to Zyzzyva, consider a scenario with f faulty replicas, one of them the primary, and suppose the faulty primary causes f correct replicas to commit to a view change and stop sending messages in the view.

[3]Unless a client can unilaterally initiate a view change. This option is unattractive when clients can be Byzantine.

In this situation, a client request may only receive $f+1$ responses from the remaining correct replicas, not enough for the request to complete in either the first or second phase—and, because fewer than $f+1$ replicas demand a view change, there is no opportunity to regain liveness by electing a new primary.

The third phase of traditional BFT agreement breaks this stalemate: by exchanging what they know, the remaining $f+1$ correct replicas can either gather the evidence necessary to complete the request after receiving only $f+1$ matching responses or determine that a view change is necessary.

Back to the first question: How does Zyzzyva avoid the third phase in the agreement sub-protocol? The insight is that what compromises liveness in the previous scenario is that the traditional view change protocol lets correct replicas commit to a view change and become silent in a view without any guarantee that their action will lead to the view change. Instead, in Zyzzyva, a correct replica does not abandon view *v* unless it is guaranteed that every other correct replica will do the same, forcing a new view and a new primary.

To ensure this property, the Zyzzyva view change sub-protocol adds an additional phase to strengthen the conditions under which a replica stops participating in the current view. In particular, a correct replica *i* that suspects the primary of view *v* continues to participate in the view, but expresses its vote of no-confidence in the primary by multicasting to all replicas a message $\langle \text{I-HATE-THE-PRIMARY}, v \rangle_{\sigma_i}$. If *i* receives $f+1$ votes of no confidence in *v*'s primary, then it commits to a view change: it becomes silent, and multicasts to all replicas a VIEW-CHANGE message that contains a proof that $f+1$ replicas have no confidence in the primary for view *v*. A correct replica that receives a valid VIEW-CHANGE message joins in the mutiny and commits to the view change. As a result, Zyzzyva's view change protocol ensures that if a correct replica commits to a view change in view *v*, eventually all correct replicas will. In effect, Zyzzyva shifts the costs needed to deal with a faulty primary from the critical path (the agreement protocol) to the view change sub-protocol, which is run only when the primary is faulty.

The Case of the Uncommitted Request

Zyzzyva replicas may never learn the outcome of the agreement protocol: only clients may know when a request has completed. How do Zyzzyva replicas identify a safe history prefix for a new view?

There are two ways in which a request r and its history may complete in Zyzzyva. Let us first consider the least problematic from the perspective of a view change: it occurs when r completes because a client receives $2f+1$ LOCAL-COMMIT messages, implying that at least $f+1$ correct replicas have stored a commit certificate for r. Traditional view change protocols already handle this case: the standard VIEW-CHANGE message sent by a correct replica includes all commit certificates known to the replica since the latest stable checkpoint. The new primary includes in the NEW-VIEW message all commit certificates that appear in any set of $2f+1$ VIEW-CHANGE messages it receives: at least one of those VIEW-CHANGE messages must contain a commit certificate for r.

The other case is more challenging: if r completes because the client receives $3f+1$ matching speculative responses, then no correct replica will have a commit certificate for r. We handle this case by modifying the view change sub-protocol in two ways. First, correct replicas add to the information included in their VIEW-CHANGE message all ORDER-REQ messages (without the corresponding client request) received since the latest stable checkpoint or commit certificate. Second, a correct new primary extends the history to be adopted in the new view to include all requests with an ORDER-REQ message containing a sequence number higher than the largest sequence number in any commit certificate that appears in at least $f+1$ of the $2f+1$ VIEW-CHANGE messages the new primary collects.

This change weakens the conditions under which a request ordered in one view can appear in a new view: we no longer require a commit certificate but also allow a sufficient number of ORDER-REQ messages to support a request's ordering. This change ensures that the protocol continues to honor ordering commitments

for any request that completes when a client gathers $3f+1$ matching speculative responses.

Notice that this change may have the side effect of assigning an order to a request that has not yet completed in the previous view. In particular, a curiosity of the protocol is that, depending on which set of $2f+1$ VIEW-CHANGE messages the primary uses, it may, for a given sequence number, find different requests with $f+1$ ORDER-REQ messages. This curiosity, however, is benign and cannot cause the system to violate safety. In particular, there can be two such candidate requests for the same sequence number only if at least one correct replica supports each of the candidates. In such a case, neither of the candidates could have completed by having a client receive $3f+1$ matching responses, and the system can safely assign either (or neither) request to that sequence number.

4.3.4 Correctness

This section sketches the proof that Zyzzyva maintains properties SAF and LIV defined above. We include complete proofs in Appendix G.

Safety

We first show that our agreement sub-protocol is safe within a single view and then show that the agreement and view change protocols together ensure safety across views.

Within a View The proof proceeds in two parts. First we show that no two requests complete with the same sequence number n. Second we show that h_n is a prefix of $h_{n'}$ for $n < n'$ and completed requests r and r'.

Part 1: A request completes when the client receives $3f+1$ matching SPEC-RESPONSE messages in phase 1 or $2f+1$ matching LOCAL-COMMIT messages in phase 2. If a request completes in phase 1 with sequence number n, then no other request can complete with sequence number n because correct replicas (a) send only

one speculative response for a given sequence number and (b) send a LOCAL-COMMIT message only after seeing $2f+1$ matching SPEC-RESPONSE messages. Similarly, if a request completes with sequence number n in phase 2, no other request can complete since correct replicas only send one LOCAL-COMMIT message for sequence number n.

Part 2: For any two requests r and r' that complete with sequence numbers n and n' and histories h_n and $h_{n'}$ respectively, there are at least $2f+1$ replicas that ordered each request. Because there are only $3f+1$ replicas in total, at least one correct replica ordered both r and r'. If $n < n'$, it follows that h_n is a prefix of $h_{n'}$.

Across Views We show that any request that completes based on responses sent in view $v < v'$ is contained in the history specified by the NEW-VIEW message for view v'. Recall that requests complete either when a correct client receives $3f+1$ matching speculative responses or $2f+1$ matching local-commits.

If a request r completes with $2f+1$ matching local-commits, then at least $f+1$ correct replicas have received a commit certificate for r (or for a subsequent request) and will send that commit certificate to the new primary in their VIEW-CHANGE message. Because there are $3f+1$ replicas in the system and $2f+1$ VIEW-CHANGE messages in a NEW-VIEW message, that commit certificate will necessarily be included in the NEW-VIEW message and r will be included in the history. Consider instead a request r that completes with $3f+1$ matching SPEC-RESPONSE messages and does not complete with $2f+1$ matching LOCAL-COMMIT messages. Every correct replica will include the ORDER-REQ for r in its VIEW-CHANGE message, ensuring that the request will be supported by at least $f+1$ replicas in the set of $2f+1$ VIEW-CHANGE messages collected by the primary of view v' and therefore be part of the NEW-VIEW message.

Liveness

Zyzzyva guarantees liveness only during periods of synchrony. To show that a request issued by a correct client eventually completes, we first show that if the primary is correct when a correct client issues the request, then the request completes. We then show that if a request from a correct client does not complete during the current view, then a view change occurs.

Part 1: If the client and primary are correct, then protocol steps 1 through 3 ensure that the client receives SPEC-RESPONSE messages from all correct replicas. If the client receives $3f+1$ matching SPEC-RESPONSE messages, the request completes—and so does our proof. A client that instead receives fewer than $3f+1$ such messages will receive at least $2f+1$ of them, since there are $3f+1$ replicas and at most f of which are faulty. This client then sends a COMMIT message to all replicas (protocol step 4b). All correct replicas send a LOCAL-COMMIT message to the client (protocol step 4b.1), and, because there are at least $2f+1$ correct replicas, the client's request completes in protocol step 4b.2.

Part 2: Assume the request from correct client c does not complete. By protocol step 4c, c resends the REQUEST message to all replicas when the request has not completed for a sufficiently long time. A correct replica, upon receiving the retransmitted request from c, contacts the primary for the corresponding ORDER-REQ message. Any correct replica that does not receive the ORDER-REQ message from the primary initiates the view change by sending an I-HATE-THE-PRIMARY message to all other replicas. Either at least one correct replica receives at least $f+1$ I-HATE-THE-PRIMARY messages, or no correct replica receives at least $f+1$ I-HATE-THE-PRIMARY messages. In the first case, the replicas commit to a view change—QED. In the second case, all correct replicas that did not receive the ORDER-REQ message from the primary receive it from another replica. After receiving an ORDER-REQ message, a correct replica sends a SPEC-RESPONSE to c. Because all correct replicas send a SPEC-RESPONSE message to c, c is guaranteed to receive at least $2f+1$ such messages. Note that c must receive fewer than $2f+1$ matching SPEC-RESPONSE messages:

otherwise, *c* would be able to form a COMMIT and complete the request, contradicting our initial assumption. If however, *c* does not receive $2f+1$ matching SPEC-RESPONSE messages, then *c* is able to form a POM message: *c* relays this message to the replicas which in turn initiate and commit to a view change, completing the proof.

4.4 Implementation Optimizations

Our implementation includes several optimizations to improve performance and reduce system cost.

Replacing Signatures with MACs Like previous work [45, 58, 63, 86, 111, 136], we replace most signatures in Zyzzyva with MACs and authenticators in order to reduce the computational overhead of cryptographic operations. The only signatures that are not replaced with MACs are client request retransmissions and the messages of the view change protocol. The technical changes to each sub-protocol required by replacing signatures with authenticators are described in Appendix [**?**]. The most noticeable difference in the agreement sub-protocol is the way Zyzzyva addresses the scenario in which replica *i* is unable to authenticate a client request; *i* cannot distinguish whether the fault lies with the primary or the client. Our procedure in this case is similar to a view change and results in correct replicas agreeing to accept the request or replace it with a *no-op* in the sequence. The checkpoint sub- protocol adds a third phase to ensure that stable checkpoints are consistent with requests that complete through speculative execution. Finally, the view change sub-protocol includes an additional phase for gathering checkpoint and commit certificate proofs as is done in PBFT [58].

Separating Agreement from Execution We separate agreement from execution [136] by requiring only $2f+1$ replicas to be execution replicas. The remaining replicas serve as witness replicas [92], aiding in the process of ordering requests but not replicating the application. Clients accept a history based on the agreement protocol described in the previous section with a slight modification: a pair of responses are

considered to match even if the response r and response hash $H(r)$ fields are not identical. A client acts on a reply only after receiving the appropriate number of matching responses and $f+1$ matching application replies from execution replicas. One consequence of this optimization is that a client may have to wait until it has received more than $2f+1$ responses before it can act in the second phase. We gain further benefit by biasing the primary selection criteria so that witness replicas are chosen as the primary more frequently than execution replicas. This favoritism reduces processor contention at the primary and allows requests to be ordered and processed faster.

Request Batching We batch concurrent requests to reduce cryptographic and communication overheads like other agreement-based replicated services [58, 86, 111, 124, 136]. Batching requests amortizes the cost of replica operations across multiple requests and reduces the total number of operations per request. One key step in batching requests is having replicas compute a single history digest corresponding to the entries in the batch. This batch history is used in responses to all requests included in the batch. If the second phase completes for any request in the batch, the second phase is considered complete for all requests in the batch and replicas respond to the retransmission of any requests in the batch with local-commit messages.

Caching Out of Order Requests The protocol described in section 4.3.2 dictates that replicas discard order request messages that are received out of order. We improve performance when the network delivers messages out of order by caching these requests until the appropriate sequence number is reached. Similarly, the view change sub-protocol can order additional requests that are not supported by $f+1$ speculative responses.

Read-Only Optimization Like PBFT [58], we improve the performance of read- only requests that do not modify the system state. A client sends read-only requests directly to the replicas which execute the

requests immediately, without recording the request in the request history. As in PBFT, clients wait for $2f+1$ matching replies in order to complete read-only operations. In order for this optimization to function, we augment replies to read requests with a replica's $\max_n$ and $\max_{CC}$. A client that receives $2f+1$ matching responses, including the $\max_n$ and $\max_{CC}$ fields, such that $\max_n = \max_{CC}$ can accept the reply to the read. Furthermore, a client that receives $3f+1$ matching replies, even if the $\max_{CC}$ and $\max_n$ values are not consistent, can accept the reply to the read.

Single Execution Response The client specifies a single execution replica to respond with a full response while the other execution replicas send only a digest of the response. This optimization is introduced in PBFT [58] and saves network bandwidth proportional to the size of responses.

Preferred Quorums Q/U [45] and HQ [63] leverage preferred quorums to reduce the size of authenticators by optimistically including MACs for a subset of replicas rather than all replicas. We implement preferred quorums for the second phase; replicas authenticate speculative response messages for the client and a subset of $2f$ other replicas. Additionally, on the initial transmission, we allow the client to specify that replicas should authenticate speculative response messages to the client only. This optimization reduces the number of cryptographic operations performed by backup replicas to three while existing BFT systems [45, 58, 63, 86, 111, 136] require a linear number of cryptographic operations at each replica.

Other optimizations First, we use an adaptive commit timer at the client to initiate the commit phase which adapts to the slowest replica in the system. Second, like PBFT, clients broadcast requests directly to all the replicas where as the primary uses just the request digest in the order request message.

4.4.1 Making the Faulty Case Fast

Commit Optimization In the presence of faults, the protocol described in section 4.3.2 requires that clients start the second phase (commit phase) if they receive fewer than $3f+1$ responses. Replicas then verify the commit certifcate and send the local-commit response. The problem with this approach is that the replicas end up splitting the batch of requests in the first phase when replies are sent back to the clients and then verify commit messages from each client separately in the second phase. Thus, replicas fail to amortize the verification cost in the second phase.

Zyzzyva addresses this problem using commit optimization where clients assign a bit to hint replicas that they send speculative replies after committing the request locally. When this bit is set, replicas broadcast order request messages (similar to prepare message in PBFT) after they receive a valid order request message from the primary and do not send speculative response immediately. If a replica receives $2f+1$ matching order request messages from other replicas it then commits the request locally (as if it recieved a valid commit certificate in the *COMMIT* message from the client), executes the request, and sends the speculative response to the client with both max_n and $max_C C$ set to the request order. Like read-only optimization, clients consider a request to be complete if they receive 2f+1 matching speculative responses with $\max_n = max_{CC}$ and deliver response to the application. Clients set the commit optimization bit whenever they complete a request using two phases. They reset the bit to zero if they receive speculative responses from all 3f+1 replicas. Clients start with this bit set to zero assuming that they are no faults in the system.

Unlike the original protocol, this optimization allows replicas to verify order request messages once for the entire batch before committing the request locally. This optimization reduces the cryptographic overhead at a replica from $3+\frac{5f+1}{b}$ crypto ops per request to $2+\frac{5f+1}{b}$ crypto ops per request. We evaluate the performance impact of this optimization in section 4.5.5.

Zyzzyva5 We introduce a second protocol, Zyzzyva5, that uses $2f$ additional *witness replicas* (the number of execution replicas is unchanged at $2f+1$) for a total of $5f+1$ replicas. Increasing the number of replicas lets clients receive responses in three message delays even when f replicas are faulty [66, 88, 96]. Zyzzyva5 trades the number of replicas in the deployed system against performance in the presence of faults. Zyzzyva5 is identical to Zyzzyva with a simple modification—nodes wait for an additional f messages, i.e. if a node bases a decision on a set of $2f+1$ messages in Zyzzyva, the corresponding decision in Zyzzyva5 is based on a set of $3f+1$ messages. The exceptions to this rule are the "I hate the primary" phase of the view change protocol and the fill-hole and confirm-request sub-protocols that serve to prove that another correct replica has taken an action—these phases still require only $f+1$ responses.

4.5 Evaluation

This section examines the performance characteristics of Zyzzyva and compares it with existing approaches. We run our experiments on 3.0 GHz Pentium-4 machines with the Linux 2.6 kernel. We use MD5 for MACs and AdHash [52] for incremental hashing. MD5 is known to be vulnerable, but we use it to make our results comparable with those in the literature. Since Zyzzyva uses fewer MACs per request than any of the competing algorithms, our advantages over other algorithms would be increased if we were to use the more secure, but more expensive, SHA-256.

For comparison, we run Castro et al.'s implementation of PBFT [58] and Cowling et al.'s implementation of HQ [63]; we scale up measured throughput for the small request/response benchmark by 9% [26] to account for their use of SHA-1 rather than MD5. We include published throughput measurements for Q/U [45]; we scale reported performance up by 7.5% to account for our use of 3.0 GHz rather than 2.8GHz machines. We also compare against measurements of an unreplicated server.

Unless noted otherwise, in our experiments we use all of the optimizations other than preferred quorums for Zyzzyva as described in §4.4. PBFT [58] does not implement preferred quorum optimization. We run

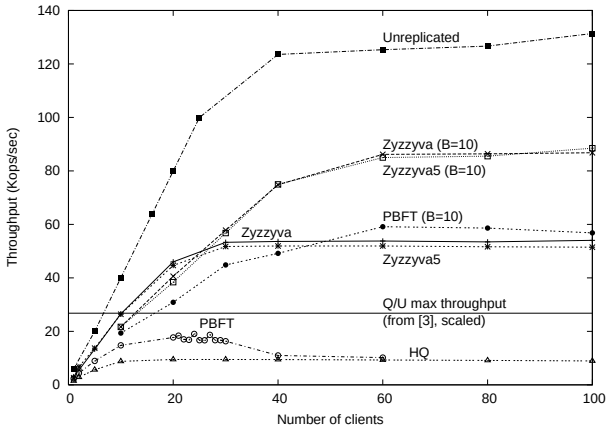


Figure 4.3: Realized throughput for the 0/0 benchmark as the number of client varies for systems configured to tolerate $f = 1$ faults.

with preferred quorum optimization for HQ [63]. We do not use the read-only optimization for Zyzzyva and PBFT unless we state so explicitly.

4.5.1 Throughput

To stress-test Zyzzyva we use the micro-benchmarks devised by Castro et al. [58]. In the 0/0 benchmark, a client sends a null request and receives a null reply. In the 4/0 benchmark, a client sends a 4KB request and receives a null reply. In the 0/4 benchmark, a client sends a null request and receives a 4KB reply.

Figure 4.3 shows the throughput achieved for the 0/0 benchmark by Zyzzyva, Zyzzyva5, PBFT, and HQ (scaled as noted above). For reference, we also show the peak throughput reported for Q/U [45] in the $f = 1$ configuration, scaled to our environment as described above. As the number of clients increases, Zyzzyva and Zyzzyva5 scale better than PBFT with and without batching. Without batching, Zyzzyva achieves a peak throughput that is 2.7 times higher than PBFT due to PBFT's higher cryptographic overhead (PBFT performs about 2.2 times more crypto operations than Zyzzyva) and message overhead (PBFT sends and receives about 3.7 times more messages than Zyzzyva). When the batch size is increased to 10, Zyzzyva's

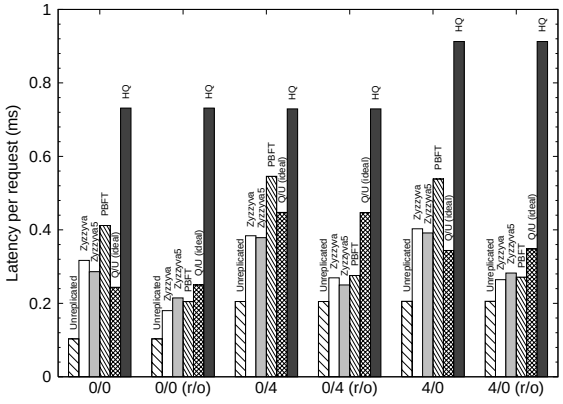


Figure 4.4: Latency for 0/0, 0/4, and 4/0 benchmarks for systems configured to tolerate $f = 1$ faults.

and Zyzzyva5's peak throughputs increase to 86K ops/sec suggesting that the protocol overhead at the primary is 12μs per batched request. With batching, PBFT's throughput increases to 59K ops/sec. The 45% difference between Zyzzyva and PBFT's peak throughput are largely accounted for PBFT's higher cryptographic overhead (about 30%) and message overhead (about 30%) compared to Zyzzyva. Zyzzyva provides over 3 times the reported peak throughput of Q/U and over 9 times the measured throughput of HQ. This difference stems from three sources. First, Zyzzyva requires fewer cryptographic operations per request compared to HQ and Q/U. Second, neither Q/U nor HQ is able to use batching to reduce cryptographic and message overheads. Third, Q/U and HQ do not take advantage of the Ethernet broadcast channel to speed up the one-to-all communication steps.

Overall, the peak throughput achieved by Zyzzyva is within 35% of that of an unreplicated server that simply replies to client request over an authenticated channel. Note that as application-level request processing increases, the protocol overhead will fall.

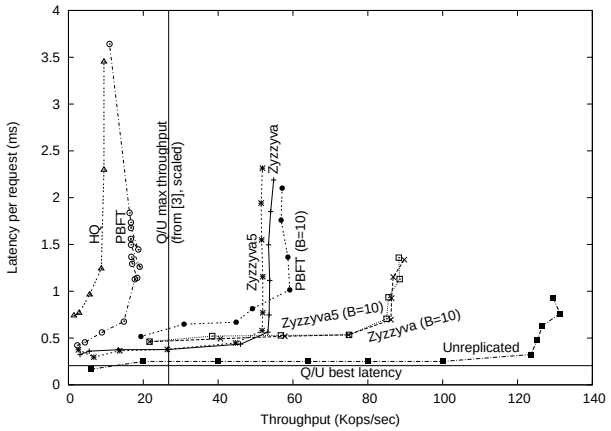


Figure 4.5: Latency vs. throughput for systems configured to tolerate $f = 1$ faults.

4.5.2 Latency

Figure 4.4 shows the latencies of Zyzzyva, Zyzzyva5, Q/U, and PBFT for the 0/0, 0/4, and 4/0 microbenchmarks. For Q/U, which can complete in fewer message delays than Zyzzyva during contention-free periods, we use a simple best-case implementation of Q/U with preferred quorums in which a client simply generates and sends $4f+1$ MACs with a request, each replica verifies $4f+1$ MACs (1 to authenticate the client and $4f+1$ to validate the OHS state), each replica generates and sends $4f+1$ MACs (1 to authenticate the reply to the client and $4f$ to authenticate OHS state) with a reply to the client, and the client verifies $4f+1$ MACs. We examine both the default read/write requests that use the full protocol and read-only requests that exploit the read-only optimization.

Zyzzyva uses fast agreement to drive its latency near the optimal for an agreement protocol—3 one-way message delays [66, 88, 96]. The experimental results in Figure 4.4 show that Zyzzyva and Zyzzyva5 achieve significantly lower latency than the other agreement-based protocols, PBFT and HQ. As expected, Q/U's avoidance of serialization gives it even better latency in low-contention workloads such as the one examined here, though Zyzzyva and PBFT can match Q/U for read-only requests where all of these protocols

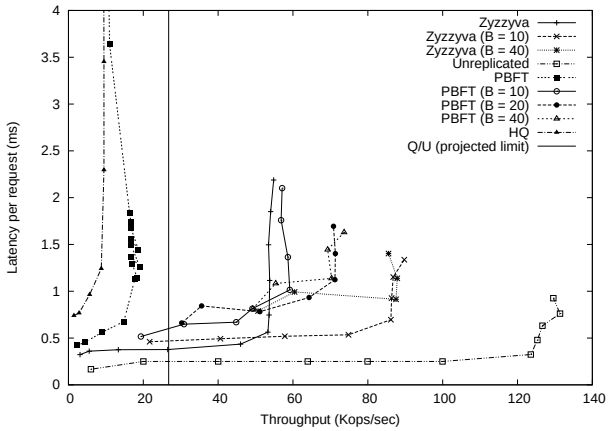


Figure 4.6: Latency vs. throughput for systems configured to tolerate $f = 1$ faults.

can complete in two message delays.

Figure 4.5 shows latency and throughput as we vary offered load. As the figure illustrates, batching in Zyzzyva, Zyzzyva5, and PBFT increases latency but also increases peak throughput. Adaptively setting the batch size in response to workload characteristics is an avenue for future work.

4.5.3 Batching

In this section we examine the effect of varying batch sizes on the peak throughputs of Zyzzyva and PBFT. Figure 4.6 showa that the peak throughput of Zyzzyva saturates at a batch size of 10 whereas the peak throughput of PBFT saturates at 20. Zyzzyva continues to outperform PBFT even with increasing batch sizes although with a reduced margin.

4.5.4 Fault Scalability

In this section we examine performance of these protocols as f, the number of tolerated faults, increases.

Figure 4.7 shows the peak throughputs of Zyzzyva, PBFT, HQ, and Q/U (reported throughput) with increasing number of tolerated faults for batch sizes of 1 and 10. Zyzzyva is robust to increasing f and

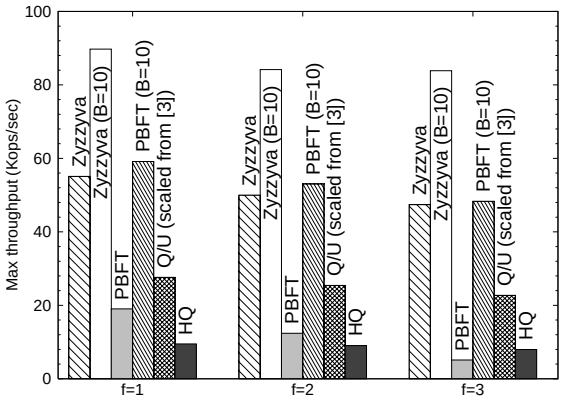
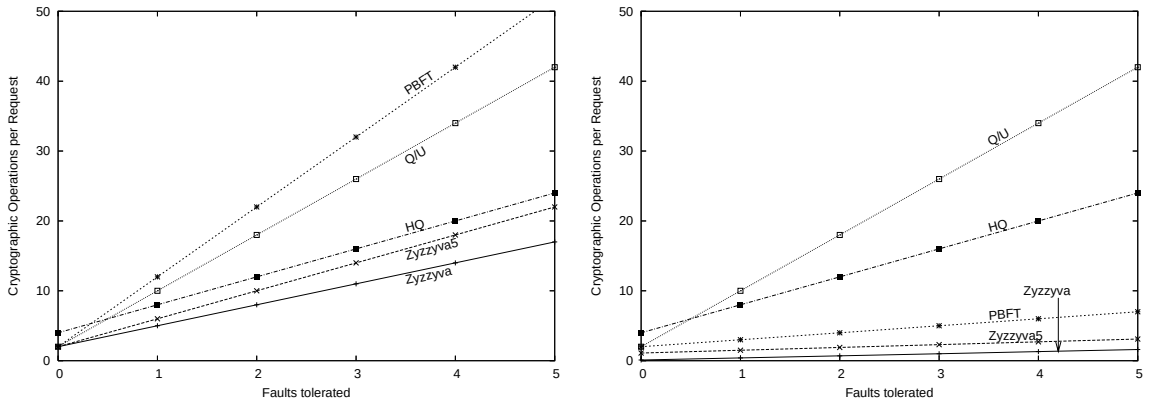


Figure 4.7: Fault scalability: Peak throughputs

continues to provide significantly higher throughput than other systems for the same reasons as explained in the throughput section. Additionally, as expected for the case with no batching, the overhead of Zyzzyva increases more slowly than PBFT with increasing f because Zyzzyva requires $2+(3f+1)$ cryptographic operations compared to $2+(10f+1)$ cryptographic operations for PBFT.

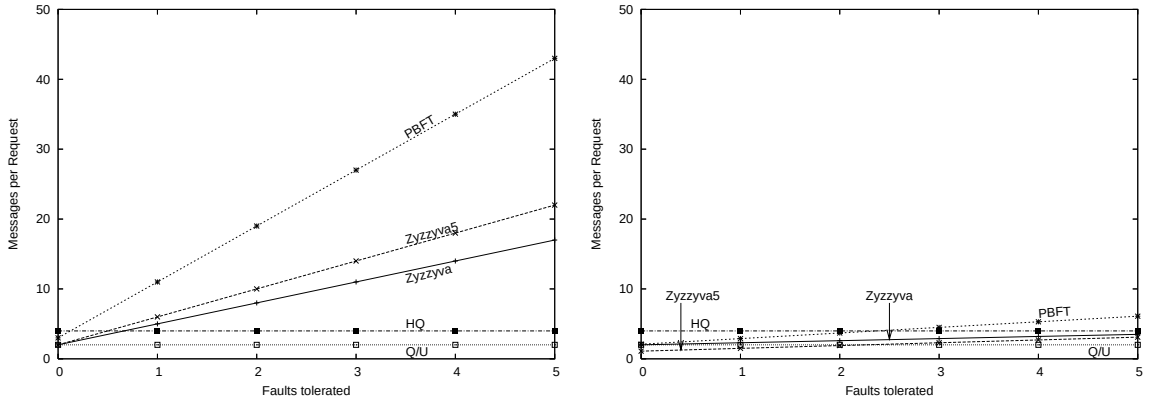
Figures 4.8 shows the number of cryptographic operations per request and the number of messages sent and received per request at the bottleneck server (the primary in Zyzzyva, Zyzzyva5, PBFT, and any server in Q/U and HQ). We believe that for these metrics, the most interesting regions are when f is small and when batching is enabled. Not coincidentally, Zyzzyva performs well in these situations, dominating all of the approaches with respect to load at the bottleneck server. Also, when f is small, Zyzzyva and Zyzzyva5 also have low message counts at the primary.

As f increases, when batching is used, Zyzzyva and Zyzzyva5 are likely to remain attractive. One point worth noting is that message counts at the primary for Zyzzyva, Zyzzyva5, and PBFT increase as f increases, while server message counts are constant with f for Q/U and HQ. In this figure, message counts do not include the multicast optimization we exploited in our experiments. Multicast reduces the number of client messages for all protocols by allowing clients to transmit their requests to all servers in one send.



Batch size = 1 Batch size = 10

Bottleneck server cryptographic operations



Batch size = 1 Batch size = 10

Bottleneck server messages

Figure 4.8: Fault scalability using analytical model

Multicast also reduces the number of server messages for Zyzzyva, Zyzzyva5, PBFT, and HQ (but not Q/U) when the primary or other servers communicate with their peers. In particular, with multicast the primary sends or receives one message per batch of operations plus an additional two messages per request regardless of f.

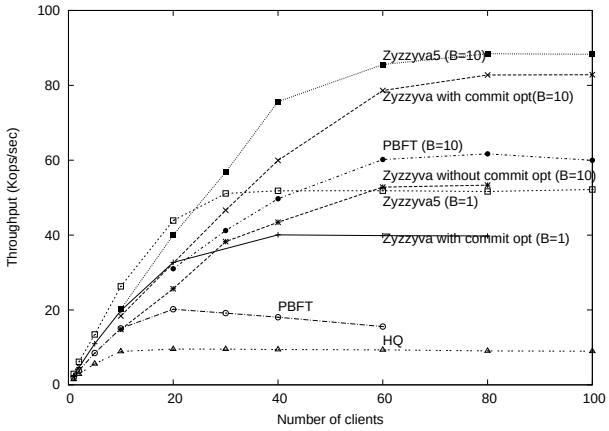


Figure 4.9: Realized throughput for the 0/0 benchmark as the number of client varies when f non-primary replicas fail to respond to requests.

We examine other metrics in Appendix F.1 such as message and cryptographic overheads at the client and find, for example, that Zyzzyva improves upon all of the protocols except PBFT by this metric. These graphs are omitted due to space constraints.

4.5.5 Performance During Failures

Zyzzyva guarantees correct execution with any number of faulty clients and up to f faulty replicas. However, its performance is optimized for the expected case of failure-free operation. In particular a single faulty replica can force Zyzzyva to execute the slower 2 phase protocol. Zyzzyva's protocol, however, remains relatively efficient in such scenarios. In particular, Zyzzyva's cryptographic overhead increases from $2+\frac{3f+1}{b}$ to $3+\frac{5f+1}{b}$ operations per batch without the commit optimization. However, commit optimization reduces the the cryptographic overhead of Zyzzyva from $3+\frac{5f+1}{b}$ to $2+\frac{5f+1}{b}$ crypt ops/request. Zyzzyva5's increased fault tolerance ensures that its overheads do not increase in such scenarios, remaining at $2+\frac{5f+1}{b}$ per batch. For comparison, PBFT uses $2+\frac{10f+1}{b}$ operations in both this scenario and fault-free.

Figure 4.9 compares throughput with increasing numbers of clients for Zyzzyva, Zyzzyva5, PBFT, and

HQ in the presence of f backup server failures. For the case when $f = 1$, with one failure and no batching ($b = 1$), Zyzzyva and Zyzzyva5 provide 1.8 and 2.6 times higher throughput than PBFT, respectively, because of additional cryptographic and message overheads as described above. Zyzzyva (with commit optimization) and Zyzzyva5 continue to outperform PBFT even with increased batch size of 10 although with a reduced margin. However, PBFT performs 15% better than Zyzzyva wthout the commit optimization because Zyzzyva ($3 + \frac{5f+1}{b}$) incurs higher overhead than PBFT ($2 + \frac{10f+1}{b}$). Also, note that Zyzzyva5 performs slightly better than Zyzzyva because the latter incurs additional processing overhead in the commit phase whereas the former does not need an additional commit phase. For the same reasons as described in the throughput section, Zyzzyva, Zyzzyva5, and PBFT outperform HQ. We do not include a discussion of Q/U in this section as the throughput numbers of Q/U with failures are not reported [45].

A limitation Zyzzyva and Zyzzyva5 share with PBFT (and HQ during periods of contention) is that a faulty primary can significantly prevent progress. These protocols replace the primary to ensure progress. Although Q/U avoids having a primary, it shares a corresponding vulnerability: a faulty client that fails to adhere to the back-off protocol can impede progress indefinitely.

4.6 Related Work

Starting with PBFT [58, 111] several systems [45, 63, 86, 136] have explored how to make Byzantine services practical. We have discussed throughout the paper how Zyzzyva builds upon these systems and how it departs from them. As its predecessors, Zyzzyva leverages ideas inspired by Paxos [90] and by work on Byzantine quorum systems [94]. In particular, Zyzzyva fast agreement protocol is based on recent work on fast Paxos [66, 88, 96].

Numerous BFT agreement protocols [58, 63, 86, 96, 111, 136] have used *tentative execution* to reduce the latency experienced by clients. This optimization allows replicas to execute a request tentatively as soon as

they have collected the Zyzzyva equivalent of a commit certificate for that request. This optimization may superficially appear similar to Zyzzyva's support for *speculative executions*—but there are two fundamental differences. First, Zyzzyva's speculative execution allows requests to complete at a client after a single phase, without the need to compute a commit certificate: this reduction in latency is not possible with traditional tentative executions. Second, and more importantly, in traditional BFT systems a replica can execute a request tentatively only after the replica's "state reflects the execution of all requests with lower sequence number, and these requests are all known to be committed" [**?**]. In Zyzzyva, replicas continue to execute request speculatively, without waiting to know that requests with lower sequence numbers have completed; this difference is what lets Zyzzyva leverage speculation to achieve not just lower latency but also higher throughput.

Q/U [45] provides high throughput assuming low concurrency in the system but requires higher number of replicas than Zyzzyva. HQ [63] uses fewer replicas than Q/U but uses multiple rounds to complete an operation. Both HQ and Q/U fail to batch concurrent requests and incur higher overhead in the presence of request contention; Singh et al. [124] add a preserializer to HQ and Q/U to address these issues.

BFT2F [91] explores how to gracefully weaken the consistency guarantees provided by BFT state machine replication when the number of faulty replicas exceeds one third (but is no more than two thirds) of the total replicas.

Speculator [100] allows clients to speculatively complete operations at the application level and perform client level rollback. A similar approach could be used in conjunction with Zyzzyva to support clients that want to act on a reply optimistically, rather than waiting on the specified set of responses.

4.7 Conclusion

By systematically exploiting speculation, Zyzzyva exhibits significant performance improvements over existing BFT agreement protocols. The throughput and latency of Zyzzyva approach the theoretical lower

bounds for any BFT protocol.

Chapter 5

SafeStore: A Durable and Practical Storage System

BFT state machine replication techniques provide better short-term availability (ability to access data when desired) but may fail to provide long-term data durability (ability to store data correctly for long durations) spanning many years or even decades in the face of broad range of threats that are possible over such long periods. Such threats to data durability include conventional hardware and software faults, environmental disruptions, organizational failures, and administrative failures caused by human error or malice. In this chapter, we present SafeStore, a distributed storage system designed to maintain long-term data durability using the principle of aggressive *fault isolation* along administrative, physical, and temporal dimensions.

5.1 Introduction

The design of storage systems that provide data durability on the time scale of decades is an increasingly important challenge as more valuable information is stored digitally [14, 49, 114]. For example, data from the National Archives and Records Administration indicate that 93% of companies go bankrupt within a year if they lose their data center in some disaster [7], and a growing number of government laws [12, 32] mandate multi-year periods of data retention for many types of information [16, 104].

Against a backdrop in which over 34% of companies fail to test their tape backups [6] and over 40% of individuals do not back up their data at all [43], multi-decade scale durable storage raises two technical challenges. First, there exist a broad range of threats to data durability including media failures [105, 120, 133], software bugs [106, 135], malware [27, 125], user error [104, 117], administrator error [73, 101],

organizational failures [34, 38], malicious insiders [37, 51], and natural disasters on the scale of buildings [9] or geographic regions [15]. Requiring robustness on the scale of decades magnifies them all: threats that could otherwise be considered negligible must now be addressed. Second, such a system has to be practical with cost, performance, and availability competitive with traditional systems.

Storage outsourcing is emerging as a popular approach to address some of these challenges [75]. By entrusting storage management to a Storage Service Provider (SSP), where "economies of scale" can minimize hardware and administrative costs, individual users and small to medium-sized businesses seek cost-effective professional system management and peace of mind vis-a-vis both conventional media failures and catastrophic events.

Unfortunately, relying on an SSP is no panacea for long-term data integrity. SSPs face the same list of hard problems outlined above and as a result even brand-name ones [13, 18] can still lose data. To make matters worse, clients often become aware of such losses only after it is too late. This opaqueness is a symptom of a fundamental problem: SSPs are separate administrative entities and the internal details of their operation may not be known by data owners. While most SSPs may be highly competent and follow best practices punctiliously, some may not. By entrusting their data to back-box SSPs, data owners may free themselves from the daily worries of storage management, but they also relinquish ultimate control over the fate of their data. In short, while SSPs are an economically attractive response to the costs and complexity of long-term data storage, they do not offer their clients any end-to-end guarantees on data durability, which we define as the probability that a specific data object will not be lost or corrupted over a given time period.

To achieve high durability, SafeStore applies aggressively the principle of *fault isolation* without compromising practicality in terms of cost, performance, and availability.

Aggressive isolation for durability. SafeStore stores data redundantly across multiple SSPs and leverages diversity across SSPs to prevent permanent data loss caused by isolated administrator errors, software bugs, insider attacks, bankruptcy, or natural catastrophes. With respect to data stored at each SSP, SafeStore employs a "trust but verify" approach: it does not interfere with the policies used within each SSP to maintain data integrity, but it provides an *audit* interface so that data owner retain end-to-end control over data integrity. The audit mechanism can quickly detect data loss and trigger data recovery from redundant storage before additional faults result in unrecoverable loss. Finally, to guard data stored at SSPs against faults at the data owner site (e.g. operator errors, software bugs, and malware attacks), SafeStore restricts the interface to provide temporal isolation between clients and SSPs so that the latter export the abstraction of write-once-read-many storage.

Making aggressive isolation practical. SafeStore introduces an efficient storage interface to reduce network bandwidth and storage cost using an *informed hierarchical erasure coding* scheme, that, when applied across and within SSPs, can achieve near-optimal durability. SafeStore SSPs expose redundant encoding options to allow the system to efficiently divide storage redundancies across and within SSPs. Additionally, SafeStore limits the cost of implementing its "trust but verify" policy through an audit protocol that shifts most of the processing to the audited SSPs and encourages them proactively measure and report any data loss they experience. Dishonest SSPs are quickly caught with high probability and at little cost to the auditor using probabilistic spot checks. Finally, to reduce the bandwidth, performance, and availability costs of implementing geographic and administrative isolation, SafeStore implements a two-level storage architecture where a local server (possibly running on the client machine) is used as a soft-state cache, and if the local server crashes, SafeStore limits down-time by quickly recovering the critical meta data from the remote SSPs while the actual data is being recovered in the background.

Contributions. We present a highly durable storage architecture that uses a new replication interface to distribute data efficiently across diverse set of SSPs and an effective audit protocol to check data integrity. We demonstrate that this approach can provide high durability in a way that is practical and economically viable with cost, availability, and performance competitive with traditional systems. We demonstrate these ideas by building and evaluating SSFS, an NFS-based SafeStore storage system. Overall, we show that SafeStore provides an economical alternative to realize multi-decade scale durable storage for individuals and small-to-medium sized businesses with limited resources. Note that although we focus our attention on outsourced SSPs, the SafeStore architecture could also be applied internally by large enterprises that maintain multiple isolated data centers.

5.2 Architecture and Design Principles

The main goal of SafeStore is to provide extremely durable storage over many years or decades.

5.2.1 Threat model

Over such long time periods, even relatively rare events can affect data durability, so we must consider broad range of threats along multiple dimensions—physical, administrative, and software.

Physical faults: Physical faults causing data loss include disk media faults [61, 133], theft [33], fire [9], and wider geographical catastrophes [15]. These faults can result in data loss at a single node or spanning multiple nodes at a site or in a region.

Administrative and client-side faults: Accidental misconfiguration by system administrators [73, 101], deliberate insider sabotage [37, 51], or business failures leading to bankruptcy [34] can lead to data corruption or loss. Clients can also delete data accidentally by, for example, executing “rm -r *”. Administrator and client faults can be particularly devastating because they can affect replicas across otherwise isolated

subsystems. For instance [37], a system administrator not only deleted data but also stole the only backup tape after he was fired, resulting in financial damages in excess of $10 million and layoff of 80 employees.

Software faults: Software bugs [106, 135] in file systems, viruses [27], worms [125], and Trojan horses can delete or corrupt data. A vivid example of threats due to malware is the recent phenomenon of ransomware [30] where an attacker encrypts a user's data and withholds the encryption key until a ransom is paid.

Of course, any of the listed faults may occur rarely. But at the scale of decades, it becomes risky to assume that no rare events will occur. It is important to note that some of these failures [9, 105, 120] are often correlated resulting in simultaneous data loss at multiple nodes while others [106] are more likely to occur independently.

Replication mechanisms optimized for one or the other type of failure may not be optimal in this setting where both failure types can happen.

Limitations of existing practice. Most existing approaches to data storage face two problems that are particularly acute in our target environments of individuals and small/medium businesses: (1) they depend too heavily on the operator or (2) they provide insufficient fault isolation in at least some dimensions.

For example, traditional removable-media-based-systems (e.g., tape, DVD-R) systems are labor intensive, which hurts durability in the target environments because users frequently fail to back their data up, fail to transport media off-site, or commit errors in the backup/restore process [35]. The relatively high risk of robot and media failures [3] and slow mean time to recover [79] are also limitations.

Similarly, although on-site disk-based [4, 23] backup systems speed backup/recovery, use reliable media compared to tapes, and even isolate client failures by maintaining multiple versions of data, they are vulnerable to physical site, administrative, and software failures.

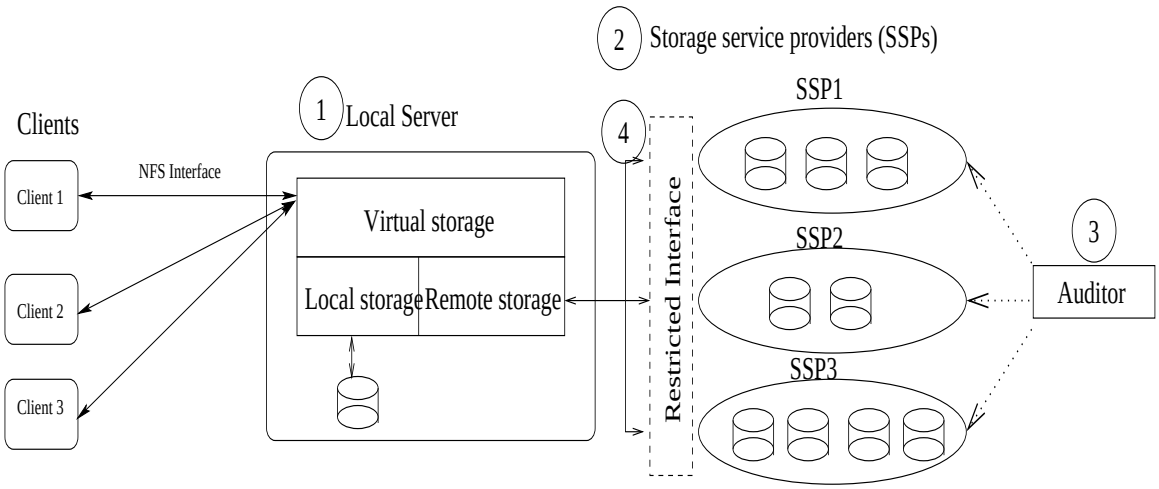


Figure 5.1: SafeStore architecture

Finally, network storage service providers (SSPs) [1, 2, 22, 31] are a promising alternative as they provide geographical and administrative isolation from users and they ride the technology trend of falling network and hardware costs to reduce the data-owner's effort. But they are still vulnerable to administrative failures at the service providers [13], organizational failures (e.g., bankruptcy [34, 75]), and operator errors [38]. They thus fail to fully meet the challenges of a durable storage system. We do, however, make use of SSPs as a component of SafeStore.

5.2.2 SafeStore architecture

As shown in Figure 5.1, SafeStore uses the following design principles to provide high durability by tolerating the broad range of threats outlined above while keeping the architecture practical, with cost, performance, and availability competitive with traditional systems.

Efficiency via 2-level architecture. SafeStore uses a two-level architecture in which the data owner's *local server* (①in Figure 5.1) acts as a cache and write buffer while durable storage is provided by multiple remote *storage service providers* SSPs ②. The local server could be running on the client's machine or a different machine. This division of labor has two consequences. First, performance, availability, and network cost are improved because most accesses are served locally; we show this is crucial in Section 5.3. Second, management cost is improved because the requirements on the local system are limited (local storage is soft state, so local failures have limited consequences) and critical management challenges are shifted to the SSPs, which can have excellent economies of scale for managing large data storage systems [1, 36, 75].

Aggressive isolation for durability. We apply the principle of aggressive isolation in order to protect data from the broad range of threats described above.

- *Autonomous SSPs:* SafeStore stores data redundantly across multiple autonomous SSPs (② in Figure 5.1). Diverse SSPs are chosen to minimize the likelihood of common-mode failures across SSPs. For example, SSPs can be external commercial service providers [1, 2, 22, 31], that are geographically distributed, run by different companies, and based on different software stacks. Although we focus on *out-sourced* SSPs, large organizations can use our architecture with *in-sourced* storage across autonomous entities within their organization (e.g., different campuses in a university system.)
- *Audit:* Aggressive isolation alone is not enough to provide high durability as data fragment failures accumulate over time. On the contrary, aggressive isolation can adversely affect data durability because the data owner has little ability to enforce or monitor the SSPs' internal design or operation to ensure that SSPs follow best practices. We provide an end-to-end audit interface (③in Figure 5.1) to detect data loss and thereby bound mean time to recover (MTTR), which in turn increases mean time

to data loss (MTTDL). In Section 5.4 we describe our audit interface and show how audits limit the damage that poorly-run SSPs can inflict on overall durability.

- *Restricted interface:* SafeStore must minimize the likelihood that erroneous operation of one subsystem compromises the integrity of another [95]. In particular, because SSPs all interact with the local server, we must restrict that interface. For example, we must protect against careless users, malicious insiders, or devious malware at the clients or local server that mistakenly delete or modify data. SafeStore's restricted SSP interface ④ provides temporal isolation via the abstraction of versioned write-once-read-many storage so that a future error cannot damage existing data.

Making isolation practical. Although durability is our primary goal, the architecture must still be economically viable.

- *Efficient data replication:* The SafeStore architecture defines a new interface that allows the local server to realize near-optimal durability using *informed hierarchical erasure coding* mechanism, where SSPs expose internal redundancy. Our interface does not restrict SSP's autonomy in choosing internal storage organization (replication mechanism, redundancy level, hardware platform, software stack, administrative policies, geographic location, etc.) Section 5.3 shows that our new interface and replication mechanism provides orders of magnitude better durability than *oblivious hierarchical encoding* based systems using existing black-box based interfaces [1, 2, 31].
- *Efficient audit mechanism:* To make audits of SSPs practical, we use a novel audit protocol that, like real world financial audits, uses self-reporting whereby auditor offloads most of the audit work to the auditee (SSP) in order to reduce the overall system resources required for audits. However, our audit takes the form of a challenge-response protocol with occasional spot-checks that ensure that an

auditee that generates improper responses is quickly discovered and that such a discovery is associated with a cryptographic proof of misbehavior [46].

- *Other optimizations:* We use several optimizations to reduce overhead and downtime in order to make system practical and economically viable. First, we use a fast recovery mechanism to quickly recover from data loss at a local server where the local server comes online as soon as the meta-data is recovered from remote SSPs even while data recovery is going on in the background. Second, we use block level versioning to reduce storage and network overhead involved in maintaining multiple versions of files.

5.2.3 Economic viability

In this section, we consider the economic viability of our storage system architecture in two different settings, outsourced storage using commercial SSPs and federated storage using in-house but autonomous SSPs, and calibrate the costs by comparing with a less-durable local storage system.

	Standalone	SafeStore In-house	SafeStore SSP (Cost+Profit)
Storage	$30/TB/month [36]	$30/TB/month [36]	$150/TB/month [1]
Network	NA	$200/TB [24]	$200/TB [1]
Admin	1 admin/[1,10,100]TB [102]	1 admin/100TB [102]	Included [1]

Table 5.1: System cost assumptions. Note that a *Standalone* system makes no provision for isolated backup and is used for cost comparison only. Also, we take into consideration the variable administrative cost for *Standalone* system [102] used by inefficient (1 admin per 1 TB of data stored), typical (1 admin per 10 TB), and efficient (1 admin per 100 TB) internet services.

We consider three components to storage cost: hardware resources, administration, and—for outsourced storage—profit. Table 5.1 summarizes our basic assumptions for a straw-man *Standalone* local storage system and for the local owner and SSP parts of a SafeStore system. In column B, we estimate the raw hardware and administrative costs that might be paid by an in-house SSP. We base our storage hardware costs on estimated full-system 5-year total cost of ownership (TCO) costs in 2006 for large-scale internet

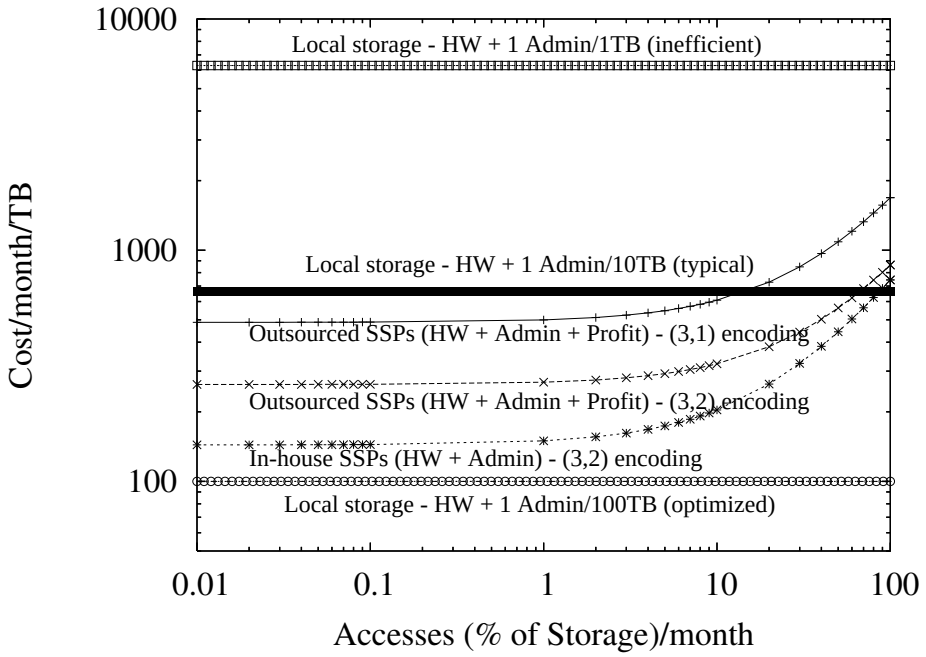


Figure 5.2: Comparison of SafeStore cost v. accesses to remote storage (as a percentage of straw-man Standalone local storage) varies.

services such as Internet Archive [36]. Note that using the same storage cost for a large-scale, specialized SSP and for smaller data owners and Standalone systems is conservative in that it may overstate the relative additional cost of adding SSPs. For network resources, we base our costs on published rates in 2006 [24]. For administrative costs, we use Gray's estimate that highly efficient internet services require about 1 administrator to manage 100TB while smaller enterprises are typically closer to one administrator per 10TB but can range from one per 1TB to 1 per 100TB [102] (Gray notes, "But the real cost of storage is management" [102]). Note that we assume that by transforming local storage into a soft-state cache, SafeStore simplifies local storage administration. We therefore estimate local hardware and administrative costs at 1 admin per 100TB.

In Figure 5.2, the storage cost of in-house SSP includes SafeStore's hardware (cpu, storage, network) and administrative costs. We also plot the straw-man local storage system with 1, 10, or 100 TB per administrator. The outsourced SSP lines show SafeStore costs assuming SSPs prices include a profit by using Amazon's S3 storage service pricing. Three points stand out. First, additional replication to SSPs increases cost (as inter-SSP data encoding, as discussed in section 5.3, is raised from (3,2) to (3,1)), and the network cost rises rapidly as the remote access rate increases. These factors motivate SafeStore's architectural decisions to (1) use efficient encoding and (2) minimize network traffic with a large local cache that fully replicates all stored state. Second, when SSPs are able to exploit economies of scale to reduce administrative costs below those of their customers, SafeStore can reduce overall system costs even when compared to a less-durable Standalone local-storage-only system. Third, even for customers with highly-optimized administrative costs, as long as most requests are filtered by the local cache, SafeStore imposes relatively modest additional costs that may be acceptable if it succeeds in improving durability.

The rest of the chapter is organized as follows. First, in Section 5.3 we present and and evaluate our novel *informed hierarchical erasure coding* mechanism. In Section 5.4, we address SafeStore's audit protocol. Later, in Section 5.5 we describe the SafeStore interfaces and implementation. We evaluate the prototype in Section 5.6. Finally, we present the related work in Section 5.7.

5.3 Data replication interface

This section describes a new replication interface to achieve near-optimal data durability while limiting the internal details exposed by SSPs, controlling replication cost, and maximizing fault isolation.

As shown in Figure 5.3, SafeStore uses hierarchical encoding comprising inter-SSP and intra-SSP redundancy: First, it stores data redundantly across different SSPs, and then each SSP internally replicates data entrusted to it as it sees fit. Hierarchical encoding is the natural way to replicate data in our setting as it tries to maximize fault-isolation across SSPs while allowing SSP's autonomy in choosing an appropriate internal

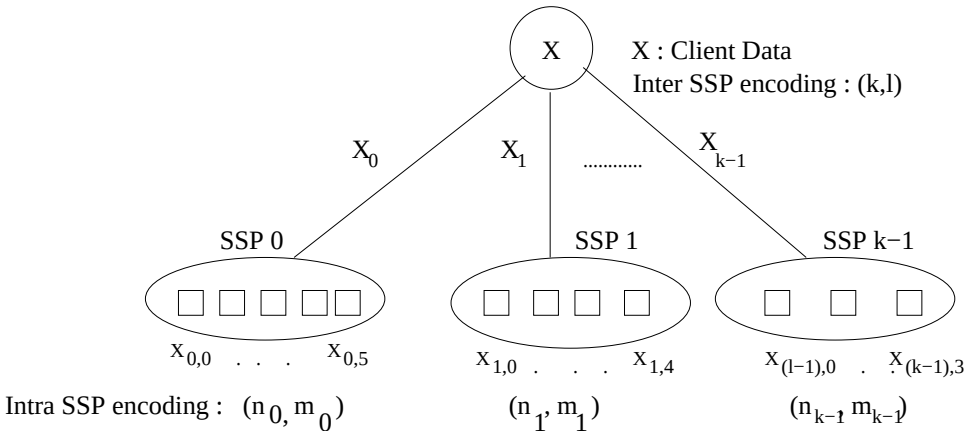


Figure 5.3: Hierarchical encoding

data replication mechanism. Different replication mechanisms such as erasure coding [110], RAID [61], or full replication can be used to store data redundantly at inter-SSP and intra-SSP levels (any replication mechanism can be viewed as some form of (k,l) encoding [129] from durability perspective, where l out of k encoded fragments are required to reconstruct data). However, it requires proper balance between inter-SSP and intra-SSP redundancies to maximize end-end durability for a fixed storage overhead. For example, consider a system willing to pay an overall 6x redundancy cost using 3 SSPs with 8 nodes each. If, for example, each SSP only provides the option of (8,2) intra-SSP encoding, then we can use at most (3,2) inter-SSP encoding. This combination gives gives 4 9's less durability for the same overhead compared to a system that uses (3,1) encoding at the inter-SSP level and (8,4) encoding at the intra-SSP level at all the SSPs.

5.3.1 Model

The overall storage overhead to store a data object is $(n_0/m_0 + n_1/m_1 + ...n_{k-1}/m_{k-1})/l$, when a data object is hierarchically encoded (as shown in Figure 5.3) using (k,l) erasure coding across k SSPs, and SSPs 0 through $k-1$ internally use erasure codings (n_0,m_0), (n_1,m_1),....(n_{k-1},m_{k-1}), respectively. We assume that

the number of SSPs(k) is fixed and a data object is (possibly redundantly) stored at all SSPs. We do not allow varying k as it requires additional internal information about various SSPs (MTTF of nodes, number of nodes, etc.) which may not be available in order to choose optimal set of k nodes. Instead, we tackle the problem of finding optimal distribution of inter-SSP and intra-SSP redundancies for a fixed k. The end-to-end data durability, as explained in Appendix B, can be estimated analytically as a function of these variables using following analytical model that considers two classes of faults. *Node faults* (e.g. physical faults like sector failures, disk crashes, etc.) occur within an SSP and affect just one fragment of an encoded object stored at the SSP. *SSP faults* (e.g., administrator errors, organizational failures, geographical failures, etc.) are instead simultaneous or near-simultaneous failures that take out all fragments across which an object is stored within an SSP.

To illustrate the approach, we consider a baseline system consisting of 3 SSPs with 8 nodes each. We use a baseline MTTDL of 10 years due to invidual node faults and 100 years for SSP failures and assume both are independent and identically distributed. We use MTTR of data of 2 days (e.g. to detect and replace a faulty disk) for node faults and 10 days for SSP failures. We use the probability of data loss of an object during a 10 year period to characterize durability because expressing end-to-end durability as MTTDL can be misleading [61] (although MTTDL can be easily computed from the probability of data loss as shown in Appendix B. Later, we change the distribution of nodes across SSPs, MTTDL and MTTR of node failures within SSPs, to model diverse SSPs. The conclusions that we draw here are general and not specific to this setup; we find similar trends when we change the total number of nodes, as well as MTTDL and MTTR of correlated *SSP faults*.

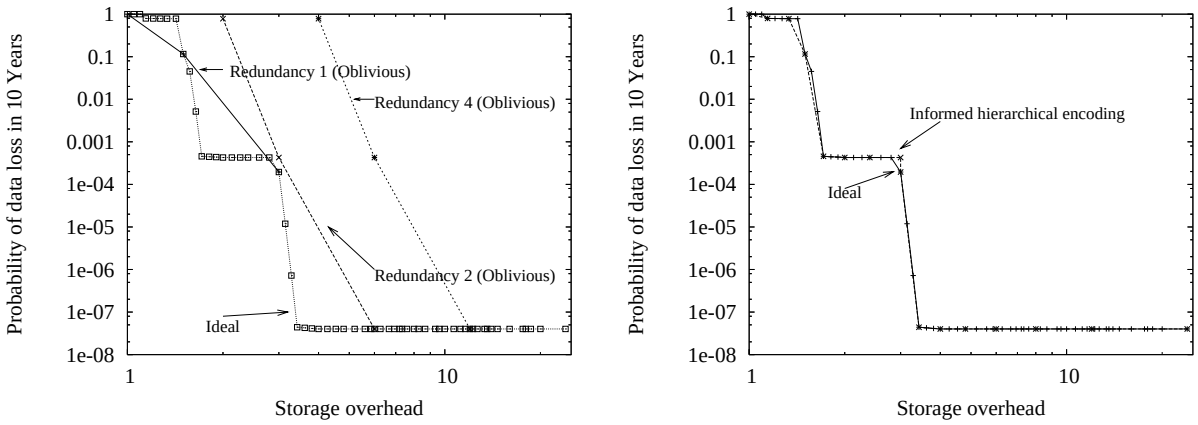


Figure 5.4: (a) Durability with Black-box interface with fixed intra-SSP redundancy (b) Informed hierarchical encoding

5.3.2 Informed hierarchical encoding

A client can maximize end-to-end durability if it can control both intra-SSP and inter-SSP redundancies. However, current black-box storage interfaces exported by commercial outsourced SSPs [1, 2, 31] do not allow clients to change intra-SSP redundancies. With such a black-box interface, clients perform *oblivious hierarchical encoding* as they control only inter-SSP redundancy. Figure 5.4(a) plots the optimal durability achieved by an *ideal* system that has full control of inter-SSP and intra-SSP redundancy and a system using *oblivious hierarchical encoding*. The latter system has 3 lines for different fixed intra-SSP redundancies of 1, 2, and 4, where each line has 3 points for each of the 3 different inter-SSP encodings((3,1), (3,2) and (3,3)) that a client can choose with such a black-box interface. Two conclusions emerge. First, for a given storage overhead, the probability of data loss of an *ideal* system is often orders of magnitude lower than a system using *oblivious hierarchical encoding*, which therefore is several 9's short of optimal durability. Second, a system using *oblivious hierarchical encoding* often requires 2x-4x more storage than *ideal* to achieve the

same durability.

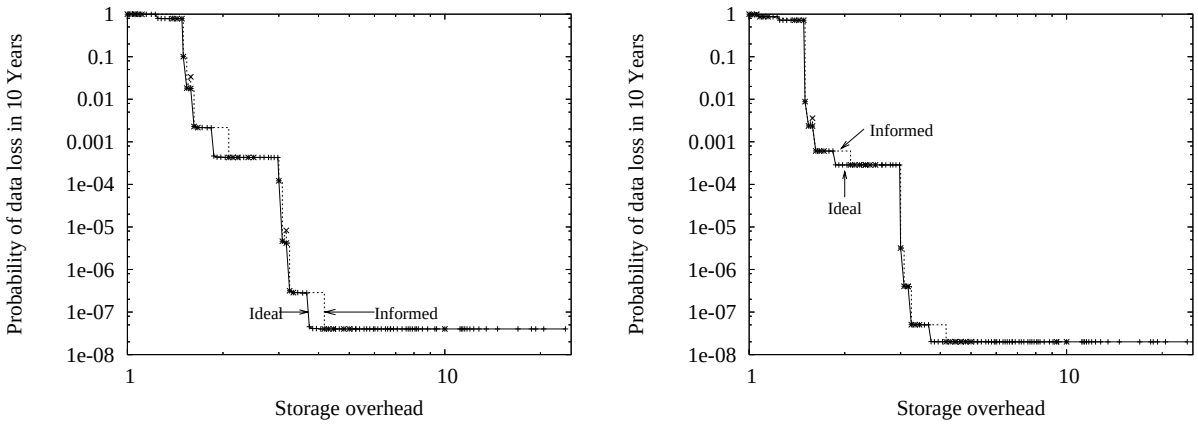
To improve on this situation, SafeStore describes an interface that allows clients to realize near-optimal durability using *informed hierarchical encoding* by exercising additional control on intra-SSP redundancies. With this interface, each SSP exposes the set of redundancy factors that it is willing to support. For example, an SSP with 4 internal nodes can expose redundancy factors of 1 (no redundancy), 1.33, 2, and 4 corresponding, respectively, to the (4,4), (4,3), (4,2) and (4,1) encodings used internally.

Our approach to achieve near-optimal end-to-end durability is motivated by the stair-like shape of the curve tracking the durability of *ideal* as a function of storage overhead (Figure 5.4(a)). For a fixed storage overhead, there is a tradeoff between inter-SSP and intra-SSP redundancies, as a given overhead O can be expressed as $1/l \times (r_0 + r_1 + ..r_{k-1})$, when (k,l) encoding is used across k SSPs in the system with intra-SSP redundancies of r_0 to r_{k-1} (where $r_i = n_i/m_i$). Figure 5.4(a) shows that durability increases dramatically (moving down one step in the figure) when inter-SSP redundancy increases, but does not improve appreciably when additional storage is used to increase intra-SSP redundancy beyond a threshold that is close to but greater than 1. This observation is backed by mathematical analysis as explained in observation 1 of Appendix B.

Hence, we propose a heuristic biased in favor of spending storage to maximize inter-SSP redundancy as follows:

- First, for a given number k of SSPs, we maximize the inter-SSP redundancy factor by minimizing l. In particular, for each SSP i, we choose the minimum redundancy factor $r'_i > 1$ exposed by i, and we compute l as $l = \lfloor (r'_0 + r'_1 + ...r'_{k-1})/O \rfloor$.

- Next, we distribute the remaining overhead ($O - 1/l \times (r'_0 + r'_1 + ..r'_{k-1})$) among the SSPs to minimize the standard deviation of the intra-SSP redundancy factors r_i that are ultimately used by the different



(a) (b)

Figure 5.5: (a) Informed hierarchical encoding with non-uniform distribution (b) Durability with different MTTDL and MTTR for node failures across SSPs

SSPs. We minimize standard deviation by initializing it to the lowest possible value (0%) and then distribute overhead across all intra-SSP redundancies so that the deviation is within the value and new intra-SSP redundancies are allowed by SSPs. If we do not find a possible set of allowable intra-SSP redundancies then we relax standard deviation constraint by increasing it gradually and follow the above step until we find an allowable set of intra-SSP redundancies.

The first rule is used to maximize inter-SSP redundancy and the second rule is to ensure that intra-SSP redundancies are uniformly distributed across SSPs. We try to distribute redundancy uniformly across all SSPs otherwise SSPs with small or no redundancy tend to loose objects faster and require expensive inter-SSP recovery to recover from such failures.

Figure 5.5(b) shows that this new approach, which we call *informed hierarchical coding*, achieves near optimal durability in a setting where three SSPs have the same number of nodes (8 each) and the same MTTDL and MTTR for internal node failures. These assumptions, however, may not hold in practice,

as different SSPs are likely to have a different number of nodes, with different MTTDLs and MTTRs. Figure 5.5(a) shows the result of an experiment in which SSPs have a different number of nodes—and, therefore, expose different sets of redundancy factors. We still use 24 nodes, but we distribute them non-uniformly (14, 7, 3) across the SSPs: informed hierarchical encoding continues to provide near-optimal durability. This continues to be true even when there is a skew in MTTDL and MTTR (due to node failures) across SSPs. For instance, Figure 5.5(b) uses the same non-uniform node distribution of Figure 5.5(a), but the (MTTDL, MTTR) values for node failures now differ across SSPs—they are, respectively, (10 years, 2 days), (5 years, 3 days), and (3 years, 5 days). Note that, by assigning the worst (MTTDL, MTTR) for node failures to the SSP with least number of nodes, we are considering a worst-case scenario for informed hierarchical encoding.

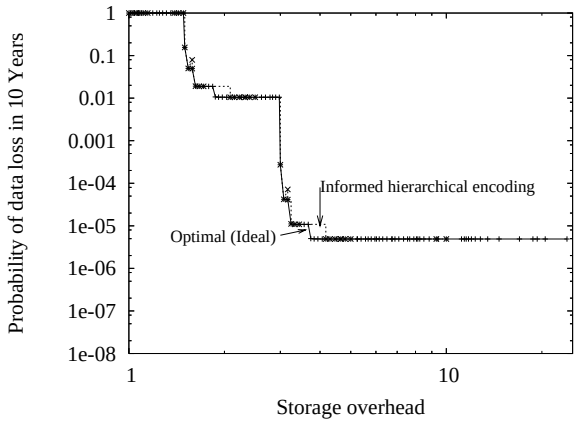


Figure 5.6: Informed hierarchical encoding with MTTDL of *correlated failures* set to 10 years with MTTR of 5 days,

We also study the sensitivity of our results to MTTDL and MTTR of *correlated failures* and total number of nodes in the system. All these results confirm the conclusion that a simple interface that allows SSPs to expose the redundancy factors they support is all it is needed to achieve, through our simple informed

hierarchical encoding mechanism, near optimal durability. As shown in Figures 5.6 and 5.7, our conclusions continues to hold: (1) when MTTDL due to *correlated failures* is changed to 10 years from 100 years and MTTR is changed from 10 days to 5 days, (2) when we increase the number of nodes, and (3) when they are non-uniformly distributed with increased number of nodes.

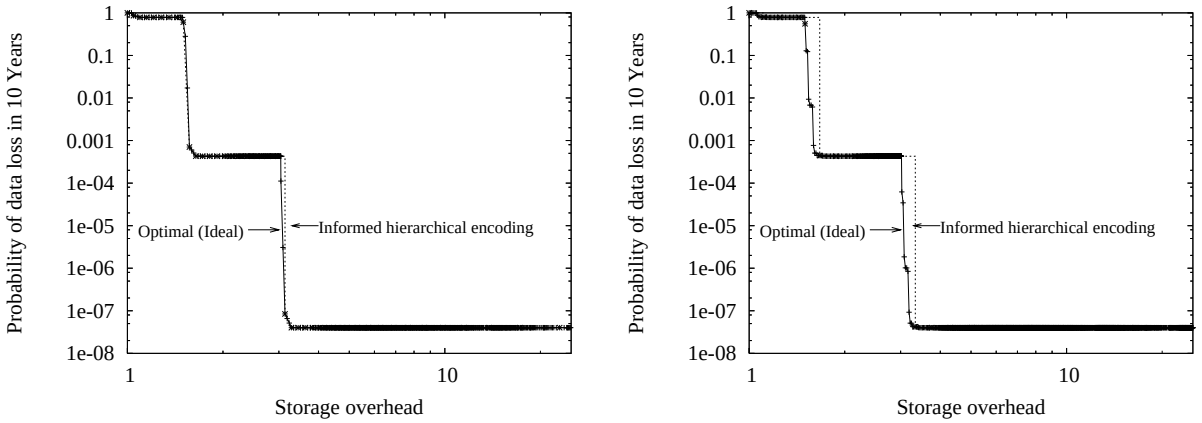


Figure 5.7: Informed hierarchical encoding (a) With 69 total nodes distributed uniformly across 3 SSPs, (b) With 69 nodes distributed non-uniformly across 3 SSPs with 10, 20, and 39 nodes each.

These results are not surprising in light of our discussion of Figure 5.4(a): durability depends mainly on maximizing inter-SSP redundancy and it is only slightly affected by the internal data management of individual SSPs.

SSPs can provide such an interface as part of their SLA (service level agreement) and charge clients based on the redundancy factor they choose when they store a data object. The interface is designed to limit the amount of detail that an SSP must expose about the internal organization. For example, an SSP with 1000 servers each with 10 disks might only expose redundancy options (1.0, 1.1, 1.5, 2.0, 4.0, 10.0), revealing little about its architecture. Note that the proposed interface could allow a dishonest SSP to cheat the client

by using less redundancy than advertised. The impact of such false advertising is limited by two factors: First, as observed above, our design is relatively insensitive to variations in intra-SSP redundancy. Second, the end to end audit protocol described in the next section limits the worst-case damage any SSP can inflict.

5.4 Audit

We need an effective audit mechanism to quickly detect data losses at SSPs so that data can be recovered before multiple component failures resulting in unrecoverable loss. An SSP *should* safeguard the data entrusted to it by following best practices like monitoring hardware health [122], spreading coded data across drives and controllers [61] or geographically distributed data centers, periodically scanning and correcting latent errors [121], and quickly notifying a data owner of any lost data so that the owner can restore the data from other SSPs and maintain a desired replication level. However, the principle of isolation argues against blindly assuming SSPs are flawless system designers and operators for two reasons. First, SSPs are separate administrative entities, and their internal details of operation may not be verifiable by data owners. Second, given the imperfections of software [27, 106, 135], operators [73, 101], and hardware [61, 133], even name-brand SSPs may encounter unexpected issues and silently lose customer data [13, 18]. Auditing SSP data storage embodies the end-to-end principle (in almost exactly the form it was first described) [116], and frequent auditing ensures a short Mean Time To Detect (MTTD) data loss, which helps limit worst-case Mean Time To Recover (MTTR). It is important to reduce MTTR in order to increase MTTDL as a good replication mechanism alone cannot improve MTTDL over a long time-duration spanning decades.

The technical challenge to auditing is to provide an end-to-end guarantee on data integrity while minimizing cost. These goals rule out simply reading stored data across the network as too expensive (see Figure 5.2) and, similarly, just retrieving a hash of the data as not providing an end-to-end guarantee (the SSP may be storing the hash not the data.). Furthermore, the audit protocol must work with data erasure-coded across SSPs, so a simple scheme that sends a challenge to multiple identical replicas and then compare

the responses such as those in LOCKSS [95] and Samsara [64] do not work. We must therefore devise an inexpensive audit protocol despite the fact that no two replicas store the same data.

To reduce audit cost, SafeStore's audit protocol borrows a strategy from real-world audits: we push most of the work onto the auditee and ask the auditor to spot check the auditee's reports. Our reliance on self-reporting by SSPs drives two aspects of the protocol design. First, the protocol is believed to be *shortcut free*–audit responses from SSPs are guaranteed to embody end-to-end checks on data storage– under the assumption that collision resistant modification detection codes [97] exist. Second, the protocol is *externally verifiable* and *non-repudiable*—falsified SSP audit replies are quickly detected (with high probability) and deliberate falsifications can be proven to any third party[1].

5.4.1 Audit protocol

The audit protocol proceeds in three phases: (1) data storage, (2) routine audit, and (3) spot check. Note that the auditor may be co-located with or separate from the owner. For example, audit may be outsourced to an external auditor when data owners are offline for extended periods. To authorize SSPs to respond to auditor requests, the owner signs a certificate granting audit rights to the auditor's public key, and all requests from the auditor are authenticated against such a certificate (these authentication handshakes are omitted in the description below.) We describe the high level protocol here and detail it in Appendix C.

Data storage. When an object is stored at an SSP, the SSP signs and returns to the data owner a *receipt* that includes the object ID, cryptographic hash of the data, and storage expiration time. The data owner in turn verifies that the signed hash matches the data it sent and that the receipt is not malformed with an incorrect id or expiration time. If the data and hash fail to match, the owner retries sending the write message

[1]We assume that provably deliberate falsification can be punished via contractual or other out-of-band means [107], but details are outside the scope of this paper.

(data could have been corrupted in the transmission); repeated failures indicate a malfunctioning SSP and generate a notification to the data owner. As we detail in Section 5.5, SSPs do not provide a delete interface, so the expiration time indicates when the SSP will garbage collect the data. The data owner collects such valid receipts, encodes them, and spreads them across SSPs for durable storage.

Routine audit. The auditor sends to an SSP a list of object IDs and a random challenge. The SSP computes a cryptographic hash on both the challenge and the data. The SSP sends a signed message to the auditor that includes the object IDs, the current time, the challenge, and the hash computed on the challenge and the data ($H(challenge + data_{objId})$). The auditor buffers the challenge responses if the messages are well-formed, where a message is considered to be well-formed if none of the following conditions are true: the signature does not match the message, the response with an unacceptably stale timestamp, the response with the wrong challenge, or the response indicates error code (e.g., he SSP detected data is corrupt via internal checks or the data has expired). If the auditor does not receive any response from the SSP or if it receives a malformed message, the auditor notifies the data owner, and the data owner reconstructs the data via cached state or other SSPs and stores the lost fragment again. Of course, the owner may choose to switch SSPs before restoring the data and/or may extract penalties under their service level agreement (SLA) with the SSP, but such decisions are outside the scope of the protocol.

We conjecture that the audit response is shortcut free: an SSP must possess object's data to compute the correct hash. An honest SSP verifies the data integrity against the challenge-free hash stored at the creation time before sending a well-formed challenge response. If the integrity check fails (data is lost or corrupted) it sends the error code for lost data to the auditor. However, a *dishonest* SSP can choose to send a syntactically well-formed audit response with bogus hash value when the data is corrupted or lost. Note that the auditor just buffers well-formed messages and does not verify the integrity of the data objects covered by the audit in this phase. Yet, routine audits serve two key purposes. First, when performed against honest SSPs, they

provide end-to-end guarantees about the integrity of the data objects covered by the audit. Second, they force dishonest SSPs to produce a signed, non-repudiable statement about the integrity of the data objects covered by the audit.

Spot check. In each round, after it receives audit responses in the routine audit phase, the auditor randomly selects α% of the objects to be spot checked. The auditor then retrieves each object's data (via the owner's cache, via the SSP, or via other SSPs) and verifies that the cryptographic hash of the challenge and data matches the challenge response sent by the SSP in the routine audit phase. If there is a mismatch, the auditor informs the data owner about the mismatch and provides the signed audit response sent by the SSP. The data owner then can create an externally-verifiable proof of misbehavior (POM) [83] against the SSP: the receipt, the audit response, and the object's data. In particular, the receipt is a signed statement with a hash of the data; the audit reply a signed claim to be storing the data and that a hash across a challenge and the data has a particular value; and the data allows anyone to verify that the receipt and audit reply refer to that data but that the challenge computation was incorrect. Note that SafeStore local server encrypts all data before storing it to SSPs, so this proof may be presented to third parties without leaking the plaintext object contents. Also, note that our protocol works with erasure coding as the auditor can reconstruct the data to be spot checked using redundant data stored at other SSPs.

5.4.2 Durability and cost

In this section we examine how the low-cost audit protocol limits the damage from faulty SSPs. The SafeStore protocol specifies that SSPs notify data owners immediately of any data loss that the SSP cannot internally recover so that the owner can restore the desired replication level using redundant data. Figures 5.4 and 5.5 illustrate the durability of our system when the SSPs follow the requirement and immediately report

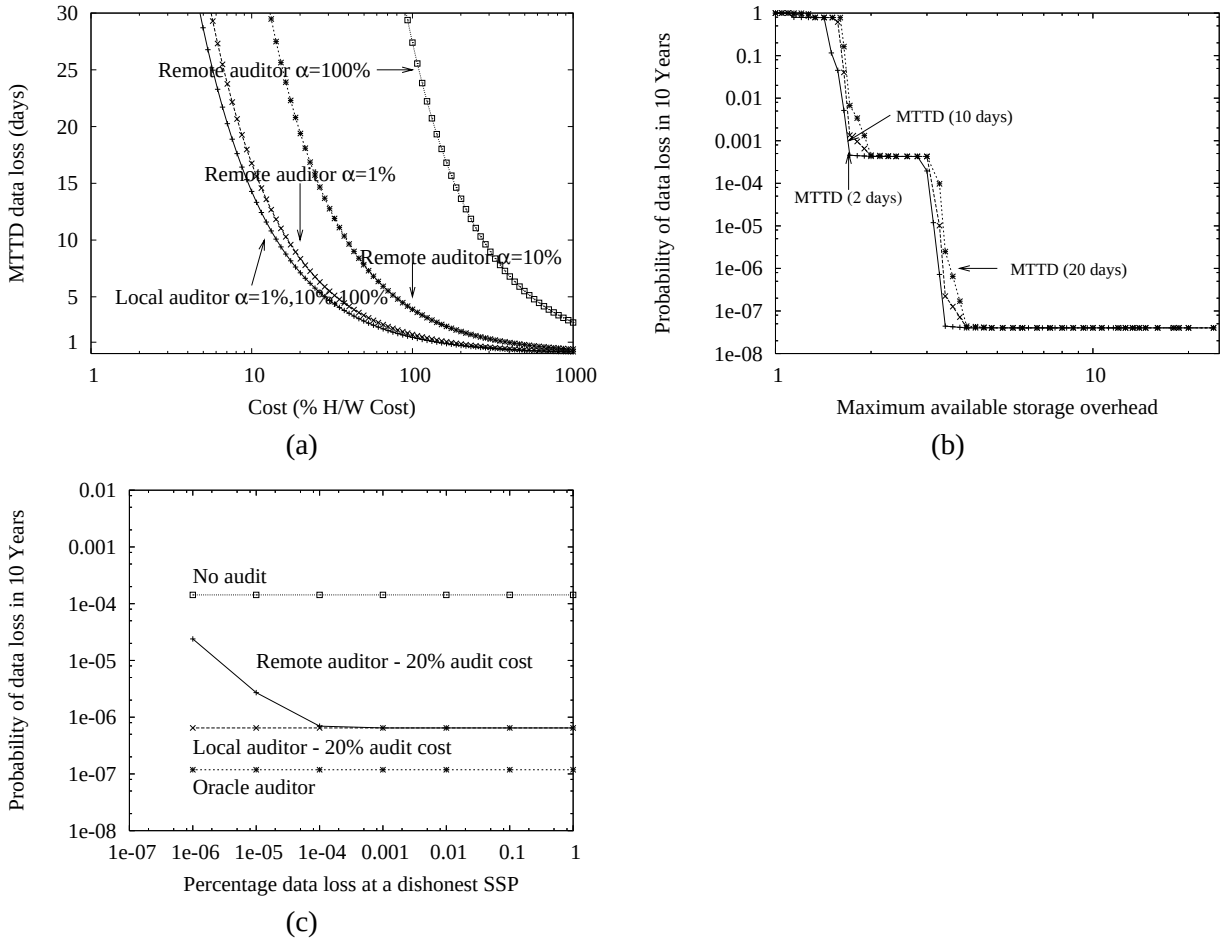


Figure 5.8: (a) Time to detect SSP data loss via audit with varying amounts of resources dedicated to audit overhead assuming honest SSPs. (b) Durability with varying MTTD. (c) Impact on overall durability with a dishonest SSP. In (a) and (c) , we use the same hardware cost model as in Figure 5.2 for disk capacity and WAN network transfers, add a cost of $0.031 per million operations for cryptographic operations–based on cryptographic benchmark results [5] for AMD opteron and using a conservative estimate of cpu cost of $850 (for a branded 1U rack server cost [42] which includes 1TB disk cost although we already included storage cost) with a 5 year TCO, add a cost of $0.027 per million IO operations for disk reads – using a conservative estimate for disk cost of $1000/TB with 100 operations/sec with a 10 year life time, and assume 20% of the SSP's data are read/written per month by the owner (separate from audits). In (c) we assume auditing is given upto 20% of total storage cost.

failures. As explained below, Figure 5.8-(a) and (b) show that SafeStore still provides excellent data durability with low audit cost, if a data owner is unlucky and selects a *passive* SSP that violates the immediate-notify requirement and waits for an audit of an object to report that it is missing. Figure 5.8-(c) shows that if a data owner is really unlucky and selects a *dishonest* SSP that first loses some of the owner's data and then lies when audited to try to conceal that fact, the owner's data is still very likely to emerge unscathed. We evaluate our audit protocol with 1TB of data stored redundantly across three SSPs with inter-SSP encoding of (3,1) (Appendix E has results for (3,2) encoding).

5.4.3 Protocol analysis when SSPs are altruistic

First, assume that SSPs are *passive* and wait for an audit to check data integrity. Because the protocol uses relatively cheap processing at the SSP to reduce data transfers across the wide area network, it is able to scan through the system's data relatively frequently without raising system costs too much. Figure 5.8-(a) plots the mean time to detect data loss (MTTD) at a *passive* SSP as a function of the cost of hardware resources (storage, network, and cpu) dedicated to auditing, expressed as a percentage of the cost of the system's total hardware resources as detailed in the caption. We also vary the fraction of objects that are spot checked in each audit round (α) for both the cases with local (co-located with the data owner) and remote (separated over WAN) auditors. We reach following conclusions: (1) As we increase the audit budget we can audit more frequently and the time to detect data loss falls rapidly. (2) audit costs with local and remote auditors is almost the same when α is less than 1%. (3) The audit cost with local auditor does not vary much with increasing α (as there is no additional network overhead in retrieving data from the local data owner) whereas the audit cost for the remote auditor increases with increasing α (due to additional network overhead in retrieving data over the WAN). (4) Overall, if a system dedicates 20% of resources to auditing, we can detect a lost data block within a week (with a local or a remote auditor with α = 1%).

Given this information, Figure 5.8-(b) shows the modest impact on overall data durability of increasing the time to detect and correct such failures when we assume that all SSPs are *passive* and SafeStore relies on auditing rather than immediate self reporting to trigger data recovery.

5.4.4 Protocol analysis when SSPs are selfish

Now consider the possibility of an SSP trying to brazen its way through an audit of data it has lost using a made-up value purporting to be the hash of the challenge and data. The BAR model [46] argues for reasoning about systems spanning multiple administrative domains by assuming that most entities are rational and will act to maximize their utility and that a small number may be Byzantine and may act arbitrarily.he audit protocol encourages rational SSPs that lose data to respond to audits honestly. In particular, we prove the following theorem in Appendix D that under reasonable assumptions about the penalty for an honest failure versus the penalty for generating a proof of misbehavior (POM), a rational SSP will maximize its utility [46] by faithfully executing the audit protocol as specified.

Theorem 2. *SafeStore audit protocol ensures that the rational SSPs (SSPs can selfishly deviate from the protocol to maximize their own benefits) follow the protocol by (1) attempting to store data reliably and (2) responds to audit requests honestly assuming an SLA that specifies appropriate penalties relative to the underlying cost of storing data. cost model.*

5.4.5 Protocol analysis when SSPs are Byzantine

But suppose that through misconfiguration, malfunction, or malice, a node first loses data and then issues *dishonest* audit replies that claim that the node is storing a set of objects that it does not have. The spot check protocol ensures that if a node is missing even a small fraction of the objects, such cheating is quickly discovered with high probability. Furthermore, as that fraction increases, the time to detect falls rapidly. The intuition is simple: the probability of detecting a dishonest SSP in k audits is given by

$$p_k = 1 - (1-p)^k$$

where p is the probability of detection in an audit, which is given by

$$p = \frac{\sum_{i=1}^{m} \binom{n}{i}\binom{N-n}{m-i}}{\binom{N}{m}}, (\text{if } n \geq m)$$

$$p = \frac{\sum_{i=1}^{n} \binom{m}{i}\binom{N-m}{n-i}}{\binom{N}{n}}, (\text{if } n < m)$$

where N is the total number of data blocks stored at an SSP, n is the number of blocks that are corrupted or lost and m is the number of blocks that are spot checked, α=(m/N) $\times$ 100.

Figure 5.8-(c) shows the overall impact on durability if a node that has lost a fraction of objects maximizes the time to detect these failures by generating *dishonest* audit replies. We fix the audit budget at 20% and measure the durability of SafeStore with local auditor (with α at 100%) as well as remote auditor (with α at 1%). We also plot the durability with *oracle detector* which detects the data loss immediately and triggers recovery. Note that the *oracle detector* line shows worse durability than the lines in Figure 5.8-(b) because (b) shows durability for a randomly selected 10-year period while (c) shows durability for a 10-year period that begins when one SSP has already lost data. Without auditing (*no audit*), there is significant risk of data loss reducing durability by three 9's compared to *oracle detector*. Using our audit protocol with *remote auditor*, the figure shows that a cheating SSP can introduce a non-negligible probability of small-scale data loss because it takes multiple audit rounds to detect the loss as it spot checks only 1% of data blocks. But that the probability of data loss falls quickly and comes closer to *oracle detector* line (with in one 9 of durability) as the amount of data at risk rises. Finally, with a *local auditor*, data loss is detected in one audit round independent of data loss percentage at the dishonest SSPs as a local auditor can spot check all the data. In the presence of dishonest SSPs, our audit protocol improves durability of our system by two 9's

WriteReceipt **write**(ID oid, byte data[], int64 size, int32 type, int64 expire);
ReadReply **read**(ID oid, int64 size, int32 type)
AttrReply **get_attr**(ID oid);
TTLReceipt **extend_expire**(ID oid, int64 expire);

Table 5.2: SSP storage interface

over a system with no audit at an additional audit cost of just 20%. The overall durability of our system improves with increasing audit budget and approaches the *oracle detector* line as described in Appendix E.

5.5 SSFS

We implement SSFS, a file system that embodies the SafeStore architecture and protocol. In this section, we first describe the SSP interface and our SSFS SSP implementation. Then, we describe SSFS's local server.

5.5.1 SSP

As Figure 5.1 shows, for long-term data retention SSFS local servers store data redundantly across administratively autonomous SSPs using erasure coding or full replication. SafeStore SSPs provide a simple yet carefully defined object store interface to local servers as shown in Table 5.2.

Two aspects of this interface are important. First, it provides non-repudiable receipts for writes and expiration extensions in order to support our spot-check-based audit protocol. Second, it provides *temporal isolation* to limit the data owner's ability to change data that is currently stored [95]. In particular, the SafeStore SSP protocol (1) gives each object an absolute expiration time and (2) allows a data owner to extend but not reduce an object's lifetime.

The temporal isolation guarantee is as follows: if an SSP is storing a desired set of data at time t, an owner can ensure that the current version is accessible until any desired time in the future even if the local server suffers an arbitrary failure.

This interface supports what we expect to be a typical usage pattern in which an owner creates a ladder of backups at increasing granularity [117]. Suppose the owner wishes to maintain yearly backups for each year in the past 10 years, monthly backups for each month of the current year, weekly backups for the last four weeks, and daily backups for the last week. Using the local server's snapshot facility (see Section 5.5.2), on the last day of the year, the local server *writes* all current blocks that are not yet at the SSP with an expiration date 10-years into the future and also iterates across the most recent version of all remaining blocks and sends *extend_expire* requests with an expiration date 10-years into the future. Similarly, on the last day of each month, the local server writes all new blocks and extends the most recent version of all blocks; notice that blocks not modified during the current year may already have expiration times beyond the 1-year target, but these extensions will not reduce this time. Similarly, on the last day of each week, the local server writes new blocks and extends deadlines of the current version of blocks for a month. And every night, the local server writes new blocks and extends deadlines of the current version of all blocks for a week. Of course, SSPs ignore *extend_expire* requests that would shorten an object's expiration time.

SSP implementation. We have constructed a prototype SSFS SSP that supports all of the features described in this paper including the interface for servers and the interface for auditors. Internally, each SSP spreads data across a set nodes using erasure coding with a redundancy level specified for each data owner's account at account creation time.

For compatibility with legacy SSPs, we also implement a simplified SSP interface that allows data owners to store data to Amazon's S3 [1], which provides a simple non-versioned read/write/delete interface and which does not support our optimized audit protocol.

Issues. There are three outstanding issues in our current implementation. We believe all are manageable. First, the approach relies on prompt failure/intrusion detection: the shorter the period of time between when a fault mistakenly deletes/modifies an object and the owner realizes that she would prefer an older version,

the more current backup that will be available. For simple failures (e.g., total disk failure), it will be easy for a data owner to quickly notice a problem. For more complex failures (e.g., malware that randomly modifies one bit in one file per day), detecting the problem is more difficult. We do not advance the state of the art in intrusion detection or fault detection, but we encourage data owners to make use of available tools [112, 122].

Second, in practice, it is likely that SSPs will provide some protocol for deleting data early. We assume that any such out-of-band early-delete mechanism is carefully designed to maximize resistance to erroneous deletion by the data owner. For concreteness, we assume that the payment stream for SSP services is well protected by the data owner and that our SSP will delete data 90 days after payment is stopped. So, a data owner can delete unwanted data by creating a new account, copying a subset of data from the old account to the new account, and then stopping payment on the old account. More sophisticated variations (e.g., using threshold-key cryptography to allow a quorum of independent administrators to sign off on a delete request) are possible.

Third, SSFS is vulnerable to resource consumption attacks: although an attacker who controls an owner's local server cannot reduce the integrity of data stored at SSPs, the attacker can send large amounts of long-lived garbage data and/or extend expirations farther than desired for large amounts of the owner's data stored at the SSP. We conjecture that SSPs would typically employ a quota system to bound resource consumption to within some budget along with an out-of-band early delete mechanism such as described in the previous paragraph to recover from any resulting denial of service attack.

5.5.2 Local Server

Clients interact with SSFS through a local server. The SSFS local server is a user level file system that exports the NFS 2.0 interface to its clients. The local server serves requests from local storage to improve

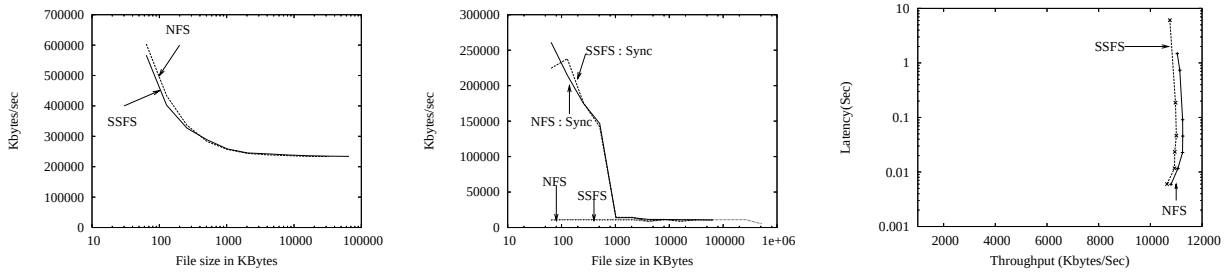


Figure 5.9: IOZONE : (a) Read (b) Write (c) Latency versus Throughput

the cost, performance, and availability of the system. Remote storage is used to store data durably to guard against local failures. The local server encrypts (using SHA1 and 1024 bit Rabin key signature) and encodes [110] (if data is not fully replicated) all data before sending it to remote SSPs, and it transparently fetches, decodes and decrypts data from remote storage if it is not present in the local cache. Our implementation thus supports policies that reduce local space demands by garbage collecting cold objects, but exploring such policies is future work; our prototype local server simply stores local copies of all objects.

All local server state except the encryption key and list of SSPs is soft state: given these items, the local server can recover the full filesystem. We assume both are stored out of band (e.g., the owner burns them to a CD at installation time and stores the CD in a safety deposit box). A more convenient (and thus more robust in terms of data durability) but lower-security alternative is to remember the list of SSPs and to encrypt the key with a password, erasure code it, and store the key fragments in well-known object IDs at the SSPs.

Snapshots: In addition to the standard NFS calls, the SSFS local server provides a snapshot interface [23] that supports file versioning for achieving temporal isolation to tolerate client or administrator failures. A snapshot stores a copy in the local cache and also redundantly stores encrypted, erasure-coded data across multiple SSPs using the remote storage interface.

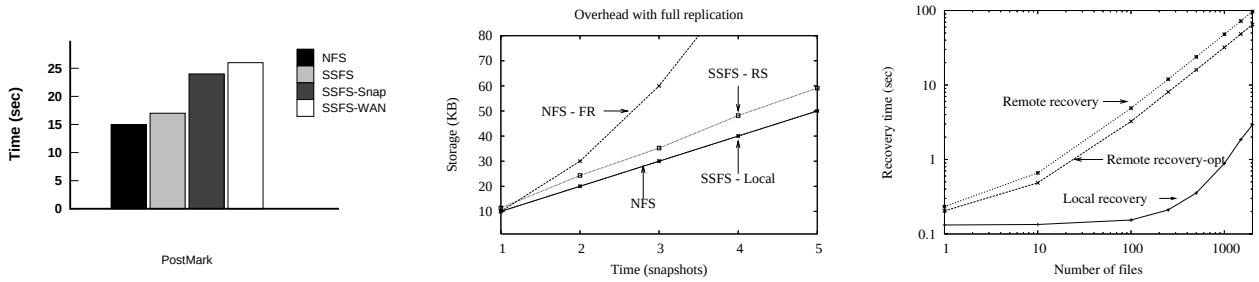
Local storage is structured carefully to reduce storage and performance overheads for maintaining multiple versions of files. SSFS uses block-level versioning [23, 108] to reduce storage overhead by storing only modified blocks in the older versions when a file is modified. For each old version, SSFS maintains a *block mask* and *size* in a meta-data file for the older version. Then, reads of the current version see no overhead, and reads of the older version are satisfied by starting with the old version and then fetching data blocks not present in the old version from later versions by sequentially checking the later versions until the block is found [108]. And as an obvious extension for the common case of files modified by appends: on an append, SSFS needs only to store the old size (and not the block mask) as all blocks are stored in later versions.

Other optimizations: SSFS uses a fast recovery optimization to recover quickly from remote storage when local data is lost due to local server failures (disk crashes, fire, etc.) The SSFS local server recovers quickly by coming online as soon as all metadata information (directories, inodes, and old-version information) is recovered and then fetching file data to fill the local cache in the background. If a missing block is requested before it is recovered, it is fetched immediately on demand from the SSPs. Additionally, local storage acts as a write-back cache where updates are propagated to remote SSPs asynchronously so that client performance is not affected by updates to remote storage.

5.6 Evaluation

To evaluate the practicality of the SafeStore architecture, we evaluate our SSFS prototype via microbenchmarks selected to stress test three aspects of the design. First, we examine performance overheads, then we look at storage space overheads, and finally we evaluate recovery performance.

In our base setup, client, local server, and remote SSP servers run on different machines that are connected by a 100 Mbit isolated network. For several experiments we modify the network to synthetically model WAN behavior. All of our machines use 933MHZ Intel Pentium III processors with 256 MB RAM and run



(a) (b) (c)

Figure 5.10: (a) Postmark: End-to-end performance (b) Storage overhead (c) Recovery

Linux version 2.4.7. We use (3,2) erasure coding or full replication ((3,1) encoding) to redundantly store backup data across SSPs.

5.6.1 Performance

Figure 5.9 compares the performance of SSFS and a standard NFS server using the IOZONE [17] microbenchmark. In this experiment, we measure the overhead of SSFS's bookkeeping to maintain version information, but we do not take filesystem snapshots and hence no data is sent to the remote SSPs. Figure 5.9(a),(b), and (c) illustrates throughput for reads, throughput for synchronous and asynchronous writes, and throughput versus latency for SSFS and stand alone NFS. In all cases, SSFS's throughput is within 12% of NFS.

Figure 5.10(a) examines the cost of snapshots. Note SSFS sends snapshots to SSPs asynchronously, but we have not lowered the priority of these background transfers, so snapshot transfers can interfere with demand requests. To evaluate this effect, we add snapshots to the Postmark [29] benchmark, which models email/e-commerce workloads. The benchmark initially creates a pool of files and then performs a specified number of transactions consisting of creating, deleting, reading, or appending a file. We set file sizes to be

between 100B and 100KB and run 50000 transactions. To maximize the stress on SSFS, we set the Postmark parameters to maximize the fraction of append and create operations. Then, we modify the benchmark to take frequent snapshots: we tell the server to create a new snapshot after every 500 transactions. As shown in the Figure 5.10(a), when no snapshots are taken SSFS takes 13% more time than NFS due to overhead involved in maintaining multiple versions. Turning on frequent snapshots increases the response time of SSFS (SSFS-snap in Figure 5.10(a)) by 40% due to additional overhead due to signing and transmitting updates to SSPs. Finally, we vary network latencies to SSPs to study the impact of WAN latencies on performance when SSPs are geographically distributed over the Internet by introducing artificial delay (of 40 ms) at the SSP server. As shown in the Figure 5.10(a), SSFS-WAN response time increases by less than an additional 5%.

5.6.2 Storage overhead

Here, we evaluate the effectiveness of SSFS's mechanisms for limiting replication overhead. SSFS minimizes storage overheads by using a versioning system that stores the difference between versions of a file rather than complete copies [108]. We compare the storage overhead of SSFS's versioning file system and compare it with NFS storage that just keeps a copy of the latest version and also a naive versioning NFS file system (NFS-FR) that makes a complete copy of the file before generating a new version. Figure 5.10(b) plots the storage consumed by local storage (SSFS-LS) and storage at one remote server (SSFS-RS) when we use a (3,1) encoding. To expose the overheads of the versioning system, the microbenchmark is simple: we append 10KB to a file after every file system snapshot. SSFS's local storage takes a negligible amount of additional space compared to non-versioned NFS storage. Remote storage pays a somewhat higher overhead due to duplicate data storage when appends do not fall on block boundaries and due to additional metadata (integrity hashes, the signed write request, expiry time of the file, etc.)

We also ran an experiment with the (3,2) encoding at remote servers using Postmark benchmark with varying snapshot frequencies and observed similar results. We omit these graphs for brevity. The above experiments examine the case when the old and new versions of data have much in common and test whether SSFS can exploit such situations with low overhead. There is, of course, no free lunch: if there is little in common between a user's current data and old data, the system must store both. Like SafeStore, Glacier uses a expire-then-garbage collect approach to avoid inadvertent file deletion, and their experience over several months of operation is that the space overheads are reasonable [74]. We plan to confirm these results in a SafeStore context by evaluating space overhead using the long-duration Harvard traces [68].

5.6.3 Recovery

We now evaluate SSFS recovery time and compare performance with and without SSFS's fast recovery optimization that allows the local server to resume operation as soon as it has recovered file system metadata and to recover the rest of the system's data in the background.

We also plot recovery time of SSFS from local storage due to reboots of the local server. Figure 5.10(c) plots recovery time as the number of 1KB files in the system varies when the data is recovered from remote SSPs. We see that local recovery is faster than the other two as it recovers from the local disk and it outperforms the other two by more than an order of magnitude for moderate number of files in the system. We also observe that remote recovery with optimization outperforms remote recovery without optimization by about 50% even with as few as 10 files. Note that recovery time is high even with the optimization as SSFS recovers all the metadata (which involves reading from remote SSPs, verifying the metadata integrity, decoding data from redundant fragments, and finally decrypting the metadata) before it starts serving the client requests. As part of our future work, we intend to reduce the recovery time significantly by bringing the system up immediately while the metadata is fetched in the background like the existing optimization for data.

5.7 Related work

Several recent studies [49, 114] have identified the challenges involved in building durable storage system for multi-year timescales.

Flat erasure coding across nodes [53, 62, 74, 132] does not require detailed predictions of which sets of nodes are likely to suffer correlated failures because it tolerates any combinations of failures up to a maximum number of nodes. However, flat encoding does not exploit the opportunity to reduce replication costs when the system can be structured to make some failure combinations more likely than others. An alternative approach is to use *full replication* across sites that are not expected to fail together [78, 95], but this can be expensive.

SafeStore is architected to increase the likelihood that failures will be restricted to specific groups of nodes, and it efficiently deploys storage within and across SSPs to address such failures. Myriad [59] also argues for a 2-level (cross-site, within-site) coding strategy, but SafeStore's architecture departs from Myriad in keeping SSPs at arms-length from data owners by carefully restricting the SSP interface and by including provisions for efficient end-to-end auditing of black-box SSPs.

SafeStore is most similar in spirit to OceanStore [77] in that we erasure code indelible, versioned data across independent SSPs. But in pursuit of a more aggressive "nomadic data" vision, OceanStore augments this approach with a sophisticated overlay-based infrastructure for replication of location-independent objects that may be accessed concurrently from various locations in the network [109]. We gain considerable simplicity by using a local soft-state server through which all user requests pass and by focusing on storing data on a relatively small set of specific, relatively conventional SSPs. We also gain assurance in the workings of our SSPs through our audit protocol.

Versioning file systems [23, 104, 113, 117, 126] provide temporal isolation to tolerate client failures by

keeping multiple versions of files. We make use of this technique but couple it with efficient, isolated, audited storage to address a broader threat model.

We argue that highly durable storage systems should audit data periodically to ensure data integrity and to limit worst-case MTTR. Zero-knowledge-based audit mechanisms [72, 97] are either network intensive or CPU intensive as their main purpose is to audit data without leaking any information about the data. SafeStore avoids the need for such expensive approaches by encrypting data before storing it. We are then able to offload audit duties to SSPs and probabilistically spot check their results. LOCKSS [95] and Samsara [64] audit data in P2P storage systems but assume that peers store full replicas so that they can easily verify if peers store identical data. SafeStore supports erasure coding to reduce costs, so our audit mechanism does not require SSPs to have fully replicated copies of data.

5.8 Conclusion

Achieving robust data storage on the scale of decades forces us to reexamine storage architectures: a broad range of threats that could be neglected over shorter timescales must now be considered. SafeStore aggressively applies the principle of *fault isolation* along administrative, physical, and temporal dimensions. Analysis indicates that SafeStore can provide highly robust storage and evaluation of an NFS prototype suggests that the approach is practical.

Chapter 6

Conclusion

The thesis of this dissertation is simple: Byzantine fault tolerance (BFT) techniques and technology trends are nearing an inflection point where significant deployment of BFT systems can be made viable. The mounting evidence of non-fail-stop behavior in real systems [37, 50, 51, 98, 101, 106, 123, 134, 135] suggest that BFT may yield significant benefits even without resorting to *n*-version programming [60, 81, 111]. The growing value of data [7, 12, 32, 114] and falling costs of hardware [8, 79] make it advantageous for service providers to trade increasingly inexpensive hardware for the peace of mind potentially provided by BFT replication. For example, the Google file systems (GFS) already uses three-way replication, by default, as a way to protect data from failures [71].

We recognize, however, that despite the advances of the last decade, Byzantine fault tolerance still carries in the mind of many practitioners, especially in the commercial world, a connotation of excessive cost, both in terms of performance losses and intellectual effort. It is, therefore, important to (1) develop techniques that minimize the costs of Byzantine fault tolerance and (2) provide compelling demonstrations of significant applications gaining robustness advantages from cost-effective, scalable BFT.

In this dissertation, as a step towards realizing this goal, we designed and implemented novel BFT replication techniques that significantly reduces performance overhead and complexity while keeping costs competitive withe existing practice. We made three contributions to this end. First, we designed and implemented CBASE, a high throughput BFT architecture to provide a general way to exploit application parallelism in order to provide high throughput. Second, we proposed Zyzzyva, a BFT state machine replication protocol

that reduces replication overheads in order to improve performance and reduce complexity. Third, we designed and implemented SafeStore, a highly durable distributed storage system that uses the principles of aggressive isolation and proactive audit to provide long-term data durability spanning many years or even decades by outsourcing storage to autonomous storage service providers.

Appendices

Appendix A

Concurrency Matrix for Network File System (NFS)

Here we explain the concurrency matrix used by the parallelizer to perform dependence analysis for replicated NFS (CBASEFS) as explained in chapter 3. The concurrency matrix for NFS (version 2.0) [25] is defined in the table A.1. The concurrency matrix (NFS-conc-matrix[18][18][2]) is defined for all 18 NFS operations as listed in the table. As explained in section 3.5.3 of chapter 3, NFS-conc-matrix[18][18][0] is the argument-independent concurrency matrix(OCM) and NFS-conc-matrix[18][18][1] is the argument-dependent concurrency matrix(OACM).

NFS OP	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Null (0)	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
Get attr (1)	1,1	1,1	1,0	1,1	1,1	1,1	1,1	1,1	1,1	1,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	1,1
Set attr (1)	1,1	1,0	1,0	1,1	0,0	1,0	1,0	1,1	1,0	1,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,1
Root (3)	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
Lookup file (4)	1,1	1,1	0,0	1,1	1,1	0,0	0,0	1,1	0,0	1,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,1
Read symlink (5)	1,1	1,1	1,0	1,0	1,1	1,0	1,1	1,1	1,1	1,1	0,0	0,0	0,0	0,0	0,0	0,0	1,1	1,1
Read file (6)	1,1	1,1	1,0	1,1	0,0	1,1	1,0	1,1	1,0	1,1	0,0	0,0	0,0	0,0	0,0	0,0	1,1	1,1
Cache (7)	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
Write (8)	1,1	1,0	1,0	1,1	0,0	1,0	1,0	1,1	1,0	1,0	0,0	0,0	0,0	0,0	0,0	0,0	1,1	0,0
Create file (9)	1,1	1,0	1,0	1,1	1,0	1,1	1,0	1,1	1,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,1	0,0
Remove file (10)	1,1	0,0	0,0	1,1	0,0	0,0	0,0	1,1	0,0	0,0	1,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
Rename file (11)	1,1	0,0	0,0	1,1	0,0	0,0	0,0	1,1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
Create (12)	1,1	0,0	0,0	1,1	0,0	0,0	0,0	1,1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
Create Symlink (13)	1,1	0,0	0,0	1,1	0,0	0,0	0,0	1,1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
Create Dir (14)	1,1	0,0	0,0	1,1	0,0	0,0	0,0	1,1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
Remove Dir (15)	1,1	0,0	0,0	1,1	0,0	0,0	0,0	1,1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	0,0	0,0
Read dir (16)	1,1	1,0	1,0	1,1	0,0	1,1	1,1	1,1	1,1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	1,1
Filesys attr (17)	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,1	1,1

Table A.1: **NFS concurrency matrix: NFS-con-matrix[18][18][2]**

Appendix B

Durability analysis

In this section, we describe the analytical models for estimating durability and storage overhead of different encoding schemes such as hierarchical data encoding, flat erasure coding, and full replication that we compare in section 5.3.

B.1 Durability

Here we describe the analytical model for estimating durability of data stored using hierarchical data encoding, flat erasure coding, and full replication. Consider a system with n nodes spread across k groups (SSPs) with n_1, n_2,..n_k nodes respectively present in groups 1,2,...k. All nodes in a group n_i fail in a correlated fashion with probability p_c due to a correlated failure events (fire, administrator failure) at an SSP. Also, a node fails independently with a failure probability p_u due to uncorrelated failure events (disk failures) at an SSP. In order to analyze durability over some time duration, we first evaluate the durability of data in an epoch and then aggregate it over multiple epochs spanning the duration as in [129]. Given this failure model, durability of hierarchical encoding, flat erasure coding, and full replication is described below

Hierarchical data encoding: The durability of data in an epoch when data is hierarchically encoded with (k,l) erasure coding across SSPs and with (n_i, m_i) encoding within an SSP group i is given by:

$$\text{Durability of data in an epoch, } D_e^{Hier} = Pr(\sum_{i=1}^{k} X_i \geq l)$$

where

$$\text{Random variable} : X_i$$

$$X_i = 0, \text{if} < m_i \text{ nodes are up in SSP i}$$

$$= 1, \text{if} \geq m_i \text{ nodes are up in SSP i}$$

$$Pr(X_i = 1) = (1 - p_c) \times \sum_{j=m_i}^{n_i} \binom{n_i}{j} (1 - p_u)^j p_u^{n_i - j},$$

Flat erasure encoding: The durability of data in an epoch when data is encoded using (n,m) flat erasure coding using all nodes across all SSPs is given by:

$$\text{Durability of data, } D_e^{Flat} = Pr((\sum_{i=1}^{k} Y_i) \geq m)$$

where Y_i is a random variable representing the number of nodes that are up in group i and n is the total number of nodes spread across all SSPs and $Pr(Y_i = l_i) = (1 - p_c) \times \binom{n_i}{l_i} p_i^j (1 - p_i)^{n_i - l_i}$ is the probability is the probability that l_i nodes out of n_i nodes in group i.

Full replication: The durability of data in an epoch when the data is fully replicated with k replicas (with one replica per SSP) is given by

$$\text{Durability of data, } D_e^{Full} = Pr(\sum_{i=1}^{k} Y_i \geq 1)$$

Overall durability: Overall durability of data (using any replication mechanism) for a time duration T is given by

$$\text{Durability of data in time duration T, } D$$

$$= \text{1-prob(data loss in time t} \leq \text{T)}$$

$$= D_e^{T/e}$$

where e is the epoch length and D_e is the durability of a given replication mechanism in an epoch.

We set epoch length to MTTR min($MTTR_u$,$MTTR_c$), where $MTTR_u$ and $MTTR_c$ are mean time to recoveries from uncorrelated node failure with in SSP and correlated SSP failures. We assume that failures in an epoch are not repaired before the end of epoch in computing D_e. The failures are assumed to be repaired before the start of next epoch (as we assume each epoch instance as a fresh Bernoulli trial while computing overall durability D as shown above). Given this epoch length, we can compute p_u and p_c from MTTDL [49] due to uncorrelated *node failure* ($MTTDL_u$) and correlated *SSP failure* ($MTTDL_c$) events as

$$p_u = MTTR_u/MTTDL_u$$

$$p_c = MTTR_c/MTTDL_c$$

Mean time to data loss (MTTDL): Durability of data in terms of MTTDL is given by

$$\text{MTTDL} = \sum_{i=0}^{\infty} iD_e{}^i(1-D_e) = D_e/(1-D_e)$$

where D_e is the durability of a given replication mechanism in an epoch.

B.1.1 Hierarchical encoding observation:

Remark B.1.1. In hierarchical encoding, the overall durability does not improve much by additional intra-SSP redundancies beyond a certain minimum value if $p_u << 1 - p_u$.

Proof:

$$Pr(X_i = 1) = (1 - p_c) \times \sum_{j=m_i}^{n_i} \binom{n_i}{j} (1 - p_u)^j p_u^{n_i - j},$$

$$= (1 - p_c) \times (1 - p_u)^{n_i} \times (1 + \sum_{j=1}^{n_i - m_i} \binom{n_i}{j} (p_u/(1 - p_u))^j),$$

$$\approx (1 - p_c) \times (1 - p_u)^{n_i} \times (1 + \sum_{j=1}^{\alpha} \binom{n_i}{j} (p_u/(1 - p_u))^j),$$

$$= (1 - p_c) \times \sum_{j=\alpha}^{n_i} \binom{n_i}{j} (1 - p_u)^j p_u^{n_i - j},$$

In most practical settings, p_u is far less than $1 - p_u$ where $MTTDL_u$ is in the order of tens of years and $MTTR_u$ is in the order of days. For example, $p_u/(1 - p_u) < 0.0002$ for MTTDL due to node failure is 5 years and MTTR of 1 day. It is explained as follows. And α depends on n_i and $p_u/(1 - p_u)$, such that $(\alpha + 1) >> (n_i - 1) \times p_u/(1 - p_u))$. For example, $\alpha \approx n_i - 1$, when n_i is in the order of tens of nodes, $MTTDL_u$ is in the order of years and $MTTR_u$ is in the order of days. This implies that overall durability saturates fairly quickly with increasing intra-SSP redundancy when inter-SSP redundancy is fixed.

B.2 Overhead

Storage overheads of different encoding schemes are described below

Hierarchical encoding: The storage overhead of hierarchical encoding scheme that uses (k,l) encoding across SSPs (inter-ssp encoding) and (n_i,k_i) encoding with in an SSP group i (intra-ssp encoding) is given by:

$$\text{Overhead} \quad = 1/l \times (n_0/m_0 + n_1/m_1 + ... + n_{k-1}/l_{k-1})$$

$$= 1/l \times (r_0 + r_1 + ... + r_{k-1})$$

Flat erasure coding: The storage overhead of scheme using (n, m) flat erasure coding is n/m.

Full replication: The storage overhead of scheme using full replication with l replicas spread across l SSP groups is l.

Appendix C

Audit protocol

1. Store data	$O \rightarrow SSP : \{objId, H(data_{objId}), t_{exp}\}_O, data_{objId}$
2. Receive receipt	$SSP \rightarrow O : \{objId, H(data_{objId}), t_{exp}\}_{SSP}$
3. Store receipt	$O \rightarrow S : SSP_{id}, \{objId, H(data_{objId}), t_{exp}\}_{SSP}$

Table C.1: **Data storage sub-protocol:** In the first phase, the data owner *O* sends a storage request to store a data object $data_{objId}$ with object id obj_id for a time duration of t_{exp} to the storage service provider *SSP*. The data owner then gathers the signed and verifiable promisary receipt from *SSP* in the second phase. It then stores the receipt from *SSP* redundantly at all storage service providers defined by set *S* in the third phase.

1. Challenge	$A \rightarrow SSP : chal, listOfObjects$
2. Response	$SSP \rightarrow A : \{objId, chal, time, H(chal + data_{objId})\}_{SSP} \mid \{objId, chal, time, FAILURE\}_{SSP}$

Table C.2: **Routine audit sub-protocol:** Auditor periodically sends a challenge to the SSP(auditee). The challenge includes a nonce *chal* and a list of objects being audited ($listofObjects$). For every data object $data_{objId}$ in the list, SSP computes the hash value $H(chal + data_{objId})$. SSP sends a signed response back to the auditor for every object. The response includes object id $objId$, current time $time$, and the hash value $H(chal + data_{objId})$. SSP can optionally send *FAILURE* message if it finds data object to be lost or corrupted.

1. Request data	$A \rightarrow O \vert SSP \vert SSP' : list2OfObjects$
2. Send data	$O \vert SSP \vert SSP' \rightarrow A : data_{objId}$

Table C.3: **Spot check sub-protocol:** Auditor spot checks the responses of routine audit protocol by reading data for a subset of objects. Auditor gathers data by reading data from the SSP being audited or other SSPs at which the data is redundantly stored or the data owner *O*.

POM =	*receipt* and *auditReply* are well-formed and signed by *SSP*
	$\wedge (objId = receipt_{objId} = auditReply_{objId})$
	$\wedge (receipt_{expires} > auditReply_{time})$
	$\wedge receipt_{H(objId)} = auditReply_{H(objId)}$
	$\wedge chal = auditReply_{chal}$
	$\wedge H(chal + data) \neq auditReply_{H(chal+data)}$

Table C.4: **Proof of mis-behavior (POM):** Auditor can generate a verifiable proof of mis-behavior, as described in this table, against an SSP if an SSP lies during the routine audit protocol by sending a fake hash value. It does so by gathering data for some random subset of objects.

Appendix D

Audit analysis with selfish SSPs

Here we show that a rational [46] SSP (that can selfishly deviate from the protocol for its own benefit) follows the protocol in the presence of a SLA that specifies appropriate penalties relative to the underlying cost of storing data.

b_{serve}	Benefit for storing and serving object until it expires
c_{store}	Cost to store and serve object until it expires (including cost of serving audit requests)
p_{audit}	Probability that object will be audited before it expires
p_{spot}	Probability that an audit reply will be spot-checked
$penalty_h$	Penalty for *honest failure* of audit (see Section 5.4
$penalty_d$	Penalty for *dishonest failure* of audit

Table D.1: **Definitions**

Theorem 3. *SafeStore audit protocol ensures that the rational SSPs (SSPs can selfishly deviate from the protocol to maximize their own benefits) follow the protocol by (1) attempting to store data reliably and (2) responds to audit requests honestly assuming an SLA that specifies appropriate penalties relative to the underlying cost of storing data. cost model.*

Proof:

- $b_{serve} > c_{store} \wedge c_{store} < min(p_{audit}\, penalty_h, p_{audit}\, p_{spot}\, penalty_d) \implies$ A rational SSP attempts to store an object until it expires.

- $penalty_h < p_{spot}\, penalty_d \implies$ A rational SSP that does not have the data needed to reply to an audit request replies with an *honest failure* rather than with a audit reply that could be used to generate a proof of misbehavior.

Example. These requirements are met by a system with $c < \$1$ (reasonable if all objects are broken into 1GB or smaller pieces and stored with expiration times of less than a year), $b = 2c$, $p_{audit} = 90\%$, $p_{spot} = 1\%$, $penalty_h = \$5$, and $penalty_d = \$1000$.

Appendix E

Additional experiments

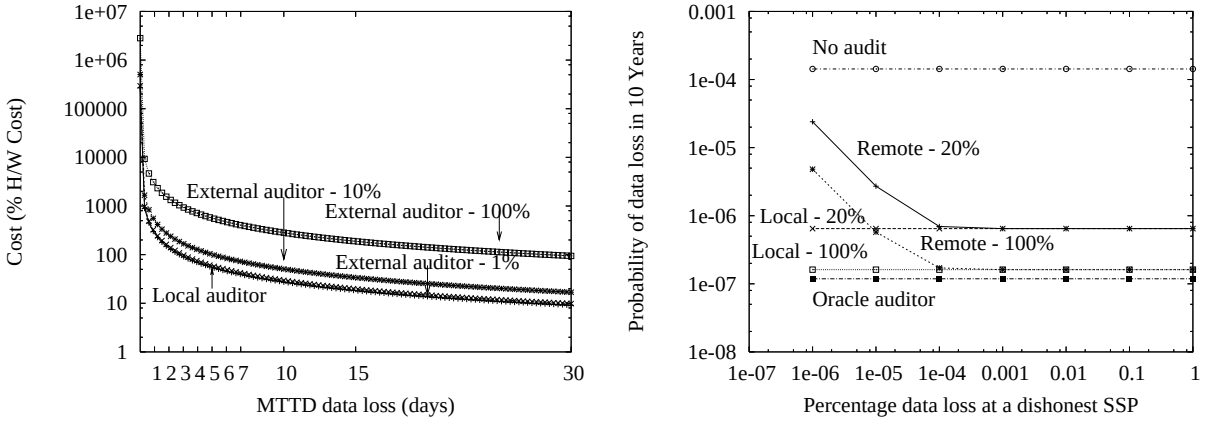


Figure E.1: Audit (a) Time to detect SSP data loss via audit with varying amounts of resources dedicated to audit overhead assuming honest SSPs with (3,2) inter-SSP redundancy. (b) Impact on overall durability with a dishonest SSP with varying audit costs (20% and 100%)

E.1 Audit

Here, we run additional experiments to evaluate our audit protocol as described in section 5.4. Figure E.1(a) plots mean time to detect data loss (MTTD) at a *passive* SSP when (3,2) encoding is used to store 1TB of data redundantly across 3 SSPs. MTTD falls rapidly with increasing audit budget similar to the system that uses (3,1) as shown in the Figure 5.8(a) of section 5.4. However, for a fixed MTTD, (3,2) encoding incurs higher audit cost per byte stored compared to (3,1) because of increased overhead due to reduced block size from 4KB (with (3,1)) to 2KB (with (3,2)). Figure E.1 illustrates the overall impact on durability in the

presence of *dishonest* SSPs with varying audit budgets. With 20% audit budget, we outperform a system with no audit by two 9's in the presence of *dishonest* SSPs. As we increase our system's audit budget from 20% to 100%, durability of our system approaches that of a system with *oracle detector*.

Appendix F

Protocol Comparisons

Table 4.1 presents a quick numerical comparison of the intrinsic overheads and requirements of 4 BFT systems. Here we present a detailed explanation of the table.

Required Replicas The first row in the table accounts for the number of replicas required to tolerate f failures in each of the four systems. PBFT, HQ, and Zyzzyva require $3f+1$ replicas to tolerate f faults while Q/U and Zyzzyva5 require $5f+1$ replicas. This total number of replicas reflects the number of replicas required to coordinate the protocol state in the systems. PBFT, Zyzzyva and Zyzzyva5 require only $2f+1$ of the total replicas to store application state. Nominally replicas that do not store application state are less expensive than full replicas that maintain the application state.

Throughput overhead The computational throughput of a distributed system is dominated by the number of operations at the most heavily loaded replica. The cryptographic operation of generating and verifying MACs at the replicas are intrinsic and present computational overhead that cannot be avoided.

In Q/U and HQ there is no distinguished replica that plays a special role; within each of these systems all replicas perform the same number of MAC operations. In Q/U and HQ each replica performs $2+8f$ and $4+4f$ MAC operations per client request respectively. In Q/U the total number of MAC operations at the bottlenecked server can be broken down as follows: one MAC operation for verifying the client request, 4f MAC operations to verify OHS (object history set), one MAC operation to generate MAC for the reply, 4f MAC operations to generate the authenticator for its history set. In HQ the bottlenecked server performs

following MAC operations per client request: one operation to verify the client request, 2f operations to generate authenticator for the grant timestamp phase (Write-1 phase), 1 operation to authenticate the Write-1 phase response to the client, 1 operation to verify the Write-2 request from the client, 2f operations to verify the write certificate in the Write-2 phase, 1 operation to authenticate the response to the client.

For PBFT we assume that the implementation uses preferred quorum optimization [63] to improve performance. The number of cryptographic operations performed by the bottlnecked server (backup replica) is $2+(8f+1)/b$ per client operation when b client operations are batched in a single execution. Here a backup replica requires 2 MAC operations per client request with one MAC operation each to verify the client request and authenticate the server response. In addition, it requires $8f+1$ operations [1] for agreeing on the order for b client operations (one operation to verify the pre-prepare message, 2f operations to generate a prepare message for other replicas, 2f operations to verify 2f prepare messages from other replicas, 2f operations to generate a commit message, 2f operations to verify 2f other commit messages). For Zyzzyva the system is bottlenecked by the primary with $2+3f/b$ MAC operations per client operation when b clients are batched. The primary in Zyzzyva requires 2 MAC operations per client request to verify the client request and to authenticate the client response. In addition, the primary requires 3f operations to generate authenticator for Order request message for a batch of b client requests. The backups require fewer operations than the primary with the preferred quorum optimization.

Network Latencies The overall latency of a request is bounded from below by the number of one way network latencies required by the system. PBFT and HQ each require 4 one way latencies Zyzzyva and Zyzzyva5 require 3 latencies. In PBFT the complete request flight includes network latencies from client to primary, primary to replicas, replicas to replicas, replicas to client. In HQ the latencies are from client to

[1] Without assuming piggybacking commit messages in other messages as in PBFT library implementation [57]. However, with this optimization the number of operations reduces to $4f+1$ operations at the bottlenecked server.

replicas, replicas to client, client to replicas, and replicas to client. Zyzzyva and Zyzzyva5 both follow the pattern of client to primary, primary to replicas, replicas to client to complete an operation. PBFT, Zyzzyva and Zyzzyva5 are all agreement based protocols and the theoretical lower bound for message delays in agreement based protocols is 3. Q/U requires only two message delays, client to replicas and replicas to client, when there is no contention in the system. In the presence of contention Q/U requires an unbounded number of messages delays. Q/U requires fewer message delays than the theoretical minimum of 3 by solving a different, slightly weaker, problem.

We expand on the Table 4.1 to compare message and cryptographic overheads of various protocols at the clients and replicas as shown in the following Tables F.1 and F.2.

	PBFT			HQ		Q/U		Zyzzyva		
	client	primary	replica	client	replica	client	replica	client	primary	replica
Sent Msg	b	6f+b	6f+b	b(6f+2)	2b	b(5f+1)	b	b	3f+b	b
Rcvd Msg	b(3f+1)	6f+b	6f+b+1	b(6f+2)	2b	b(5f+1)	b	b(3f+1)	b	1
Tot Msg	3fb+b	12f+2b	12f+2b+1	b(12f+4)	4b	b(10f+2)	2b	3fb+2b	3f+2b	b+1
Crypt Gen	b(3f+1)	6f+b	6f+b	b(6f+2)	b(3f+2)	b(5f+1)	b(5f+1)	b(3f+1)	3f+b	3f+b
Crypt Ver	b(2f+1)	4f+b	4f+b+1	b(4f+2)	b(2f+3)	b(4f+1)	b(4f+1)	b(3f+1)	b	b+1
Crypt Tot	5fb+2b	10f+2b	10f+2b+1	b(10f+4)	b(5f+2)	b(9f+2)	b(9f+2)	b(6f+2)	3f+2b	3f+2b+1
Sent Msg	b	4f+b	4f+b	b(4f+2)	2b	b(4f+1)	b	b	3f+b	b
Rcvd Msg	b(2f+1)	4f+b	4f+1	b(4f+2)	2b	b(4f+1)	b	b(3f+1)	b	1
Tot Msg	b(2f+2)	8f+2b	8f+b+1	2b(4f+2)	4b	2b(4f+1)	2b	3bf+2b	3f+2b	b+1
Crypt Gen	b(2f+1)	4f+b	4f+b	b(4f+2)	b(2f+2)	b(4f+1)	b(4f+1)	b(3f+1)	3f+b	b
Crypt ver	b(2f+1)	4f+b	4f+b+1	b(4f+2)	b(2f+2)	b(4f+1)	b(4f+1)	b(3f+1)	b	b+1
Crypt Tot	2b(2f+1)	8f+2b	2(4f+b)+1	2b(2f+2)	2b(2f+2)	2b(4f+1)	2b(4f+1)	b(6f+2)	3f+2b	2b+1

Table F.1: Overhead comparison of various protocols at clients and servers. The protocols under comparison tolerate f failures. Message overhead is measured as the number of messages sent or received. Here we have a setup with b clients with 1 request/client and the protocols use a batch size of b. The above table includes the total overhead for b clients in the clients column and per client overhead can be calculated by dividing it by b. The first two sub-tables (message and cryptographic overheads) list the overhead without the preferred quorum optimization and the last two sub-tables assume preferred quorum optimization. The overheads for Zyzzyva5 is listed in the following table.

	Zyzzyva-5		
	client	primary	replica
Sent Msg	b	5f+b	b
Rcvd Msg	b(5f+1)	b	1
Tot Msg	5fb+2b	5f+2b	b+1
Crypt Gen	b(5f+1)	5f+b	5f+b
Crypt Ver	b(4f+1)	b	b+1
Crypt Tot	b(9f+2)	5f+2b	5f+2b+1
Sent Msg	4f+b	4f	b
Rcvd Msg	b(4f+1)	b	1
Tot Msg	b(4f+2)	4f+b	b+1
Crypt Gen	b(4f+1)	4f+b	b
Crypt ver	b(4f+1)	b	b+1
Crypt Tot	2b(4f+1)	4f+2b	2b+1

Table F.2: Here is the overheads column for Zyzzyva5 (continued from previous table).

Appendix G

PKI Protocol Description

The goal of the system is to ensure that if the results of operations o and o' are accepted by a non-faulty client as valid, then the linearized order of requests are consistent. Unlike previous systems, we involve the client in this determination and do not rely solely on the replicas to ensure the property. Consequently, we define the following client centric predicate:

We formalize the statement that a client accepts a request with the predicate client–delivered(o,n,h,c), which is true if non-faulty client c accepts the result of o as the n operation in the sequence of requests whose total order through n is defined by history h.

The system is designed to maintain the following safety property:

SAFETY: client–delivered(o,n,h,c), client–delivered(o',n',h',c') and $n' \geq n$ implies that h is a prefix of h' and $o = o'$ if $n = n'$.

G.1 Agreement Protocol

Figure 4.1 illustrates the basic flow of the agreement sub-protocol during a view. Since replicas execute requests speculatively in the order proposed by the primary without communicating with each other replicas, the key challenge is ensuring that clients only act upon replies that correspond to stable requests that were, in fact, executed in a total order that is guaranteed to eventually *commit* at all correct servers. The protocol is constructed so that a client knows that a request will eventually be *committed* when it receives $3f+1$

Label	Meaning
c	Client ID
CC	Commit certificate
d	Digest of client request message $d = H(m)$
i, j	Server IDs
h_n	History through sequence number n $h_n = H(h_{n-1}, d)$
m	Message containing client request
max_n	Max sequence number accepted by replica
n	Sequence number
o	Operation requested by client
OR	Order Request message
POM	Proof Of Misbehavior
r	Application reply to a client operation
t	Timestamp assigned to an operation by a client
v	View number

Table G.1: Labels given to fields in messages.

matching responses or acknowledgements from $2f+1$ replicas that they have received a *commit certificate* comprising a *local commit* from $2f+1$ replicas.

To describe how the system deals with this and other challenging, but standard, issues—lost messages, faulty primary, faulty clients, etc.—we follow a request through the system, defining the rules a server uses to process each message. The numbers in Figure 4.1 correspond to numbers in the text identifying major steps in the protocol and Table G.1 summarizes the labels we give fields in messages. Most readers will be happier if on their first reading they skip the text marked Additional Pedantic Details.

0. Replicas begin the protocol with a predefined base state.

Replica i begins the protocol in view 0 with an empty history and assigns the first request a sequence number of 1.

1. Client sends request to the primary.

A client c requests an operation o be performed by the replicated service by sending a $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$

message to the replica it believes to be the primary (i.e., the primary for the last response the client received.)

Additional Pedantic Details: If the client guesses the wrong primary, the retransmission mechanisms discussed in step 4c below forwards the request to the current primary. The client's timestamp t is included to ensure exactly-once semantics of execution of requests so that no request is executed more than once by the replicated service.

2. Primary receives request, assigns sequence number, and forwards ordered request to replicas.

When the primary p receives a new request $m = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ from client c, the primary assigns a sequence number n in view v to the request and relays a $\langle \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_p}, m \rangle$ message to the backup replicas where v indicates the view in which the message is being sent, n is the proposed sequence number for m, $d = H(m)$ is the digest of m, $h_n = H(h_{n-1}, d)$ is a digest summarizing the history, and ND is a set of values for non-deterministic application variables (time in file systems, locks in databases, etc.) required for execution.

Additional Pedantic Details: The primary only takes the above actions if $t > t_c$ where t_c is the highest timestamp previously received from c and m can be verified. If m cannot be verified then the primary drops the request and does nothing.

3. Replica receives ordered request, speculatively executes it, and responds to the client.

Upon receipt of an order request message $\langle \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_p}, m \rangle$ from the primary p, replica i discards the message if $n \leq max_n$ where max_n is the largest sequence number in its history. If $n = max_n + 1$, m is a well-formed request message, d is a correct digest of m, and $h_n = H(h_{n-1}, d)$, then i accepts the order request message. Upon accepting the message, i appends the ordered request to its history, executes the request using the current application state to produce a reply r, and sends to c a speculative response

message $\langle\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$ where $OR = \langle \text{ORDER-REQ}, v, n, h_n, d, ND\rangle_{\sigma_p}$.

Additional Pedantic Details: A replica may only accept and speculatively execute requests in sequence-number order, but message loss or a faulty primary can introduce holes in the sequence number space. If $n > max_n + 1$, then i discards [1] the order request message and initiates the fill hole protocol described in the section G.1.2. If the order request is inconsistent with the history of order requests replica i has received, then the two order requests showing the inconsistency consist of a POM and the replica (a) forwards the POM to all other replicas and (b) initiates a view change.

4. Client gathers speculative responses.

Next, the client receives speculative responses $\langle\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$ from the replicas. Speculative responses from distinct replicas *i match* if they have identical *v*, *n*, h_n, $H(r)$, *c*, *t*, and *r* fields. There are four cases to consider. The first three handle varying numbers of matching speculative replies without considering *OR* while the last considers only the *OR* portion of the message.

4a. Client receives $3f + 1$ matching responses and completes the request.

In the absence of faults, the client receives matching speculative response messages from all $3f + 1$ replicas. The client then considers the request and its history to be *complete* and delivers the reply *r* to the application. Zyzzyva guarantees that even if there is a view change, all correct replicas will always execute this request at this point in their history to produce this response. Notice that although the client has a proof that the request's place in history is irrevocably set, no server has such a proof. Indeed, servers at this point cannot determine if a request has completed in its final order or if it will have to roll back its state because a faulty primary ordered the request inconsistently across replicas.

[1] We cache out-of-order requests as an optimization as explained in the section 4.4 but omit this optimization here for simplicity.

4b. Client receives between $2f+1$ and $3f$ matching responses, assembles a commit certificate, and transmits the commit certificate to the replicas.

If the network, primary, or some replicas are faulty, the client c may never receive responses from all $3f+1$ replicas. The client therefore sets a timer when it first issues a request, and when this timer expires, if c has received matching speculative responses from between $2f+1$ and $3f$ replicas, then c sends a commit message $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$ where CC is a commit certificate consisting of a list of $2f+1$ replicas, the replica-signed portions of the $2f+1$ matching speculative responses from those replicas, and the corresponding $2f+1$ replica signatures.

Additional Pedantic Details: CC contains $2f+1$ signatures on the speculative reply message and a list of $2f+1$ nodes, but, since all the responses received by c from replicas are identical, c only needs to include *one* replica-signed portion of the speculative response. Also note that, for efficiency, CC does not include the body r of the reply but only the hash $H(r)$.

4b.1. Replica receives a commit message from a client containing a commit certificate and acknowledges with a local-commit message.

When a replica i receives a commit message $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$ containing a valid commit certificate CC proving that a request should be executed with a specified sequence number and history in the current view, the replica first ensures that its local history is consistent with the one certified by CC. If the history certified by CC matches replica i's local history, then i updates its *max commit certificate* state if the sequence number is higher than the stored certificate's sequence number and sends a local commit message $\langle \text{LOCAL-COMMIT}, v, d, h, i, c \rangle_{\sigma_i}$ to c. If the history certified by CC does not match i's local history, then i initiates a view change.

Additional Pedantic Details: If i's history is inconsistent with the history certified by CC then i constructs a POM from CC and an appropriate order request. Replica i then initiates a view change and forwards the

POM to all other replicas.

4b.2. Client receives a local commit messages from $2f+1$ replicas and completes the request.

The client resends the commit message until it receives corresponding local commit messages from $2f+1$ distinct replicas. The client then considers the request and its history to be *complete* and delivers the reply r to the application. The system guarantees that even if there is a view change, all correct replicas will always execute this request at this point in their history to produce this response.

4c. Client receives fewer than $2f+1$ matching responses and resends its request to all replicas, which forward the request to the primary in order to ensure the request is assigned a sequence number and eventually executed.

Client. If the network or primary is faulty, the client c may never receive matching responses from $f+1$ replicas. The client therefore sets a second timer when it first issues a request, and when this timer expires, resends the request to all replicas. It then resets its timers and continues gathering speculative responses.

Replica. When non-primary replica i receives a request $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ from client c there are two possible actions for i to take. If the time stamp in the request matches the time stamp for the the currently cached request for client c, then i resends the cached response to c. If instead the request has a higher timestamp than the currently cached response, then i sends a confirm request message $\langle \text{CONFIRM-REQ}, v, m, i \rangle_{\sigma_i}$ where $m = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ to the primary p and starts a timer. If the replica accepts an order request message for this request before the timeout, it processes the order request message as described above. If the timer fires before the order request message is received from the primary, the replica initiates a view change.

Primary. Upon receiving $\langle \text{CONFIRM-REQ}, v, m, i \rangle_{\sigma_i}$ from replica i, the primary p checks the client's timestamp for the request. If the request is new, p sends a new order request message using the next sequence number to order as described in step 2; otherwise, p sends to i the cached order request message for the most

recent request from *c*.

Additional Pedantic Details: If replica *i* has received a commit certificate or a stable checkpoint for a subsequent request or the request was ordered in a previous view, then the replica sends both a speculative response and a local-commit response to the client even if the client has not received a commit certificate for the retransmitted request. Additionally, if replica *i* does not receive the order request from the primary, the replica sends the confirm request message to all other replicas. Upon receipt of a confirm request message from another replica *j*, replica *i* sends the order request message it received from the primary to *j*; if *i* did not receive the request from the client, *i* acts as if the request came from the client itself.

4d. Client receives replies indicating inconsistent ordering by the primary and sends a proof of misbehavior to the replicas, which initiate a view change to oust the faulty primary.

If client *c* receives a pair of speculative reply messages containing valid order-request messages $OR = \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_j}$ for the same request ($d = H(m)$) in the same view *v* with differing sequence number *n* or history h_n, then the pair of order-request messages constitutes a proof of misbehavior (*POM*) against the primary. Upon receipt of a *POM*, *c* sends a primary faulty message $\langle \text{POM}, v, POM \rangle_{\sigma_c}$ to all replicas. Upon receipt of a valid *POM*, a replica initiates a view change.

Note that cases 4b through 4d are not mutually exclusive.

G.1.1 View Change

The Zyzzyva view change sub-protocol is similar to traditional view change sub-protocols with two key exceptions. First, while replicas in traditional view change protocols commit to the view change as soon as they suspect the primary to be faulty, replicas in Zyzzyva only commit to a view change when they know that all other correct replicas will join them in electing a new primary. Second, Zyzzyva weakens the condition under which a request appears in the new view's history. The protocol proceeds as follows.

VC1. Replica initiates the view change by sending an accusation against the primary to all replicas.

Replica i initiates a view change by sending $\langle$I-HATE-THE-PRIMARY, $v\rangle_{\sigma_i}$ to all replicas, indicating that the replica is dissatisfied with the behavior of the current primary. In previous protocols, this message would indicate that replica i is no longer participating in the current view. In Zyzzyva, this message is only a hint that i would like to change views. Even after issuing the message, i continues to faithfully participate in the current view.

VC2. Replica receives $f+1$ accusations that the primary is faulty and commits to the view change.

Replica i commits to a view change into view $v+1$ by sending an indictment of the current primary, consisting of $\langle$I-HATE-THE-PRIMARY, $v\rangle_{\sigma_j}$ from $f+1$ distinct replicas j, and the message $\langle$VIEW-CHANGE$, v+1, CC, O, i\rangle_{\sigma_i}$ to all replicas. CC is either the most recent commit certificate for a request since the last view change, $f+1$ VIEW-CONFIRM messages if no commit certificate is available, or a NEW-VIEW message if neither of the previous options are available. O is i's ordered request history since the commit certificate indicated by CC. At this point, a replica stops accepting messages relevant to the current view and does not respond to the client until a new view has started.

VC3. Replica receives $2f+1$ view change messages.

Primary. Upon receipt of $2f+1$ VIEW-CHANGE messages, the new primary p constructs the message $\langle$NEW-VIEW$, v+1, P\rangle_{\sigma_p}$ where P is a collection of $2f+1$ VIEW-CHANGE messages defining the initial state for view $v+1$.

Replica. The replica starts a timer. If the replica does not receive a valid NEW-VIEW message from the new primary before the timer expires, then the replica initiates a view change into view $v+2$.

Additional Pedantic Details: If a replica commits to change to view $v+2$ before receiving a new view

message for view $v+1$, then the replica uses the set of ordered requests from view v to form its view change message. The length of the timer in the new view grows exponentially with the number view changes that fail in succession.

VC4. Replica receives a valid new view message and sends a view confirmation message to all other replicas.

Replicas determine the state of the new view based on the collection of $2f+1$ VIEW-CHANGE messages included in the NEW-VIEW message. The most recent request with a corresponding commit certificate (or old new view message) is accepted as the last request in the base history. The most recent request that is ordered subsequent to the commit certificate by at least $f+1$ VIEW-CHANGE messages is accepted. Replica i forms the message $\langle \text{VIEW-CONFIRM}, v+1, n, h, i \rangle_{\sigma_i}$ based on the NEW-VIEW message and sends the VIEW-CONFIRM message to all other replicas.

Additional Pedantic Details: When evaluating the NEW-VIEW message, a commit certificate from the most recent view takes priority over anything else, followed by $f+1$ VIEW-CONFIRM messages, and finally a NEW-VIEW message with the highest view number.

VC5. Replica receives $2f+1$ matching VIEW-CONFIRM messages and begins accepting requests in the new view.

Upon receipt of $2f+1$ matching VIEW-CONFIRM messages, replica i begins the new view v.

Additional Pedantic Details: The exchange of view confirm messages is not strictly necessary for safety and can be optimized out of the protocol, but including them simplifies our safety proof by ensuring that if a correct replica begins accepting messages in new view v, then no other correct replica will accept messages in view v with a different base history. This step allows replicas to consider a confirmed view change to be functionally equivalent to a commit certificate for all requests in the base history of the new view.

G.1.2 State Transfer and Garbage Collection

Checkpoint Protocol

CP1. When replica i receives the order request message for the $CP\_INTERVAL^{th}$ request since the last checkpoint, the replica sends the speculative response to all other replicas in addition to the client.

CP2. Replica receives a commit certificate for the $CP\_INTERVAL^{th}$ request, forms a checkpoint message, and relays the checkpoint message to all other replicas.

After receiving a commit certificate for the request and processing it as in step 4b.1, replica i forms a $\langle \text{CHECKPOINT}, n, h, a, i \rangle_{\sigma_i}$ message and sends it to all replicas. n is the sequence number, h is the history, and a is a snapshot of the application state when every request in history h has been executed.

Additional Pedantic Details: The application snapshot state includes the cached responses to client requests that are ordered at $n' \leq n$. If the replica has not received the speculative response corresponding to the commit certificate, then the replica initiates the fill hole protocol described in the section G.1.2.

Additional Pedantic Details: Replica i can receive a commit certificate from the client or by receiving $2f+1$ matching speculative response messages directly from other replicas. The replica considers commit certificates gathered in either manner to be equivalent.

CP3a. Replica receives $f+1$ matching checkpoint messages and considers the checkpoint stable.

After receiving $f+1$ matching checkpoint messages, replica i considers the request stable and garbage collects any request with sequence number $\leq n$ and makes an up call into the application to garbage collect application state.

Fill Hole

The Fill Hole sub protocol is used when a replica misses an order request message from the primary – either because the network is faulty or the primary is faulty. The purpose of the fill hole protocol is to ensure that non-faulty replicas receive an order request message for each request. When a hole is recognized due to an out of order order request message, a non-faulty replica simply requests the ordered requests from the primary; the history sent to fill the hole is unconstrained. If the hole is recognized due to the receipt of a commit certificate, then the sequence of order request messages that are used to fill that hole must culminate in the history specified by the commit certificate.

FH1. Replica recognizes that there is a hole in the sequence of ordered requests and asks the primary to fill in the hole.

Replica i sends a fill hole message $\langle \text{FILL-HOLE}, v, max_n + 1, n, i \rangle_{\sigma_i}$ to the primary and starts a timer.

Additional Pedantic Details: Replica i recognizes the existence of a hole when it receives an order request message out of order, when it receives a commit certificate for a request that it has not received an order request for, or when it receives a checkpoint message without receiving all appropriate order requests.

FH2. Primary receives a fill hole request message and sends order requests for the missing requests to the replica.

Upon receiving a message $\langle \text{FILL-HOLE}, v, k, n, i \rangle_{\sigma_i}$ from replica i, the primary p sends an order request message $\langle \langle \text{ORDER-REQ}, v, n', h_{n'}, d, ND \rangle_{\sigma_p}, m' \rangle$ to i for each request m' that p ordered at sequence number $k \leq n' \leq n$ during the current view v.

Additional Pedantic Details: If the primary has a stable checkpoint for a request subsequent to k then the proof of checkpoint stability and the checkpoint itself can be sent as a replacement for some subset of the order requests.

FH2a. Replica receives every missing order request before the timer expires.

Replica *i* processes the order request messages as described in protocol step 3 and removes the timer.

Additional Pedantic Details: Contradictory order request messages constitute a proof of misbehavior as discussed in protocol step 4d.

FH2b.1. Replica does not receive every missing order request before the timer expires and asks the other replicas for help in filling in the holes.

When the timer fires the replica broadcasts the $\langle \text{FILL-HOLE}, v, k, n, i \rangle_{\sigma_i}$ to all other replicas and waits for responses.

FH2b.2. Replica receives a fill hole message from another replica and responds with the appropriate set of order requests.

Upon receipt of $\langle \text{FILL-HOLE}, v, k, n, i \rangle_{\sigma_i}$ from replica *i*, replica *j* forwards the order request message for sequence numbers *k* through *n* to *i*.

Additional Pedantic Details: If replica *j* receives fill hole messages from $f+1$ distinct replicas during view *v*, then *i* initiates the view change protocol.

Additional Pedantic Details: If replica *i* receives an order request message that is inconsistent with one that it received (either directly conflicting or resulting in an impossible history combination, then that consists of a POM and can be distributed to all other replicas to force the view change.

G.1.3 Key Differences

Our protocol differs in a couple of key ways, each difference influences the others.

1. The agreement protocol can end after 1 or 2 replica messages. PBFT ends after 2 or 3 replica (non-primary) messages.

2. The checkpoint protocol includes a commit phase which is a second all to all communication that confirms the state of the checkpoint. PBFT requires only a single all to all communication.

3. The view change protocol includes an all to all confirmation of the new view. PBFT begins accepting messages in the new view as soon as the new view message is received.

4. We divorce the initiation of a view change from the commitment to the view change.

By shortening the common case agreement protocol, we have required replicas to remain active even when they locally think that the primary is in fact faulty. We have also moved a confirmation step from the common agreement protocol to the uncommon checkpoint and view change protocols.

G.1.4 Safety and Liveness

The primary purpose of the safety proof is to ensure that no non-faulty replica will make a bad choice that results in the system being in an inconsistent state. The purpose of liveness is to ensure that, when relevant, a non-faulty replica eventually makes a choice.

Safety

We define the following predicates to facilitate our proof of the safety properties of Zyzzyva. Note that each of these predicates is defined with respect to non-faulty clients and replicas.

- client–delivered$(o,n,h,c) \equiv$ client c accepts the response to operation o as the n^{th} request in the request history h.

- op–completed$(o,v,n,h,c) \equiv$ client c accepts the response to operation o as the n^{th} request in the request history h based off of view v messages.

- ordered$(o,v,n,h,i) \equiv$ replica i received an order request for operation o as n^{th} request in the history h during view v.
- commit–cert$(o,v,n,h,i) \equiv$ replica i received a commit certificate confirming operation o as the n^{th} request in history h during view v.
- checkpoint–vote$(v,n,h,i) \equiv$ replica i has a history h of length n when initiating a checkpoint.
- checkpoint$(v,n,h,i) \equiv$ replica i considers the checkpoint at sequence number n with history h to be stable in view v after receiving $2f+1$ matching checkpoint confirmation messages.
- view–vote$(v,n_{CC},h_{CC},n_{SR},h_{SR},i) \equiv$ replica i commits to a view change with a history of ordered requests that extends to h_{SR} and a history of committed requests extending to h_{CC}.
- new–view$(v,n,h,i) \equiv$ replica i receives a new view message with history h of length n.
- confirm–view$(v,n,h,i) \equiv$ replica i receives $2f+1$ confirm view messages and begins view v with history h of length n as the base state.
- locally–commited$(o,v,n,h,i) \equiv \exists n' \geq n$ and h' such that h is a prefix of h' such that commit–cert(o,v,n,h,i) or confirm–view(v,n',h',i).

Throughout the proofs we make the following assumptions:

- There are $3f+1$ replicas.
- At most f replicas are faulty.
- All non-faulty replicas follow the protocol faithfully.

The safety property that we are interested in maintaining is that if non-faulty clients c and c' accept requests o and o' with sequence number n and n', then $n < n'$ implies that o was executed prior to o'.

Theorem 4 (Safety). *If* client–delivered(o,n,h,c), client–delivered(o',n',h',c'), *c and c' are non-faulty clients, and $n \leq n'$ then h is a prefix of h'.*

We first observe that Zyzzyva is based on a series of views. The following observation follows directly from the protocol description.

Observation 1. *If* client–delivered(o,n,h,c) *for non faulty client c then $\exists$ view v such that* op–completed(o,v,n,h,c).

Proof. Protocol steps 4a and 4b.2. □

It follows from Observation 1 that in order to prove Theorem 4 it is sufficient to prove the following Theorem describing the relationship between operations that complete at correct clients c and c' in view v and v' respectively.

Theorem 5. *If* op–completed(o,v,n,h,c) *and* op–completed(o',v',n',h',c'), *$n' \geq n$, and c and c' are non-faulty, then h is a prefix of h'.*

Protocol steps 4a and 4b.2 define the conditions under which a non-faulty client accepts requests. The following Observation expresses those conditions with respect to the predicates defined above.

Observation 2. *If* op–completed(o,v,n,h,c) *then either (a) $\forall$ non-faulty replicas i* ordered(o,v,n,h,i) *or (b) $\exists$ $f+1$ non-faulty replicas j such that* locally–commited(o,v,n,h,j).

Proof. Clients accept a response in steps 4a and 4b.2.

In step 4a, a client accepts the response only after having received matching speculative response messages from all $3f+1$ replicas. Non-faulty replicas only send speculative response messages if they order the request based on an order request message from th primary in 3b, completing the proof.

In step 4b.2, a client accepts the response only after having received matching local commit messages from $2f+1$ replicas. Non-faulty replicas send local commit messages after having received a commit certificate for that request in step 4b.1 or observing that a subsequent message has been committed as described in 4c. □

Non-faulty replicas begin accepting messages in view v only after the view is confirmed by a quorum of other replicas. The following Lemma shows that all non-faulty replicas that begin a view v do so with identical histories and sequence numbers.

Lemma 1. *If* confirm–view(v,n,h,i) *and* confirm–view(v,n',h',j) *for non-faulty replicas i and j then* $n=n'$ *and* $h=h'$.

Proof. If $v=0$ then the result follows from protocol step 0.

Assume $v>0$. Since confirm–view(v,n,h,i) is true only when i receives $2f+1$ matching view confirmation messages and there are $3f+1$ replicas total with at most f faults, i and j there is at least one non-faulty replica whose view confirmation message was accepted by both i and j. It follows from protocol step VC4 that non-faulty replicas send only one view confirmation message, so $n=n'$ and $h=h'$. □

We now show certain relationships between the states at replicas for a given sequence number n in view v. Any non-faulty replica that orders at most one request at sequence number n in view v

Lemma 2. *If* ordered(o,v,n,h,i) *and* ordered(o',v,n,h',i) *for non-faulty replica i then* $h=h'$ *and* $o=o'$.

Proof. Follows from protocol step 3 that non-faulty replicas order exactly one request at sequence number n during view v. □

If a non-faulty replica accepts a commit certificate for a request at sequence number n in view v, then at least $f+1$ non-faulty replicas ordered that request with sequence number n.

Lemma 3. *If* commit–cert(o,v,n,h,i) *at non-faulty replica* i *then* $\exists f+1$ *non-faulty replicas* j *such that* ordered(o,v,n,h,j).

Proof. It follows from protocol step 4b.1 that a non-faulty replica only accepts valid commit certificates. It follows from the definition of a valid commit certificate in protocol step 4b and the assumptions of $3f+1$ total replicas and at most f faulty replicas that a commit certificate includes matching speculative response messages from at least $f+1$ non-faulty replicas. It follows from protocol step 3 that each of these non-faulty replicas ordered the request at sequence number n, completing the proof. □

Further, if $f+1$ non-faulty replicas order the same request at n then any commit certificate at n must be for the same request.

Lemma 4. *If* commit–cert(o,v,n,h,i) *and* $\exists f+1$ *non-faulty replicas* j *such that* ordered(o',v,n,h',j) *then* $h=h'$ *and* $o=o'$.

Proof. Protocol step 4b implies that a commit certificate requires $2f+1$ replicas to send matching speculative replies. Since there are $3f+1$ replicas in total, at least one of the $2f+1$ replicas is in the set that ordered h' so $h=h'$. □

Similarly, there can be only one commit certificate for a given sequence number n in view v.

Lemma 5. *If* commit–cert(o,v,n,h,i) *and* commit–cert(o',v,n,h',j) *for non-faulty replicas i and j then* $h = h'$ *and* $o = o'$.

Proof. Follows from Lemmas 4 and 3. □

Lemmas 2 through 5 imply that only one operation can complete with sequence number n in view v. The following Lemmas relate operations that are ordered and committed by replicas with sequence numbers $n-1$ and n in view v.

Lemma 6. *If* ordered(o,v,n,h,i) *and* confirm–view(v,n_0,h_0,i) *and* $n > n_0+1$ *then* ordered$(o',v,n-1,h',i)$ *or* commit–cert$(o',v,n-1,h',i)$ *and* $h' = h-o$.

Proof. Follows from protocol steps 3 and 4b.1 that non-faulty replica i does not order the n^{th} request unless it has ordered or received a commit certificate for the $n-1^{st}$ request or n is the first sequence number of view v. □

Lemma 7. *If* ordered(o,v,n,h,i) *for non-faulty replica i, then* $\exists n_0,h_0$ *such that* $n_0 < n$ *and* h_0 *is a prefix of h and* confirm–view(v,n_0,h_0,i).

Proof. It follows from protocol step 0 and steps VC2 VC5 that non-faulty replicas only accept view v messages when the view has been confirmed, so $\exists n_0,h_0$ such that confirm–view(v,n_0,h_0,i). It follows from protocol steps 3 and protocol step 4b.1 that $n > n_0$. In order to show that h_0 is a prefix of h, we proceed by induction on the difference between n and n_0.

Base case of $n = n_0+1$. It follows from 3 that non-faulty replica i appends o to the history h_0 in order to get history h and h_0 is thus a prefix of h.

Inductive step for $n > n_0 + 1$. It follows from Lemma 6 that ordered$(o', v, n-1, h', i)$ or commit–cert$(o', v, n-1, h', i)$ such that $h' = h - o$ is true. In the first case the proof is complete by the inductive hypothesis. In the second case, it follows from Lemma 3 that there are $f+1$ non-faulty replicas that ordered o' as the $n-1^{st}$ request. It then follows from Lemma 1 that non-faulty replicas confirm the same base state for a view and from protocol steps 0, VC2, and VC5 that non-faulty replicas only accept view v messages if they have confirmed the view has begun. The inductive step then completes the proof. □

Lemma 8. *If* commit–cert(o, v, n, h, i) *for non-faulty replica* i *and* $\exists f+1$ *non-faulty replicas* j *such that* ordered(o', v, n', h', j) *and* $n' \geq n$ *then* h *is a prefix of* h'.

Proof. We proceed by induction on $n' - n$.

For the base case of $n' = n$, the conclusion follows from Lemma 4.

Consider the case when $n' > n$. It follows from Lemma 6 that ordered$(o'', v, n'-1, h'-o', j)$ or commit–cert$(o'', v, n'-1, h'-o', j)$. In the first case, we proceed by induction on commit–cert(o, v, n, h, i) and ordered$(o'', v, n'-1, h'-o', j)$. In the second case it follows fom Lemma 3 that $\exists f+1$ non-faulty replicas k such that ordered$(o'', v, n'-1, h'-o', k)$, and we again proceed by induction. □

Lemma 9. *If* commit–cert(o, v, n, h, i) *and* commit–cert(o', v, n', h', j) *for non-faulty replicas* i *and* j *and* $n' \geq n$ *then* h *is a prefix of* h'.

Proof. We proceed by induction on $n' - n$.

Base case of $n = n'$ then the result follows from Lemma 5.

Consider the case where $n < n'$. It follows from Lemma 3 that $\exists f+1$ non-faulty replicas k such that ordered(o', v, n', h', k). It follows from Lemma 6 that ordered$(o'', v, n'-1, h'-o, k)$ or commit–cert$(o'', v, n'-$

$1, h' - o, k)$ is true for each of the $f+1$ ks. In the first case the conclusion follows from Lemma 8. In the second case we proceed by induction on commit–cert(o, v, n, h, i) and commit–cert$(o'', v, n' - 1, h' - o, k)$. □

Lemma 10. *If* $\exists f+1$ *non-faulty replicas* i *such that* ordered(o, v, n, h, i) *and* commit–cert(o', v, n', h', j) *for non-faulty replica* j *and* $n' \geq n$ *then* h *is a prefix of* h'.

Proof. If $n' = n$ then the conclusion follows from Lemma 4.

If $n' > n$ then we proceed by induction on the number of sequence numbers between n and n' that have valid commit certificates in view v.

For the base case when there are no such sequence numbers, Lemma 3 implies that $\exists f+1$ non-faulty replicas k such that ordered(o', v, n', h', k). Since there are $3f+1$ replicas total and at most f faulty replicas, it follows that at least one replica i is also a replica k. The conclusion thus follows from repeated application of Lemma 6.

For the inductive step, consider the sequence number n'' such that $n < n'' < n'$ and commit–cert(o'', v, n'', h'', k) for non-faulty replica k. It follows from Lemma 9 that h'' is a prefix of h' and we proceed by induction on ordered(o, v, n, h, i) and commit–cert(o'', v, n'', h'', k). □

Having established properties relating requests that are ordered and committed during view v, we no proceed to relate these requests to the view change that transfers the system from view v to view $v+1$.

The requests that are ordered and committed by a specific replica are present in the view vote submitted by that replica according to the following Lemmas.

Lemma 11. *If* view–vote$(v, n_{CC}, h_{CC}, n_{SR}, h_{SR}, i)$ *for non-faulty replica* i*, then* $n_{CC} \leq n_{SR}$ *and* h_{CC} *is a prefix of* h_{SR}.

Proof. Follows from protocol step VC2. □

Lemma 12. *If* ordered(o,v,n,h,i) *is true* $\forall$ *non-faulty replicas* i *and* view–vote$(v+1,n_{CC},h_{CC},n_{SR},h_{SR},j)$ *is true for non-faulty replica* j *then* $n \leq n_{SR}$ *and* h *is a prefix of* h_{SR}.

Proof. It follows from Lemma 4 that no other request will be inserted at sequence number n during view v. Either n is less than the most recent commit certificate received by j or greater than that sequence number. In the first case the conclusion follows from protocol step VC2 and Lemma 11. In the second case it follows from protocol step VC2 and protocol step 3. □

Lemma 13. *If* commit–cert(o,v,n,h,i) *and* view–vote$(v+1,n_{CC},h_{CC},n_{SR},h_{SR},i)$ *are true for non-faulty replica* i *then* $n \leq n_{CC}$ *and* h *is a prefix of* h_{CC}.

Proof. It follows from Lemma 5 that no other commit certificate can be accepted by i at sequence number n in view v. Either n is the largest sequence number for which i has received a commit certificate or $\exists n' > n$ such that commit–cert(o',v,n',h',i). In the first case, the conclusion follows from protocol step 3 and VC2. In the second case it follows from Lemma 9 that h is a prefix of h' and we proceed by induction on commit–cert(o',v,n',h',i) and view–vote$(v+1,n_{CC},h_{CC},n_{SR},h_{SR},)$. □

Lemma 14. *If* confirm–view(v,n_0,h_0,i) *and* view–vote$(v+1,n_{CC},h_{CC},n_{SR},h_{SR},i)$ *for non-faulty replica* i *then* $\forall n{:}n_0 < n \leq n_{CC}$ commit–cert(o,v,n,h,i) *or* ordered(o,v,n,h,i) *and* h *is a prefix of* h_{CC} *and* $\forall n'{:}n_{CC} < n' \leq n_{SR}$ ordered(o',v,n',h',i) *and* h' *is a prefix of* h_{SR}.

Proof. Follows from protocol steps 3, 4b.1, VC2. □

We now relate requests that complete based on messages sent during view v to any initial case that can be considered for view $v+1$.

Lemma 15. *If* $\exists f+1$ *non-faulty replicas* i *such that* commit–cert(o,v,n,h,i) *and* new–view$(v+1,n',h',j)$ *then* $n' \geq n$ *and* h *is a prefix of* h'.

Proof. It follows from Lemma 13 that $\exists f+1$ non-faulty replicas i such that $n_{CC} \geq n$ in their respective view votes. Since there are $3f+1$ replicas and the new view message is composed of $2f+1$ view votes by protocol step VC3, at least one of the view votes with $n_{CC} > n$ must be included in every new view message. It thus follows from protocol step VC4 that h is a prefix of h'. □

Lemma 16. *If* ordered(o,v,n,h,i) *is true for all non-faulty replicas* i *and* new–view$(v+1,n',h',j)$ *for non-faulty replica* j *and* $n \leq n'$ *then* h *is a prefix of* h'.

Proof. It follows from Lemma 12 that all for all non-faulty replicas i $n_{SR} \geq n$ in their view votes. Since there are $3f+1$ replicas total and at most f faulty replicas (both by assumption) and the new view message contains view votes from $2f+1$ replicas by protocol step VC3, the new view message contains at least $f+1$ non-faulty replicas. It thus follows from protocol step VC4 that h is a prefix of h'. □

Lemma 17. *If* confirm–view(v,n,h,i) *and* new–view$(v+1,n',h',j)$ *for non-faulty replicas* i *and* j *then* $n' \geq n$ *and* h *is a prefix of* h'.

Proof. It follows from protocol step VC5 that a non-faulty replica commits to a view after receiving $2f+1$ matching commit view messages. It follows from protocol step VC4 that non-faulty replicas send commit view messages only after receiving $2f+1$ a valid view change message. It follows from protocol step VC2 that any non-faulty replicas will include either the new view message for view v or a commit certificate with higher sequence number in view v in their view $v+1$ view vote messages. Since a new view consists of $2f+1$ view vote messages and there are $3f+1$ total replicas, at most f of which are faulty, at least one

non-faulty replica is included in both sets and it follows from protocol step VC4 that $n' \geq n$ and h is a prefix of h'. □

Lemma 18. *If* confirm–view(v,n,h,i) *and* confirm–view$(v+1,n',h',j)$ *for non-faulty replicas i and j then* $n' \geq n$ *and h is a prefix of* h'.

Proof. Follows from protocol step VC4 and Lemma 17. □

The following Lemma shows that for any confirmed views v and $v' > v$, the history confirmed in v is a prefix of the history confirmed in v'.

Lemma 19. *If* confirm–view(v,n,h,i) *and* confirm–view(v',n',h',j) *for non-faulty replicas i and j and* $v' > v$ *then* $n' \geq n$ *and h is a prefix of* h'.

Proof. If $v' = v+1$ then the conclusion follows from Lemma 18.

If $v' > v+1$ then we proceed by induction on the number of views v'' such that $v < v'' < v'$ and confirm–view(v'',n'',h'',k) for non-faulty replica k.

In the case where no such view v'' exists, no requests can be ordered or committed in views $v \leq x < v''$, so by protocol step VC2 the view votes for non faulty replica k' are the same in all views x. It thus follows from Lemma 17 and VC4 that the conclusion holds.

In the case where such a view v'' exists, we proceed by induction on the pair of views v and v'' and the pair v'' and v. □

Lemma 20. *If* op–completed(o,v,n,h,c) *for non-faulty client c and* confirm–view(v',n',h',i) *for non-faulty replica i and* $v < v'$ *then* $n \leq n'$ *and h is a prefix of* h'.

Proof. It follows from Observation 2 that (a) $\forall$ non-faulty replicas i ordered(o,v,n,h,i) or (b) $\exists\, f+1$ non-faulty replicas j such that commit–cert(o,v,n,h,j) or (c) $\exists$ non-faulty replica k such that confirm–view(v'',n'',h'',k), $v < v'' < v'$ and $n < n'' \leq n'$.

If $v' = v+1$ then it follows from Lemmas 15, 16, and 17 that $n \leq n'$ and h is a prefix of h'.

If $v' > v+1$, then we proceed by induction on the number of views v''' such that $v < v''' < v'$ and $\exists$ non-faulty replica g such that confirm–view(v''',n_0,h_0,g). If such a view exists then h_0 is a prefix of h' by Lemma 19 and h is a prefix of h_0 by induction on v and v'''. In the case where no such view v''' exists, no request can be ordered or committed in views $v \leq x < v'''$, so by protocol step VC2 the view votes for the non-faulty replicas g are the same in all views x. It thus follows from Lemmas 15, 16, and 17 that $n \leq n'$ and h is a prefix of h'. □

Having established the previous 20 preliminaries, we can finally show a set of Lemmas that relate completed operations and the histories associated with the completed operations. First, operations that complete in the same view

Lemma 21. *If* op–completed(o,v,n,h,c) *and* op–completed(o',v,n',h',c') *and* $n' \geq n$ *then* h *is a prefix of* h'.

Proof. If op–completed(o,v,n,h,c) then confirm–view(v,n_0,h_0,i) is true for some non-faulty replica i. Observation 2 that either all non-faulty replicas i ordered(o,v,n,h,i) or at least $f+1$ non-faulty replicas j locally–commited(o,v,n,h,j), similarly either all non-faulty replicas i ordered(o',v,n',h',i) or at least $f+1$ non-faulty replicas k locally–commited(o',v,n',h',k).

There are three cases to consider.

Case 1: $n_0 < n \leq n'$. It follows from Since $n > n_0$ by assumption, locally–commited$(o,v,n,h,j) \Rightarrow$ commit–cert(o,v,n,h,j). A similar argument holds for op–completed(o',v,n',h',c'). The conclusion then

follows from Lemmas 6, 8, 9, and 10.

Case 2: $n \leq n_0 < n'$. It follows from the definition of locally–commited(o,v,n,h,i) and the assumption that $n \leq n_0$ that confirm–view(v,n_0,h_0,i) and h is a prefix of h_0. It follows from the assumption that $n_0 < n'$ and op–completed(o',v,n',h',c') that either for all non-faulty replicas i ordered(o',v,n',h',i) or $\exists\, f+1$ non-faulty replicas k such that commit–cert(o',v,n',h',k). If the latter is true, then it follows from Lemma 3 that $f+1$ non-faulty replicas j ordered(o',v,n',h',j). It thus follows from Lemma 7 that h_0 is a prefix of h' and hence that h is a prefix of h'.

Case 3: $n \leq n' \leq n_0$. It follows from the definition of local commit that h and h' are both prefixes of h_0. Since h is a history of length n and h' is a history of length $n' \geq n$, h must be a prefix of h'. □

Lemma 22. *If* op–completed(o,v,n,h,c) *and* op–completed(o',v',n',h',c') *and* $n \leq n'$ *and* $v < v'$ *then* h *is a prefix of* h'.

Proof. It follows from Observation 2 that (a) $\forall$ non-faulty replicas i ordered(o',v',n',h',i) or (b) $\exists\, f+1$ non-faulty replicas j such that commit–cert(o',v',n',h',j) or (c) $\exists$ non-faulty replica k such that confirm–view(v',n'',h'',k), and $n' \leq n''$

If (a) or (b) hold then it follows from Lemmas 3 and 7 that confirm–view(v',n_0,h_0,i) for non-faulty replica i and h_0 is a prefix of h'. It then follows from Lemma 20 and $v < v'$ that h is a prefix of h_0, completing the proof.

If (c) holds, then h' is a prefix of h_0 by definition and it follows from Lemma 20 that h is a prefix of h_0. Since $n \leq n'$, h must be a prefix of h'. □

Lemma 23. *If* op–completed(o,v,n,h,c) *and* op–completed(o',v',n',h',c') *and* $n \leq n'$ *and* $v > v'$ *then* h *is a prefix of* h'.

Proof. It follows from Observation 2 and Lemmas 3 and 7 that confirm–view(v, n_0, h_0, i) is true for some non-faulty replica i. It then follows from Lemma 20 that $n' \leq n_0$ and h' is a prefix of h_0. It thus follows from $n \leq n'$ and Observation 2 that confirm–view(v, n_0, h_0, i) is true for at least $f+1$ non-faulty replicas and that h is a prefix of h'. Since $n' > n$ h must be a prefix of h'. □

Theorem 4 *If* client–delivered(o, n, h, c), client–delivered(o', n', h', c'), *c and c' are non-faulty clients, and $n \leq n'$ then h is a prefix of h'.*

Proof. It follows from Observation 1 that it is sufficient to show the conclusion given op–completed(o, v, n, h, c) and op–completed(o', v', n', h', c'). If $v = v'$ the conclusion follows from Lemma 21. If $v < v'$ the conclusion follows from Lemma 22. If $v > v'$ the conclusion follows from Lemma 23. □

Liveness

The primary liveness property Zyzzyva maintains is that a non-faulty client eventually receives a response to every request that it issues. We maintain this property under the eventual synchrony assumption which states that the system will eventually be synchronous for a sufficiently long period of time.

Lemma 24. *During periods of synchrony, if the primary p is correct then $\forall$ operation o issued by non-faulty client c eventually $\exists$ sequence number n and history h such that* client–delivered(o, n, h, c).

Proof. If the client and primary are correct then protocol steps 1 through 3 ensure that the client receives SPEC-RESPONSE messages from all non-faulty replicas. Either the client receives $3f+1$ matching SPEC-RESPONSE messages or the client receives fewer than $3f+1$ such messages. In the former case the request completes, completing the proof. In the latter case the client receives at least $2f+1$ matching requests since there are $3f+1$ total replicas and at most f replicas can be faulty. The client then sends a COMMIT to

all replicas in protocol step 4b. All non-faulty replicas send a LOCAL-COMMIT message to the client in protocol step 4b.1, and because there are at least $2f+1$ non-faulty replicas the client request completes in protocol step 4b.2. □

Lemma 25. *During periods of synchrony, if a non-faulty client c issues a request during view v then either* $\exists$ *sequence number n and history h such that* client–delivered(o,n,h,c) *or* $\exists$ $f+1$ *non-faulty replicas i such that*

Theorem 6. *During periods of synchrony, if a non-faulty client c issues a request for operation o then* $\exists$ *sequence number n and history h such that* client–delivered(o,n,h,c).

Proof. Follows from Lemmas 24, and 25. □

Liveness in PKI-Zyzzyva is based on a couple of properties. First, if a non-faulty client does not receive a response in a timely fashion then it retransmits its request in protocol step 4c. Second, if a replica does not receive an order request from the primary in a timely fashion or detects inconsistent primary behavior then it initiates a view change in steps 4c, 4d, FH2a, and FH2b.2. Finally, if the new primary is faulty and does not complete the view change then non-faulty replicas initiate another view change.

G.2 Non-PKI Zyzzyva

Here we describe the necessary modifications to replace signatures with authenticators in Zyzzyva. We replace all signatures with authenticators, with a few exceptions related to: (1) view changes, (2) client request retransmission, and (3) a corner case of the fill hole protocol required in order to ensure consistency in the presence of faulty replicas.

G.2.1 Agreement

4. Client gathers speculative responses.

[Modification] We remove the *OR* field from the speculative response message. Since the *OR* field contains a MAC, rather than a signature, the client cannot convince another node of anything based on that field.

4b. Client receives between $2f+1$ and $3f$ matching responses, assembles a commit certificate, and transmits the commit certificate to the replicas.

[Modification] The client forms a potential commit certificate with every available speculative response, rather than only $2f+1$, since replicas may be unable to authenticate every message that the client can authenticate. A replica considers a commit certificate to be valid it it can authenticate at least $2f+1$ of the included speculative responses.

4b.1a. Replica receives a commit message from a client containing a commit certificate and acknowledges with a local-commit message.

4b.1b. Replica receives a commit message with at least f verifiable and at most $2f$ matching.

The replica sends its speculative response (without the full application reply but with just the hash) or a local-commit message (if it is already locally committed) to all other replicas. If the request is locally committed at the replica it also forwards local-commit message to the client.

4b.1b.1. Replica receives speculative response or local-commit messages directly from other replicas.

If the replica broadcasted local-commit messages in the previous step and does not receive $2f+1$ matching of local-commit message, it initiates a view change. speculative or local-commit messages) have matching history digest, then the replica responds to the client as in step 4b.1a. Otherwise the replica and initiates

the view change protocol.

4b.1b.2. Replica receives CC confirm request from another replica.

If the replica received a valid CC, then it responds with an affirmative message. Otherwise it sends no message.

4b.1b.3. Replica receives an affirmative CC confirmation from $f+1$ distinct replicas for matching requests.

The replica treats the series of messages as a commit certificate and acts as described in step 4b.

Additional Pedantic Details: If necessary, the replica responds with CC confirmations as in step 4b.1b.2

[Addition] A replica that authenticates at least $f+1$ of the included speculative responses authenticates the request if it has not already done so and has not speculatively responded to a different request for that sequence number.

4d. Client receives replies indicating inconsistent ordering by the primary and sends a proof of misbehavior (POM) to the replicas, which initiate a view change to oust the faulty primary.

[Modification] A *POM* consists of 2 distinct speculative responses for the same sequence number *n* each supported by $f+1$ replicas. Note that the replicas may be unable to verify the *POM*.

G.2.2 View Change.

All messages sent as part of the view change protocol contain digital signatures. The additional expressive power of digital signatures simplifies the protocol design at limited cost, since view changes occur rarely. We add the following additional proof gathering phase to the view change protocol—it runs after replicas receive $f+1$ indictments of the primary and before the view change messages are sent.

VC2.1. Replica requests a signature for the most recent commit and checkpoint certificate it has received.

After receiving $f+1$ primary condemnations, replica *i* sends a request to all other replicas for a signature corresponding to the most recent commit and checkpoint certificates it has received.

VC2.2. Upon receipt of a signature request for commit and checkpoint certificates, the replica sends signatures back to the requesting replica.

If replica j has received an order request for the specified commit certificate, then it replies with a signature for that order request. If replica j has sent the checkpoint message previously then j responds with a signature for that checkpoint.

VC2.3. Upon receipt of $f+1$ signatures for the same commit certificate, a replica completes the proof and sends a view change message.

A replica considers $f+1$ signatures to complete commit and checkpoint certificate proofs. Once the replica has complete proofs, the replica is able to send a view change message to the new primary.

Modification. View change messages consist of the proof for a stable checkpoint and every order request received since that checkpoint. The order requests are augmented with a collection of $f+1$ signatures for the order request at some sequence number after the checkpoint. This collection of signatures corresponds to a tentative commit certificate.

Additional Pedantic Details: A collection of $f+1$ matching signatures for the commit certificate constitutes a *strong* proof for the commit certificate; a collection of $f+1$ signatures that for a combination of order requests and commit certificates constitutes a *weak* proof for the commit certificate.

VC3. Replica receives $2f+1$ view change messages.

Modification. The primary gathers $2f+1$ non-conflicting view change messages to form a new view message. A pair of view change messages are conflicting if they contain commit certificate proofs for different values at sequence number n. A view change message with a commit certificate proof for sequence number n' is considered to implicitly contain a commit certificate for all sequence numbers $n \leq n'$.

We believe that digital signatures can be avoided in the view change protocol after replicas commit to view change using View-change-ack messages similar to PBFT.

G.2.3 Checkpoint

We modify the following step in checkpoint protocol while the other steps are the same except that digital signatures are replaced by MACs.

> CP3. Replica receives $2f+1$ matching checkpoint messages from another replica and accepts the checkpoint as stable if it has not already done so.

[Modification] Like PBFT, we modify the checkpoint protocol to include $2f+1$ MACs for non-pki version. A checkpoint is considered stable by replica i when i has received either $2f+1$ matching (and authenticatable) checkpoint messages from another replica.

Fill Hole

There is an additional problem encountered in the fill hole protocol when digital signatures are not used to sign client requests. It is possible that a replica, upon receipt of an order request from the primary, is unable to authenticate the client issued request. Unfortunately, it is impossible to tell if this failure is the fault of the client—for providing an incorrect authenticator—or the fault of the primary—for maliciously modifying the client's authenticator. In order to address this issue, we introduce a new sub protocol to the fill holes procedure—the request authentication protocol. The request authentication protocol ensures that either all

non-faulty replicas authenticate a response or agree that the n^{th} request in the execution order should be considered a no-op. This protocol is executed whenever a replica receives an order-request message from the primary containing a request that the replica is unable to authenticate.

AR1. Replica requests an authentication proof from the primary.

Upon receiving a request that a replica is unable to authenticate, replica *i* sends a message to the primary requesting proof that the request is in fact valid.

AR2. Primary receives the authentication request and responds with an available proof.

AR2a.1. Primary responds with a signed client request or commit certificate proving that the request was issued by the client.

The signature is assumed to be unforgeable and consequently proves that the client issued the request. A commit certificate must include speculative responses from at least $f + 1$ non-faulty replicas, so when transferred guarantees that at least one non-faulty replica was able to authenticate the request.

Additional Pedantic Details: A commit certificate is only a valid proof if the replicas use the two layer MAC authentication described in the modification to protocol step 3. If the second layer of MACs is not used, then a commit certificate is not transferable proof.

AR2a.2. Replica receives the signed client request, authenticates the request, and proceeds as normal.

AR2b.1. Primary has not received a signed client request or a valid commit certificate and requests an authentication proof from the other replicas.

If the primary does not have a valid signature or commit certificate for the request, then the primary gathers proof by requesting signatures from the replicas.

AR2b.2. Replica receives the signature request from the client, forms a response, and stops replying to the client.

The replicas response is a signed message containing one of the following: (a) a signature authenticating the request, (b) a signed commit certificate for that request (or a subsequent request that includes the intervening requests), or (c) a signature claiming that the request was unauthenticatable. After receiving the signature request from the primary, the replica does not perform any other actions until the signature query is resolved.

Additional Pedantic Details: The step of stopping responses to the client is necessary to ensure safety and prevent a client from receiving $2f+1$ local commit messages for a request that is eventually no-oped out of existence.

AR2b.3. Primary receives responses, generates the proof, and distributes the proof to the replicas.

The proof consists of either (a) a single signed commit certificate, (b) $f+1$ signed messages authenticating the request, or (c) $2f+1$ messages that contain neither (a) nor (b). Once the primary has gathered one of these three proofs the primary sends the proof to all other replicas.

AR2b.4. Replica receives the authentication proof from the primary.

If the proof authenticates the request (i.e. is *a* or *b* from above), then the replica proceeds as normal. If the proof does not authenticate the request (i.e. is *c* above), then the replica inserts a no-op at the appropriate place in the history and proceeds as normal for future requests—any requests that were previously ordered with higher sequence numbers are discarded and the requests themselves must be reordered using the request history that includes the no-op. In either case, after receiving the response from the primary the replica accepts order request messages again and sends the authentication proof to all other replicas.

AR2b.5. Replica receives authentication proof from other replicas.

If the replica receives two different authentication proofs, then the replica initiates the view change protocol.

Additional Pedantic Details: This ensures that a faulty primary that authenticates two distinct orders will in fact be caught and force an inevitable view change.

Future Requests. In order to limit the amount of damage that a faulty client can cause the system, subsequent requests issued by that same client must all be signed. If they do not include a signature, then they are not accepted by the primary. If the primary forwards a request from that client that does not contain a valid signature in the future, then the primary is considered faulty and the replica initiates the view change protocol.

G.2.4 Safety and Liveness

The proof of safety for non-PKI Zyzzyva is virtually identical to the proof of safety for PKI Zyzzyva; the intuitive conditions under which a client accepts the response to a request remain unchanged as do the conditions under which a replica orders or locally commits a request. Consequently, within a view the protocol and the proofs do not substantially change. The steps to maintain safety become more complicated when the view change procedure is considered.

Safety

We introduce a new predicate that captures the state of a replica that has gathered a collection of signatures that serve as a tentative proof for a commit certificate.

- commit–proof(o,v,n,h,i) that is true when replica i has gathered $f+1$ matching signatures for the order request of operation o at sequence number n with history h.

Based on this predicate and the protocol we are first able to show that non-faulty replicas gather proofs for commit certificates only if the commit certificates are consistent with each other.

Lemma 26. *If* commit–proof(o,v,n,h,i) *and* commit–proof(o',v,n',h',j) *for non-faulty replicas* i *and* j *and* $n' \geq n$ *then* h *is a prefix of* h'.

Proof. It follows from protocol step VC2.1 that non-faulty replica i (j) requests signatures to form a commit proof for sequence number n (n') only if i (j) has received a valid commit certificate for sequence number n (n'). It thus follows from Lemma 9 that h is a prefix of h'. □

Based on Lemma 26, we are able to show that a view change messages sent by non-faulty replicas are non-conflicting.

Lemma 27. *If* view–vote$(v,n_{CC},h_{CC},n_{SR},h_{SR},i)$ *and* view–vote$(v,n'_{CC},h'_{CC},n'_{SR},h'_{SR},j)$ *for non-faulty replicas* i *and* j *and* $n_{CC} \leq n'_{CC}$ *then* h_{CC} *is a prefix of* h'_{CC}.

Proof. The conclusion follows from protocol step VC2.3 and Lemma 26. □

Lemma 27 ensures that a non-faulty primary can always gather $2f+1$ consistent view change messages if given enough time.

Liveness

Liveness in the protocol is ensured based on the assumption of eventual synchrony. As was discussed in the previous section, view change messages from non-faulty replicas are guaranteed to be non-conflicting so

when allowed enough time through a period of synchrony and the exponentially growing timeouts the new primary for view v is guaranteed to collect $2f+1$ non-conflicting view change messages and consequently be able to form a valid new view message. Within a view, liveness is ensured during periods of synchrony due to the agreement protocol steps beginning at protocol step 4b.1b. If all non-faulty replicas process requests based on protocol step 4b.1a then the client receives the response to the request and the system is live. If a non-faulty replica reaches protocol step 4b.1b then either it receives sufficient messages to authenticate the commit certificate or there are at least $f+1$ non-faulty replicas that are unable to authenticate the commit certificate but are able to force a view change to occur.

Bibliography

[1] Amazon S3 Storage Service. http://aws.amazon.com/s3.

[2] Apple Backup. http://www.apple.com.

[3] Concerns raised on tape backup methods. http://searchsecurity.techtarget.com.

[4] Copan Systems. http://www.copansys.com/.

[5] Cryptographic Benchmarks. http://www.eskimo.com/~weidai/benchmarks.html.

[6] Data loss statistics. http://www.adrdatarecovery.com/content/adr_loss_stat.html.

[7] Data loss statistics. http://www.hp.com/sbso/serverstorage/protect.html.

[8] Disk at the pice of Tape - An In-Depth Examination. http://www.copansys.com/library/index.shtml.

[9] Fire destroys research center. http://news.bbc.co.uk/1/hi/england/hampshire/4390048.stm.

[10] GMail. http://www.gmail.com.

[11] Google video. http://video.gmail.com.

[12] Health Insurance Portability and Accountability Act (HIPAA). 104th Congress, United States of America Public Law 104-191.

[13] Hotmail incinerates customer files. http://news.com.com, June 3rd, 2004.

[14] “How much information ?”. http://www.sims.berkeley.edu/projects/how-much-info/.

[15] Hurricane Katrina. http://en.wikipedia.org.

[16] Industry data retention regulations. http://www.veritas.com/van/Articles/4435.jsp.

[17] IOZONE micro-benchmarks. http://www.iozone.org.

[18] Lost Gmail Emails and the Future of Web Apps. http://it.slashdot.org, Dec 29, 2006.

[19] Microsoft Hot Mail. http://mail.live.com.

[20] Microsoft Live photos. http://photos.live.com.

[21] Microsoft Sky Drive. http://skydrive.live.com.

[22] NetMass Systems. http://www.netmass.com.

[23] Network Appliance. http://www.netapp.com.

[24] Network bandwidth cost. http://www.broadbandbuyer.com/formbusiness.htm.

[25] NFS :Network file system protocol specification. InternetRFC1094.

[26] OpenSSL. http://www.openssl.org/.

[27] OS vulnerabilities. http://www.cert.com/stats.

[28] Picasa Web. http://picasaweb.google.com.

[29] Postmark macro-benchmark. http://www.netapp.com/tech_library/postmark.html.

[30] Ransomware. http://www.networkworld.com/buzz/2005/092605-ransom.html.

[31] Remote Data Backups. http://www.remotedatabackup.com.

[32] Sarbanes-Oxley Act of 2002. 107th Congress, United States of America Public Law 107-204.

[33] Spike in Laptop Thefts Stirs Jitters Over Data. Washington Post, June 22, 2006.

[34] SSPs: RIP. Byte and Switch, 2002.

[35] Tape Replacement Realities. http://www.enterprisestrategygroup.com/ESGPublications.

[36] The Wayback Machine. http://www.archive.org/web/hardware.php.

[37] US secret service report on insider attacks. http://www.sei.cmu.edu/about/press/insider-2005.html.

[38] Victims of lost files out of luck. http://news.com.com, April 22, 2002.

[39] Yahoo Mail. http://mail.yahoo.com.

[40] Yahoo Mail. http://photos.yahoo.com.

[41] You Tube. http://www.youtube.com.

[42] ServeRAID - Recovering from multiple disk failures. http://www.pc.ibm.com/qtechinfo/MIGR-39144.html, 2001.

[43] “data backup no big deal to many, until...”. http://money.cnn.com, June 2006.

[44] A. Adya et.al. FARSITE: Federated, available, and reliable storage for incompletely trusted environment. In *Proc. of 5th OSDI*, 2002.

[45] Michael Abd-El-Malek, Greg Ganger, Garth Goodson, Mike Reiter, and Jay Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[46] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. of SOSP '05*, pages 45–58, October 2005.

[47] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[48] Algirdas Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proc. IEEE COMPSAC*, pages 149–155, November 1977.

[49] M. Baker, M. Shah, D.S.Rosenthal, M. Roussopoulos, Petro Maniatis, T.J. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *EuroSys*, 2006.

[50] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE TODSC*, 1(1):87–96, 2004.

[51] L. Bassham and W. Polk. Threat assessment of malicious code and human threats. Technical report, NIST, Computer Security Division, 1994.

[52] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementally at reduced cost. In *EUROCRYPT97*, 1997.

[53] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Total Recall: System support for automated availability management. In *Proceedings of 1st NSDI*, CA, 2004.

[54] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, New York, NY, USA, 1993. ACM.

[55] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of 3rd OSDI*, February 1999.

[56] M. Castro and B. Liskov. Proactive recovery in a Byzantine Fault-Tolerant System. In *Proceedings of 4th OSDI*, October 2000.

[57] Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, January 2001.

[58] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, November 2002.

[59] F.W. Chang, M. Ji, S. T. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proceeedings of FAST*, 2002.

[60] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Proceedings of 8th Symp. on Fault-Tolerant Computing*, 1978.

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

[62] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries*, San Fransisco, CA, Aug 1999.

[63] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. OSDI*, November 2006.

[64] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. of SOSP03*.

[65] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM Press.

[66] Partha Dutta, Rachid Guerraoui, and Marko Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report EPFL/IC/200499, EPFL, February 2005.

[67] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

[68] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *FAST03*, March 2003.

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

[70] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement in t + 1 rounds. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 31–41, New York, NY, USA, 1993. ACM.

[71] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proceedings of 19th ACM Symp. on Operating Systems Principles*, October 2003.

[72] Philippe Golle, Stanisław Jarecki, and Ilya Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Financial Cryptography (FC 2002)*, volume 2357 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2003.

[73] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, October 1990.

[74] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Higly durable, decentralized storage despite massive correlated failures. In *Proceedings of 2nd NSDI*, CA, March 2004.

[75] Ragib Hassan, William Yurcik, and Suvda Myagmar. The evolution of storage service providers. In *StorageSS'05*, VA,USA, 2005.

[76] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[77] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS*, 2000.

[78] F. Junquiera, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. Surviving internet catastrophes. In *Proceedings of the Usenix Annual Technical Conference*, April 2005.

[79] K. Keeton and E. Anderson. A backup appliance composed of high-capacity disk drives. In *HP Laboratories SSP Technical Memo HPL-SSP-2001-3*, April 2001.

[80] S. King and P. Chen. Backtracking intrusions. In *Proc. SOSP*, 2003.

[81] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *Software Engineering*, 12(1):96–109, January 1986.

[82] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A Durable and Practical Storage System. In *USENIX Annual Technical Conference*, Monterey, CA, June 2007.

[83] R. Kotla, L. Alvisi, and M. Dahlin. Safestore: A durable and practical storage system. Technical report, University of Texas at Austin, 2007. UT-CS-TR-07-20.

[84] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. *Technical Report: UTCS-TR-03-58*, Dec. 2003.

[85] R. Kotla and M. Dahlin. High-throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[86] R. Kotla and M. Dahlin. High-throughput byzantine fault tolerance. In *DSN*, June 2004.

[87] L. Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):51–58, April 2001.

[88] L. Lamport. Lower bounds for asynchronous consensus. In *Proc. FUDICO*, pages 22–23, June 2003.

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

[90] Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[91] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant services. In *NSDI*, 2007.

[92] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the harp file system. In *Proc. SOSP*, 1991.

[93] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[94] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.

[95] Petros Maniatis, Mema Roussopoulos, T J Giuli, David S. H. Rosenthal, Mary Baker, and Yanto Muliadi. Lockss: A peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, Feb. 2005.

[96] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE TODSC*, 3(3):202–215, July 2006.

[97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[98] Shubhendu S. Mukherjee, Joel S. Emer, and Steven K. Reinhardt. The soft error problem: An architectural perspective. In *HPCA*, 2005.

[99] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. In *Proc. OSDI*, 2006.

[100] Edmund B. Nightingale, Peter Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[101] D. Openheimer, A. Ganapathi, and D. Patterson. Why do internet systems fail, and what can be done about it. In *Proceedings of 4th USITS*, Seattle,WA, March 2003.

[102] Dave Patterson. A conversation with jim gray. *ACM Queue*, pages vol. 1, no. 4, June 2003.

[103] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), April 1980.

[104] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Trans. on Storage*, 1(2):190–212, May. 2005.

[105] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andr Barroso. Failure Trends in a Large Disk Drive Population. In *Proceeedings of FAST*, 2007.

[106] V. Prabhakaran, L. Bairavasundaram, N Agrawal, H. Gunawi A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *Proc. of SOSP '05*, 2005.

[107] H. E. Ramadan. Abort, retry, litigate: Dependable systems and contract law. In *Proceedings of HotDep '06*, 2006.

[108] K. M. Reddy, C. P. Wright, A. Hammer, and E. Zadok. A Versatile and user-oriented versioning file system. In *FAST*, 2004.

[109] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST03*, March 2003.

[110] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Comp. Comm. Review*, 27(2), 1997.

[111] R. Rodrigues, M. Castro, and B. Liskov. BASE : Using abstraction to improve fault tolerance. In *Proceedings of 18th ACM Symp. on Operating Systems Principles*, October 2001.

[112] M. Roesch. Snort–lightweight intrusion detection for networks. In *Proc LISA*, 1999.

[113] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[114] David S. H. Rosenthal, Thomas S. Robertson, Tom Lipkis, Vicky Reich, and Seth Morabito. Requirements for digital preservation systems: A bottom-up approach. *D-Lib Magazine*, 11(11), Nov. 2005.

[115] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *ACM Symposium on Operating Systems Principles*, pages 1–15, 1999.

[116] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM TOCS*, November 1984.

[117] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of 17th ACM Symp. on Operating Systems Principles*, December 1999.

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

[119] F. Schnieder. Implementing fault-tolerant services using the state machine approach. *ACM Comp. Surveys*, 22(3):299–319, Sept. 1990.

[120] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceeedings of FAST*, 2007.

[121] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proc. MASCOTS*, October 2004.

[122] Seagate. Get S.M.A.R.T for reliability. Technical Report TP-67D, Seagate, 1999.

[123] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. DSN*, 2002.

[124] Atul Singh, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Conflict-free quorum-based bft protocols. Technical Report 2007-1, Max Planck Institute for Software Systems, August 2007.

[125] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of 6th OSDI*, 2004.

[126] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in a comprehensive versioning file system. In *Proc. of FAST 2003*.

[127] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, Washington, USA, October 2007.

[128] U. Voges and L. Gmeiner. Software diversity in reacter protection systems: An experiment. In *In IFAC Workshop SAFECOMP79*, May 1979.

[129] H. Weatherspoon and J. Kubiatowicz. Erasure Coding versus replication: A quantitative comparison. In *Proceedings of IPTPS*, Cambridge,MA, March 2002.

[130] M. Welsh, D. Culler, and E. Brewer. SEDA : An architecture for well conditioned, scalable internet services. In *Proceedings of 18th ACM Symp. on Operating Systems Principles*, October 2001.

[131] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. ICDCS*, 2000.

[132] J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliccote, and P. K. Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, Aug. 2000.

[133] Q. Xin, T. Schwarz, and E Miller. Disk infant mortality in large storage systems. In *Proc of MASCOTS '05*, 2005.

[134] Junfeng Yang, Can Sar, and Dawson Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proc. OSDI*, 2006.

[135] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of 6th OSDI*, December 2004.

[136] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of 19th ACM Symp. on Operating Systems Principles*, October 2003.

Index

Vita

Ramakrishna Rao Kotla was born in Hyderabad, India, on March 5, 1976, of Sanjeeva Rao Kotla and Varalaxmi Kotla. He studied at various schools before completing the high school at St. Mary's High school, and then attended St. Mary's Junior college in Hyderabad, India. He then studied at the Indian Institute of Technology, Kharagpur, where he received the Bachelor of Technology degree in Electronics and Electrical Communication Engineering in May 1998. Thereafter, he worked as Research and Development Engineer at Synopsys Inc., in Bangalore,India, and Austin, USA, during 1998-2001.

He started his full time graduate studies at the University of Texas in August 2001. He received Master of Science in Engineering degree in Electrical and Computer Engineering in 2003. He received two best paper Awards at ACM Symposium on Operating Systems Principles(SOSP) and USENIX Annual Technical Conference(USENIX) for the research papers describing parts of his Doctoral research work. He is working at Microsoft Research as a researcher since March 2008.

Permanent address: Plot No. 15, Sikhara Enclave, Champapet, Hyderabad,
Andhra Pradesh, 500079, India

This dissertation was typeset with LaTeX† by the author.

†LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.