A read/write protocol family for versatile storage infrastructures

JAY J. WYLIE

October 2005

CMU–PDL–05–108

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis committee

Prof. Lorenzo Alvisi (The University of Texas at Austin)
Prof. Gregory R. Ganger, Chair (Carnegie Mellon University)
Prof. Pradeep K. Khosla (Carnegie Mellon University)
Prof. Michael K. Reiter (Carnegie Mellon University)

To the Pittsburgh Left and all that it signifies: generosity, patience, and simplicity.

Abstract

The ideal storage infrastructure scales to meet new demands. Traditionally, the emphasis has been on the capacity and performance scalability of a storage infrastructure. Current trends towards massive storage infrastructures comprised entirely of commodity components demand broader forms of scalability. The next generation of storage infrastructures must scale to tolerate more and varied types of faults. Fault-scalability, the ability to tolerate large numbers of faults efficiently, is needed so that the simultaneous failures of multiple commodity components can be tolerated. Versatility, the ability to store objects with radically different resiliency (fault-tolerance) and performance requirements simultaneously and efficiently, is needed so that a deployed storage infrastructure can meet new demands as they are identified.

This dissertation develops a set of related protocols for reading and writing data objects—called the *Read/Write Protocol Family* (R/W-PF)—that enables a versatile storage infrastructure to be built. The R/W-PF provides versatility: objects with different per-object resiliency requirements can be stored in the same storage infrastructure. The costs (response time, number of servers required, etc.) of storing an object are commensurate with its resiliency requirements. The R/W-PF incorporates versatile storage mechanisms, such as erasure codes, witnesses, and quorums, in its design, allowing the efficiency of read and write access to stored objects to be tuned to meet capacity and performance requirements.

Measurements of PASIS, a prototype storage system based on the R/W-PF, demonstrate its versatility. These measurements show that the R/W-PF also provides fault-scalability. Measurements show the differing performance costs associated with various resiliency requirements and the workload-dependent merits of the storage mechanisms incorporated in the R/W-PF. The significant trade-offs associated with resiliency and storage mechanism choices underscore the importance of versatility in storage infrastructures.

Acknowledgements

I have been fortunate to work with a number of great people during my time at Carnegie Mellon. I have enjoyed every moment of my collaboration with my colleagues Garth Goodson and Mike Abd-El-Malek. I thank them for their patience with me and their contributions to this research. I am especially grateful for their contributions to the development of the PASIS prototype.

Greg Ganger has always been available as an advisor and taught me how to think carefully and creatively. Mike Reiter has always supported my research direction and taught me how to think about distributed systems. Pradeep Khosla admitted me to ECE and gave me the freedom to pursue the research that interested me. Lorenzo Alvisi gave detailed feedback on my dissertation and asked probing questions at my defense. I thank all of my committee members for their contributions to my intellectual development.

The Parallel Data Lab has been a wonderful environment in which to work. I thank all of my PDL colleagues for making the last five years enjoyable. Karen Lindenfelser, Linda Whipkey, and Joan Digney are great people and have helped me out too many times to count. I thank the staff and students who have maintained the computing infrastructure I abused during my time here. Finally, I thank all of the industry attendees at various PDL events from member companies of the Parallel Data Consortium for their constructive feedback.

I thank both of my parents for their support and encouragement. I thank Krista, Emree and Madeline for always being there for me. I thank Joelle, Aaron, and Sophie for being my family in Pittsburgh. I thank Jacques Katz and David Petrou for improving Pittsburgh's quality of life. Finally, I am grateful for all of the friends I made at the University of Waterloo and at Carnegie Mellon University during this twelve year intellectual odyssey.

Contents

Figures

Tables

1 Introduction

This chapter motivates the need for versatile storage infrastructures and overviews the Read/Write Protocol Family (R/W-PF), the primary contribution of this dissertation. It presents the *thesis statement* and outlines how this dissertation proves the thesis statement.

1.1 Versatile storage infrastructures

Fault-tolerant distributed storage systems protect data by spreading it redundantly across a set of servers. In the design of storage access protocols, determining which kinds of faults to tolerate, the number of faults to tolerate, and assumptions to make about the system model, are important and difficult decisions. Fault models range from tolerating crash failures to Byzantine (arbitrary) failures, and system assumptions about delays range from synchronous to asynchronous. These decisions affect the storage access protocol employed, which can have a major impact on cost and performance. A storage access protocol can be designed to provide consistency under very weak assumptions—e.g., Byzantine failures in an asynchronous system—but this induces potentially unnecessary costs. Making few, weak assumptions, requires that more servers be accessed to provide a given capacity. Alternatively, designers can "assume away" certain faults. Such a design decision is made to improve performance of the storage system or to decrease its cost.

Traditionally, the fault model decision is hard-coded during the design of the storage access protocol. This traditional approach has two significant shortcomings. First, it limits the utility of the resulting storage system. Either the storage access protocol incurs unnecessary costs for some stored objects, or it cannot be used to store other objects with more stringent requirements. The natural consequence is distinct storage access protocols, system designs, and system implementations for each distinct fault model. Second, all objects stored in any given storage system must use a similar fault model, either paying unnecessary costs for less critical data or under-protecting more critical data.

We promote an alternative approach in which the decision of which faults to tolerate is shifted from design time to object creation time. This shift is achieved through the use of a *family* of storage access protocols that share a common client-server interface. A *protocol family* supports different fault models in the same way that most access protocols support varied numbers of failures: by simply changing the number (set) of servers accessed and the corresponding client logic for such accesses. A protocol family enables a given storage infrastructure to be used for a mix of fault models and number of faults tolerated, chosen independently for each stored object.

1.2 Thesis statement

A *protocol family* approach to building read/write storage enables the construction of versatile storage infrastructures that, once deployed, can be tuned to each stored object's specific resiliency requirements and performance requirements.

This thesis statement was validated in the following manner:

(1) The Read/Write Protocol Family (R/W-PF) was developed as was an accompanying proof of its correctness. The R/W-PF embodies and demonstrates the feasibility of the protocol family concept.

(2) A prototype storage system, called PASIS, was built with the R/W-PF to illustrate versatile distributed storage and enable experimentation.

(3) Empirical and analytical results show that there are resiliency and performance trade-offs among the member protocols of the R/W-PF.

These trade-offs delineate the broad range of versatility that the R/W-PF offers, in the context of the PASIS prototype.

(a) Experiments with PASIS determined the costs of the different resiliency provided by different protocol family members—that is, the type and number of faults tolerated as well as the timing model—in terms of client and server computation, average response time, and average system throughput.

(b) Experiments with PASIS determined the workload-specific costs and benefits of erasure-coding stored objects, employing quorums to provide throughput-scalability, and employing witnesses to improve network- and storage-efficiency.

(c) Experiments with PASIS determined the relationship between the workload (e.g., block size, read/write ratio) and performance.

(d) Analysis of quorum constructions and erasure codes identified the following characteristics of R/W-PF members: the minimum number of servers required to store objects; the network- and storage-efficiency of storing objects; and the fault-tolerance of objects stored.

An R/W-PF member is specified by its resiliency model—that is, its timing model, server failure model, and client failure model—and its versatile storage mechanisms—that is, its quorum construction, erasure-coding, and witness use. Chapter 3 describes the R/W-PF in detail and provides comments that place it in context relative to other techniques. To ensure the correct operation of R/W-PF members, limits are placed on their versatility: Chapter 5 identifies bounds on quorum construction and witness use based on the resiliency model and erasure-coding of the R/W-PF member. A detailed design of the R/W-PF is presented in Chapter 4. Building on these chapters, Chapter 6 presents a proof that R/W-PF members provide linearizable, wait-free read and write operations. Read operations by faulty clients and write operations that do not complete are excluded from the set of linearizable operations. An assumption of unbound storage capacity is required for operations to be wait-free. Taken together, Chapters 3–6 provide a concrete example of a protocol family, demonstrating the concept and its feasibility.

We built PASIS, a prototype storage system that employs the R/W-PF. Chapter 7 describes the design and implementation of PASIS. The PASIS prototype supports all of the resiliency models for R/W-PF membership that are described in Chapter 3. Some threshold quorum systems, a broad range of erasure codes, and space-efficient witnesses, are implemented in the PASIS prototype. The existence of the PASIS software artifact validates that versatile storage can be built with a protocol family.

Chapter 8 describes the evaluation of the R/W-PF. Experiments were run with the PASIS prototype to explore, empirically, a large portion of its "versatility-space". Sections 8.3 and 8.4 report response time and throughput measurements, respectively, for a large number of R/W-PF members. Both the resiliency and the space-efficiency due to erasure coding of a R/W-PF member is evaluated via these experiments. As such, these results show items 3a and 3b of the thesis statement. Other experiments measured the ability of threshold quorums to provide throughput-scalability, and of witnesses to improve the network-efficiency, of R/W-PF members (cf. Section 8.6). These results provide additional support of item 3b of the thesis statement. Other experiments, described in Chapter 8, measured the impact of object size and workload read/write ratio on the performance of R/W-PF members. These results show item 3c of the thesis statement.

The number of servers required by an R/W-PF member, as well as its space-efficiency, is determined analytically in Chapter 5. The analysis of universe size (number of servers) and space-efficiencies is made more concrete for a large set of R/W-PF members in Section 8.3.1. Collectively, these analyses show item 3d of the thesis statement. Overall, the experimental results demonstrate that the R/W-PF enables the construction of versatile storage infrastructures and that versatility is important because of the significant inherent trade-offs among resiliency, performance, and cost in storage systems.

Although not claimed directly as a contribution of this thesis, the performance results from the PASIS prototype demonstrates that an *efficient* versatile storage infrastructure can be built with the R/W-PF. If versatility came with prohibitive performance costs, the contributions of this thesis would be weakened. The versatility provided by the R/W-PF is not a panacea. Stringent resiliency requirements have real costs: storage mechanisms can ameliorate some such costs, but not all.

1.3 The Read/Write Protocol Family

The Read/Write Protocol Family (R/W-PF) is a family of storage access protocols that exploit object versioning within servers to provide consistent quorum-based access to erasure-coded objects efficiently. The R/W-PF covers a broad range of resiliency models—timing model, client failure model, and server failure model—with no changes to the client-server interface. R/W-PF members are distinguished by choices enacted in client logic: the number (set) of servers accessed and the logic employed during reads and writes. Weaker assumptions require the use of more servers and additional client computation. A single server implementation is sufficient to build versatile storage infrastructures with the R/W-PF.

Today's storage systems are dominated by heavily engineered systems comprised of dual ported disks and redundant internal networking that exhibit limited scalability. Such systems encode stored objects via striping (RAID 0), two- & three-fold replication (RAID 1), replicated striping (RAID 1/0), XOR-based parity (RAID 3/4/5), or XOR-based array codes (RAID 6). Such systems also access stored objects under the assumptions of synchrony. Single fail-stop failures of any components are tolerated and some multiple component failures are tolerated. There are members of the R/W-PF that are similar to each of these common storage configurations. However, there are also R/W-PF members that go beyond these basic configurations. For example, there are R/W-PF members that tolerate more numbers and types of failures under different timing models.

The R/W-PF incorporates versatile storage mechanisms that improve its members' performance and efficiency. The R/W-PF allows stored objects to be erasure-coded. Replication and other RAID levels are special-case erasure codes that the R/W-PF incorporates. The inclusion of erasure codes allow R/W-PF members to store objects that tolerate many server failures in a space-efficient manner. The R/W-PF incorporates witnesses—the use of space-efficient object digests in lieu of the object itself—that improve the network- and storage-efficiency of its members. Because the R/W-PF is quorum-based, it benefits from the throughput-scalability afforded by different quorum constructions. Quorum constructions are a generalization of voting systems which are employed in some contemporary storage systems.

1.4 Example R/W-PF trade-offs

There are inherent trade-offs among resiliency, performance, and cost in distributed storage systems. Versatile storage infrastructures built with the R/W-PF expose these trade-offs in the form of different R/W-PF members. To motivate the value of a versatile storage infrastructure, some trade-offs are illustrated in this section.

Consider storing a set of objects that must tolerate two server failures. If clients and servers may only crash and the timing model is synchronous, then the object can be replicated across three servers. Experimental results in Chapter 8 for the Benign, Synchronous, Replication, $t = 2$ R/W-PF member correspond to these requirements. (The symbol t indicates the total number of server failures tolerated.) To illustrate trade-offs, consider another R/W-PF member that employs an erasure code with $m = 6$ rather than replication. (The symbol m indicates the number of servers that an object is striped over; the space-efficiency of the R/W-PF member increases with m.) This corresponds to the Benign, Synchronous, Constant, $t = 2$ R/W-PF member in Chapter 8.

Consider the trade-off in terms of cost: the usable capacity provided by each server, i.e., the space-efficiency of the R/W-PF member. Because three-fold replication is employed by the Replication member, each server provides only 33% of its storage capacity as usable capacity. If the more space-efficient Constant member is employed, then each server provides 75% of its storage capacity as usable capacity. Even though both members tolerate two server failures, each member has a different cost in terms of usable capacity. However, to tolerate two server failures, the Replication member requires 3 servers, whereas the Constant member requires 8 servers. The Constant member increases space-efficiency but reduces reliability.

Consider the performance trade-off in terms of response time and throughput. The experiments from which these performance results are drawn are described in Chapter 8. The performance results are for concurrency- and failure-free operations. The response time for read operations is 0.6 ms for both R/W-PF members. The response time for write operations is 1.55 ms for the Replication member, whereas for the Constant member it is 1.95 ms. See Figure 8.10 in Section 8.3.3 for more details about response time. Given a cluster of twelve servers, the read throughput is 380 MiB/s for the Replication member and is 500 MiB/s for the Constant member. The write throughput is 210 MiB/s for the Replication member and 340 MiB/s for the Constant member. The Replication member is more responsive than the Constant member but provides less throughput.

The Replication and Constant members considered differ only in the space-efficiency of the erasure-code employed. This single decision leads to differences in cost, write response time, read throughput, and write throughput.

Compare these Benign, Synchronous members with a member that tolerates malevolent clients and a single malevolent server in an asynchronous timing model. This corresponds to the Malevolent$^+$, Asynchronous, Default, $t = 1$ R/W-PF member in Chapter 8. Such a member employs an erasure code with $m = 2$ and requires five servers. Thus each server provides 40% of its storage capacity as usable capacity for the Malevolent$^+$ member. The read response time is 1.6 ms and the write response time is 2.35 ms. These response times are higher than the Benign members because of the costs associated with tolerating malevolent components. Given twelve servers, the read throughput for the Malevolent$^+$ member is 455 MiB/s and the write throughput is 190 MiB/s.

The Malevolent member considered differs from the Benign members both in terms of the resiliency provided (types of failures, number of server faults tolerated, and timing model) and in the space-efficiency of the erasure code employed. Again more trade-offs among cost, read and write response time, and read and write throughput exist.

The R/W-PF allows different trade-offs among cost, resiliency, and performance to be made. A storage infrastructure built using the R/W-PF is versatile because it allows different R/W-PF members to be employed and so different trade-offs to be made for different stored objects. A large portion of the "versatility-space" provided by the R/W-PF in the PASIS storage system is explored in Chapter 8—only three of the hundreds of R/W-PF members evaluated are discussed in this section.

2 Background and related work

This chapter describes cluster-based storage infrastructures and discusses trends that demand such infrastructures tolerate a broader range of faults than do today's storage servers. It overviews the operation of members of the R/W-PF, as well as the versatility such members can provide. Finally, prior work related to the techniques employed by the R/W-PF is discussed.

2.1 Cluster-based storage infrastructure

Today's enterprise storage is dominated by large monolithic disk array systems, extensively engineered to provide high reliability and performance in a single system. However, this approach comes with significant expense and introduces scalability problems: any given storage enclosure has an upper bound on how many disks it can support. To reduce costs and provide scalability, many are pursuing cluster-based storage solutions [for example: Anderson et al., 1996; EMC Corp., 2003; EqualLogic Inc., 2003; Frølund et al., 2003; Ganger et al., 2003; Ghemawat et al., 2003; IBM Almaden Research Center, 2003; Lee and Thekkath, 1996]. Cluster-based storage eliminates this upper bound by replacing the single system with a collection of smaller, lower-performance, less-reliable servers (sometimes referred to as *storage bricks*). Data and work are redundantly distributed among the bricks to achieve higher performance and reliability. The argument for the cluster-based approach to storage follows from both the original RAID argument of Patterson et al. [1988] and arguments for cluster computing over monolithic supercomputing. Cluster-based storage has scalability and cost advantages, but most designs lack versatility.

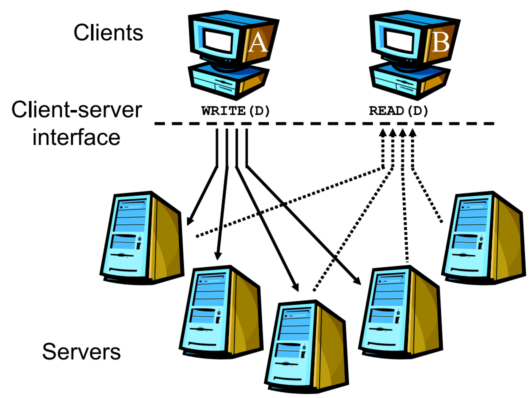


Figure 2.1. Cluster-based storage infrastructure.

Figure 2.1 illustrates, at a high-level, the architecture of a cluster-based storage infrastructure. To write an object D, Client A issues write requests to a set (quorum) of servers. To read D, Client B issues read requests to a set of servers that intersects the set of servers to which D was last written. This architecture can provide access to objects even when some servers have failed. To provide reasonable storage system semantics, a read must observe the latest value of D written. For shared storage systems, this usually means linearizability [Herlihy and Wing, 1990] of read and write operations.

2.2 Beyond a single synchronous fail-stop failure

Today's enterprise storage systems are typically monolithic and very expensive, based on special-purpose, high-availability components with comprehensive internal redundancy. These systems are engineered and tested to tolerate harsh physical conditions and continue operating under almost any circumstance. Cluster-based storage is a promising alternative to today's monolithic storage systems. The concept is that collections of smaller servers should be able to provide performance and reliability competitive with today's solutions, but at much lower cost and with greater scalability. The cost reductions would come from using commodity components for each server and exploiting economies of scale. Each server would provide a small amount of the performance needed, but with lower reliability than required. As with previous arguments for RAID and cluster computing, the case for cluster-based storage anticipates that high levels of reliability and performance can be obtained by appropriate redundancy and workload distribution across servers. If successful, cluster-based storage should be much less expensive (per byte) than today's enterprise storage systems [Frølund et al., 2003], while providing similar levels of reliability and availability.

Corbett et al. [2004] motivate the need for tolerating double disk failures in traditional high-end storage servers because trends conspire to make "whole-disk/whole-disk" and "whole-disk/media" failures more likely. Specifically, disk capacity is increasing at a greater rate than disk bandwidth. As such, the time required to reconstruct data after a disk failure is increasing. As well, increasing capacity increases the likelihood that some disk has a media failure. This is exacerbated by the fact that the cost of *scrubbing* (i.e., validating the integrity of stored data by reading it) increases as capacity increases. Indeed, "whole-disk/media" failures dominate reliability in the analysis. Since even enterprise storage must tolerate more than one disk failure at a time, requirements on cluster-based storage will likely specify that three, or even more, server faults be tolerated.

Beyond tolerating more faulty servers than today's systems, storage infrastructures should tolerate faulty servers that exhibit non-fail-stop behavior. Whereas enterprise storage servers have redundant internal interconnects, cluster-based storage is susceptible to network partitions and servers that crash and recover, independent of any centralized storage controller.

Two trends indicate that Byzantine failures [Lamport et al., 1982]—arbitrary, potentially malicious failures—of components should be tolerated. First, the amount of software involved and the consumer-quality components that are likely to be integrated into cluster-based storage systems lead to the real risk of faulty components taking unspecified actions. For example, we have been told that disks occasionally write data sectors to the wrong location [Kleiman, 2002]. Such a fault is pernicious: a disk drive returns success codes after performing a mis-aligned write, and subsequent reads observe seemingly well-formed data. Second, some cluster-based storage will be deployed in a more open, less well-administered manner. For example, the FARSITE project intends to use desktop computers as a cluster-based storage infrastructure [Adya et al., 2002]. For these reasons, it is important that a storage infrastructure be versatile, so that it can "scale" up the fault model and tolerate Byzantine failures of components.

In a high-speed local-area network context, it may be reasonable to make synchrony assumptions. However, quality of service and isolation of network traffic are still open problems in most LAN settings. Without the guarantees of QoS or isolation, the asynchronous timing model avoids making assumptions that could lead to loss of consistency. As well, if servers are federated in a MAN or WAN setting, it may not be feasible to engineer the storage infrastructure to have synchronous bounds.

2.3 R/W-PF overview

The R/W-PF is designed to provide a versatile cluster-based storage infrastructure. Each member of the R/W-PF works roughly as follows. To perform a write, a client sends requests to a quorum of servers. Each write request includes a logical timestamp and a fragment. The fragment could be an object replica, or it could be an erasure-coded fragment of the object. Servers retain all versions of fragments they are sent.

To perform a read, a client fetches the latest fragment versions from a quorum of servers. The client considers the timestamps of the fragments received and determines whether they comprise a complete write. During concurrency- and failure-free access, they do comprise a complete write. If they do not, historical fragments are fetched until a complete write is observed. Only in certain cases of failures or concurrency are there additional overheads incurred to maintain consistency.

A member of the R/W-PF is distinguished by its resiliency model and the storage mechanisms it employs. The resiliency model is specified in three parts. First, a timing model, either asynchronous or synchronous, is specified. Second, a server failure model is specified. R/W-PF members can tolerate a mix of Byzantine and benign failures. Both the mixture of types of faults tolerated, and the number of each type of fault is specified. Third, a client failure model is specified: either benign or Byzantine.

The storage mechanisms of a R/W-PF member are also specified in three parts. First, the erasure code is specified. The choice of erasure code is constrained by the server failure model since the erasure code must provide sufficient redundancy to tolerate the specified number of server faults. The space-efficiency of the R/W-PF member is determined by the erasure code specified. Second, the quorum construction is specified. Again, the choice is constrained by the server failure model. Additionally, the quorum construction is constrained by the timing model and may also be constrained by the erasure code. Third, witness use is specified—witness use is constrained by the server failure model, timing model, erasure code, and quorum construction for the member.

Servers export the same interface regardless of which R/W-PF member is used to read and write objects to them. This is illustrated in Figure 2.1 by the dashed client-server interface line. To perform reads and writes, clients issue quorum remote procedure calls (quorum RPCs) to sets of servers. There are four quorum RPCs that clients may invoke:

– **c_qprc_read_latest** that reads the latest fragment hosted at each server in a quorum.

– **c_qrpc_read_previous** that reads the latest fragment, prior to some specified logical time, at each server in a quorum.

– **c_qrpc_read_time** that reads the latest timestamp at each server in a quorum.

– **c_qrpc_write** that writes a timestamp–fragment pair to each server in a quorum.

This client-server interface is sufficient to implement R/W-PF members that meet a broad range of resiliency and performance requirements.

The resiliency models and storage mechanisms define the versatility of the R/W-PF. The R/W-PF is able to provide such broad versatility with such a simple interface because the protocol logic is localized to clients. The timing model and server failure model dictate a minimum number of servers that a client must interact with for a given R/W-PF member. The client failure model dictates aspects of how a client must encode and decode an object. The erasure code dictates other aspects of how a client must encode and decode an object. The quorum construction dictates how many (or at least which sets of) servers a client may access to read and write an object. The witness use dictates whether or not a client sends or requests a fragment in its read or write request. Because all of these decisions are localized in client logic, the R/W-PF does not require an extensive server interface.

2.4 Related work

The R/W-PF builds on a large body prior work. This section reviews the most relevant: versatility, consistency protocols, and quorum-based protocols. Additional related work pertinent to the resiliency models and versatile storage mechanisms of the R/W-PF is discussed in Chapter 3.

The R/W-PF grew out of the PASIS project that focused on extremely reliable storage [Wylie et al., 2000]. The original PASIS prototype used erasure codes to store archival data. The initial prototype lead to an interest in understanding the performance, availability, reliability, and security tradeoffs between different *data distribution schemes* (methods of encoding and decoding objects) [Wylie et al., 2001]. From there, the focus moved to understanding if it was possible to leverage versioning servers to build extremely reliable, efficient read/write storage, rather than archival storage. The techniques and methods for efficiently tolerating Byzantine failures for an asynchronous erasure-coded storage system resulted from this effort [Goodson et al., 2004a]. Recently, this work was extended to include *lazy verification*, a mechanism that efficiently tolerates writes of erasure-coded objects by Byzantine faulty clients with bound storage capacity [Abd-El-Malek et al., 2005b].

In our work towards the Byzantine fault-tolerant protocol, we realized that the quorum-based nature of the protocol lent itself to localizing nearly *all* of the resiliency decisions in the client logic. Given this insight, we began exploring the versatility that a family of read/write protocols could provide [Goodson et al., 2003, 2004b; Wylie et al., 2004]. The prototype implementation of the R/W-PF is a key component of the Ursa Minor storage system currently under development [Ganger et al., 2005].

2.4.1 Versatility

Protocol families

We do not have a rigorous definition of the term "protocol family" to which we subject ourselves or others. The R/W-PF allows for great versatility with a simple interface to servers. This is the key characterization of the term "protocol family" for the sake of this dissertation. However, there is prior work on versatile or modular protocols, which this section reviews.

Hadzilacos and Toueg [1994] developed a modular approach to fault-tolerant broadcast. They developed a reliable broadcast primitive that provides different broadcast guarantees (reliable, FIFO, causal, atomic), and does so for various timing models.

Cristian et al. [1995] systematically derive a logical family of atomic broadcast protocols for a range of fault models (from crash faults to a subset of Byzantine faults) in the synchronous timing model. Cristian et al., uses the term "family" to refer to the logical construction of the protocols rather than their implementation. Members of the R/W-PF are realized in a common implementation as well as being logically related. Mishra et al. [2002] extend the work of Cristian et al. to an implementation for the timed asynchronous model [Cristian and Fetzer, 1999]. Specifically, the Timewheel group communication system provides nine distinct semantics (one of three atomicity guarantees paired with one of three order guarantees) on a per broadcast granularity. The Timewheel system does not tolerate Byzantine failures.

Hiltunen et al. [1999] developed a framework for distributed services that are easily configured to handle different failures. The framework is distinguished from Timewheel in that high-level protocols are built out of micro-protocols [Bhatti and Schlichting, 1995] that implement individual semantic properties (e.g., an atomicity guarantee or an ordering guarantee) or mask particular failure types. The framework provides a methodology for building micro-protocols and for composing micro-protocols into high-level protocols, as well as a run-time system for distributed services.

Both Timewheel and the configurable framework provide a modular means of employing distinct protocols. The R/W-PF is distinguished from each of these in that servers have a narrow, fixed client-server interface that supports all members of the protocol family. Additionally, the client implementation is shared across protocol family members.

Some prior work on adaptation in distributed systems has a similar motivation as the R/W-PF. For example, distributed systems have been designed that adapt—that is, change aspects of the protocol being executed—based on observed changes in the execution environment, or on user demand [Hiltunen and Schlichting, 1996; Renesse et al., 1998; Chen et al., 2001]. These systems tend to take a "middleware" approach to the problem, in contrast to the narrow, client-server interface approach of the R/W-PF.

The term "family" has been used as a modifier for other ideas and techniques in the computer science literature, for example, "family of hash functions". We hope no confusion follows from our use of the term "family".

Versatility in cluster-based storage

Petal [Lee and Thekkath, 1996], xFS [Anderson et al., 1996], and NASD [Gibson et al., 1998] are early systems that laid the groundwork for today's cluster-based storage designs. More recent examples include FARSITE [Adya et al., 2002], FAB [Saito et al., 2004], EMC's Centera [EMC Corp., 2003], EqualLogic's PS series product [EqualLogic Inc., 2003], Lustre [Lustre, 2004], Panasas' ActiveScale Storage Cluster [Panasas, Inc., 2005], and the Google file system [Ghemawat et al., 2003]. Each of these systems hard-codes most choices about the manner in which objects are stored. For example, Petal replicates data for fault tolerance, tolerates only server crashes (i.e., fail-stop servers), and uses chained declustering to disperse data and load across nodes in the cluster; these choices apply to all data. xFS also uses one choice for the entire system: parity-based fault tolerance for server crashes and data striping for dispersing load. In contrast, the R/W-PF provides versatility on a per-object basis.

FAB [Saito et al., 2004] and RepStore [Zhang et al., 2004] offer two erasure codes (replication or erasure coding) rather than just one. FAB allows the choice to be made on a per-volume basis at volume creation time. RepStore, which has been designed and simulated, uses AutoRAID-like algorithms to select which to use for which data in an adaptive manner. AutoRAID [Wilkes et al., 1996] automates versatile storage in a monolithic disk array controller. Most disk array controllers allow specialized choices to be made for each volume. AutoRAID goes beyond this by internally and automatically adapting the choice for a data block (between RAID 5 and mirroring) based on usage patterns.

2.4.2 Consistency

Consistent access to cluster-based storage requires a protocol that addresses three sources of problems: access concurrency, servers failures, and client failures (resulting in partial or corrupt updates). Many protocols have been proposed, implemented, and used to address various mixes of these problems. The R/W-PF implements an atomic register [Lamport, 1985], which we refer to as a read/write object (or, just object), that permits multiple readers and multiple writers access.

Most storage systems assume benign crash failures by clients and servers, simplifying the problems significantly. Under such assumptions, access concurrency can be addressed via leases [Gray and Cheriton, 1989], optimistic concurrency control [Kung and Robinson, 1981], or serialization through a primary [for example: Liskov et al., 1991; Lee and Thekkath, 1996]. Partial writes by clients that fail can be addressed by two-phase commit [Gray, 1978] or by post-hoc repair (in systems using replication).

Most systems implementing Byzantine fault-tolerant services adopt the state machine approach [Lamport, 1978; Schneider, 1990] wherein all operations are processed by all server replicas in the same order. There is a long tradition of building increasingly efficient, or specialized, Byzantine fault-tolerant agreement-based systems: the protocol of Bracha and Toueg [1985] was improved upon by Rampart [Reiter, 1995], SecureRing [Kihlstrom et al., 2001], BFT [Castro and Liskov, 2002], SINTRA [Cachin and Poritz, 2002], and FaB [Martin and Alvisi, 2005]. An alternative to agreement-based protocols, that provide similar semantics, are protocols based on Byzantine quorum systems [for example: Malkhi and Reiter, 1998; Malkhi et al., 2001; Abd-El-Malek et al., 2005a]. These approaches linearize arbitrary operations, whereas the R/W-PF only linearizes reads and writes.

The R/W-PF is optimistic and designed to place work on clients rather than servers. The optimistic approach allows most read operations to complete in a single phase of client-server communication. Only reads concurrent to a write or a failure may incur additional phases of communication. Studies of distributed storage systems indicate that concurrency is uncommon [Baker et al., 1991; Noble and Satyanarayanan, 1994]: for example, they indicate that writer-writer and writer-reader sharing occurs in under 1% of operations. Failures, although more common than administrators and users would like, are still uncommon events. The client is responsible for most of the computation costs of a R/W-PF member. This follows the the well-known principle of shifting work from servers to clients to increase scalability [Howard et al., 1988].

Versioning storage

The R/W-PF relies on versioning servers to provide consistency efficiently. We contrast our use of versioning to maintain consistency with systems in which each write creates a new, immutable version of an object [for example: Mullender, 1985; Reed, 1983]. Such systems shift consistency problems to the metadata mechanism that resolves object names to a version. Systems that employ such an approach (e.g., Past [Rowstron and Druschel, 2001] and CFS [Dabek et al., 2001]) require a version tracking service to find the latest version, whereas, the R/W-PF does not. Some systems (e.g., FAR-

SITE [Adya et al., 2002] and OceanStore [Kubiatowicz et al., 2000]) use replicated state machines for such metadata functions. Another option is to layer higher-level services, as needed, atop a base read/write service—for example, Frangipani [Thekkath et al., 1997] provides file services atop Petal [Lee and Thekkath, 1996]. Ivy [Muthitacharoen et al., 2002] provides decentralized read/write access to immutable stored data in a fashion similar to some members of the R/W-PF. Per-client update logs (which are similar to server histories) are merged by clients at read time. Ivy differs from the R/W-PF in that it does not provide strong consistency semantics in the face of data redundancy, concurrent accesses, or failures.

2.4.3 Quorum-based protocols

Herlihy and Tygar [1987] were perhaps the first to apply quorums to the problem of protecting the integrity (and confidentiality) of replicated data against Byzantine faulty servers, as do some R/W-PF members. Their work, however, did not focus on achieving strong data semantics in the face of concurrent access, did not admit the full range of system models of the R/W-PF, and did not include an implementation or performance analysis.

There are many Byzantine fault-tolerant protocols for implementing read-write objects using quorums [for example: Herlihy and Tygar, 1987; Malkhi and Reiter, 1998; Pierce, 2001; Martin et al., 2002]. Of these, the "Listener's" approach of Martin et al. [2002], in that it uses a type of versioning, is closest to members of the R/W-PF that tolerate Byzantine faulty components. In a R/W-PF member, a read may retrieve fragments for several versions of the object in the course of identifying the object value to return. Similarly, a read in the Listener's protocol "listens" for updates (versions) from servers until an object value is observed that can be returned. Conceptually, our approach differs in that clients read past versions, rather than listening for future versions sent by servers.

The Listener's protocol stores only replicated objects not erasure-coded objects. Compared with a similar member of the R/W-PF, the Listener's protocol tolerates a higher fraction of faulty servers. The Listener's protocol tolerates *poisonous* [Martin et al., 2002] writes by Byzantine faulty clients. However, to do so, the Listener's protocol requires server-to-server broadcast: correct servers broadcast values they receive to other servers. The self-verifying nature of replicated objects is taken advantage of, so that servers "reach agreement" on the object value written.

Malkhi and Reiter [2000] use an *echo* phase in a quorum-based protocol to protect against poisonous writes (to ensure writes are *justified* in their terminology). The echo phase ensures that a Byzantine faulty client sends the same object value to each server and relies upon the self-verifying nature of replicated objects to do so. Perhaps the first system to employ such an echo phase was Rampart [Reiter, 1995].

Cachin and Tessaro [2005a,b] implement AVID, which extends the Listener's approach of protecting against poisonous writes to erasure-coded objects. Since fragments of erasure-coded objects are not self-verifying, AVID requires each server to receive sufficient fragments to verify the erasure coding of the written object.

The R/W-PF relies on versioning servers, in conjunction with cross checksums [Gong, 1989] embedded in timestamps, to implement *self-validating timestamps* [Goodson et al., 2004a]. Self-validating timestamps protect erasure-coded objects from poisonous writes. However, the technique permits Byzantine faulty clients to introduce an unbound amount of work for subsequent readers. *Lazy verification* is an extension to the Byzantine fault-tolerant members of the R/W-PF that bounds the number of poisonous writes a single Byzantine faulty client can introduce [Abd-El-Malek et al., 2005b].

3 The Read/Write Protocol Family

We describe R/W-PF membership in terms of *resiliency models* and *storage mechanisms*. Resiliency models range from restrictive assumptions to the near absence of assumptions about the operating environment. Many mechanisms exist to improve the efficiency of distributed protocols in general, and additional mechanisms exist to improve the efficiency of read/write storage access protocols in specific. Storage mechanisms are the set of such mechanisms that are incorporated in the R/W-PF.

3.1 Resiliency models

Broadly speaking, resiliency models specify the fault-tolerance of a R/W-PF member. Table 3.1 summarizes the specification of the resiliency model. To specify a resiliency model, one item from each row is selected: a timing model, a server failure model, and a client failure model. As discussed in Section 3.1.2, network failures are not modeled directly because most such failures can be mapped to a server failure model.

3.1.1 Timing models

Two timing models may be specified in the R/W-PF:

- *Asynchronous*, in which no timeliness assumptions are made.
- *Synchronous*, in which known bounds on delays in the system are assumed.

Model	Specification
Timing model	Asynchronous or synchronous with timeout δ.
Server failure model	Benign server failure model type (crash, omission, or crash-recovery), $\mathcal{T}$ all sets of faulty servers to tolerate, and $\mathcal{B}$ all sets of malevolent servers to tolerate.
Client failure model	Either crash or Byzantine failures.

Table 3.1. Specification of resiliency model summary.

With asynchronous members no assumptions about message transmission delays or execution rates are made, except that they are finite and non-zero. In contrast, with synchronous members, known bounds are assumed for message transmission delays between correct clients and servers, as well as the execution rates of clients and servers. To specify a synchronous timing model the bound on delays is specified. It is sufficient to specify a single bound δ on the total delay. This bound accounts for the delay in a client sending a request to a server, the delay in that server processing the request, and the delay in sending that server's response back to the client.

Comments

The timed asynchronous model of [Cristian and Fetzer, 1999] and the partially synchronous model of [Dwork et al., 1988] are not directly included in the R/W-PF. The timed asynchronous model allows components to have local clocks that "proceed within a linear envelope of real-time", restricts channel failures to be omission faults or performance faults, and requires services to meet timeliness guarantees. Such a model is appealing because the FLP impossibility result [Fischer et al., 1985] can be circumvented allowing many difficult problems to be solved with a deterministic protocol that terminates. Since the R/W-PF simply implements an atomic read/write register, the FLP impossibility result does not apply. The restrictions on failure types and the focus on timeliness guarantees fit poorly with R/W-PF members that tolerate malevolent components. Providing a quality of service guarantee (i.e., timeliness) in the face of malevolent components that may attempt denial of service attacks is an open problem. Thus, we focus on the safety and liveness of reads and writes in R/W-PF members but not on the timeliness of such operations.

In the partially synchronous model, periods of synchronous operation are assumed to occur, but they do so at unspecified times or with unknown bounds on the delay. Although the R/W-PF does not allow a "partially synchronous" timing model to be specified, a combination of synchronous timing model and crash-recovery servers (discussed below) very closely follows one definition of a partially synchronous system. In such R/W-PF members, the bound on delay is specified, but the time at which the bound becomes true is not. During periods of time when the bound is true, progress can be made.

Often, distributed systems take advantage of loosely synchronized clocks to improve performance [Liskov, 1991]. We do not explicitly include loosely synchronized clocks in the timing model. However, we do consider them in the prototype implementation of the R/W-PF (see Section 7.9).

3.1.2 Server failure models

A broad range of server failure models may be specified in the R/W-PF:

- *Crash failures*, in which some servers crash and never respond to requests again.

- *Omission failures*, in which some servers simply drop some received requests or fail to send some responses [Perry and Toueg, 1986].

- *Crash-recovery failures*, in which all servers may crash and recover, but there is a time after which some set of servers are guaranteed to be up and to not crash again [Aguilera et al., 2000].

- *Byzantine failures*, in which some servers may exhibit arbitrary behavior [Lamport et al., 1982].

In the synchronous timing model, crash failures can be reduced to fail-stop failures, in which servers detectably crash [Schlichting and Schneider, 1983; Schneider, 1984]. Omission failures are a strict generalization of crash failures. For example, a server may receive and send messages to only a subset of clients as if the network were partitioned: to the excluded clients, such an omission failure is indistinguishable from a crash failure. The crash-recovery failure model is a strict generalization of the omission failure model. The Byzantine failure model is a strict generalization of the crash-recovery failure model. Servers that are Byzantine faulty may act maliciously; they may manipulate stored data, send different replies to similar requests from distinct clients, and so on. Since the Byzantine failure model is a strict generalization of the crash-recovery failure model, another term — *malevolent* — is used to categorize those servers that in fact exhibit out-of-specification, non-crash behavior.

Aguilera et al. [2000] uses the following terminology to describe servers (processes) in the crash-recovery model: each server can be classified as always-up, eventually-up, eventually-down, or unstable. A server that is *always-up* never crashes. A server that is *eventually-up* crashes at least once, but there is a time after which it is permanently up. A server that is *eventually-down* crashes at least once, and there is a time after which it is permanently down. A server that is *unstable* crashes and recovers infinitely many times.

We extend the crash-recovery terminology as follows. A server is *good* if it is always-up or eventually-up. A server is *faulty* if it is eventually-down, unstable, or malevolent. Each server is either good or faulty. A server is *benign* if it is good, eventually-down, or unstable. Each server is either benign or malevolent. Since the crash-recovery failure model is a strict generalization of the omission and crash failure models, these terms are well defined regardless of which failure model is specified.

In the interest of allowing the richest set of possible server failure models, the server failure model for the R/W-PF membership is a hybrid failure model [Meyer and Pradhan, 1991; Thambidurai and Park, 1988]. To specify a hybrid failure model, sets of faulty servers are specified, subsets of which may be malevolent. We extend the definition of a fail prone system given by Malkhi and Reiter [1998] to accommodate the hybrid server failure model.

We assume a universe U of servers such that $|U| = n$. The system is characterized by two sets: the faulty sets $\mathcal{T} \subseteq 2^U$ and the malevolent sets $\mathcal{B} \subseteq 2^U$. (The notation 2^{set} denotes the power set of *set*). In any execution, all faulty servers are included in some $T \in \mathcal{T}$ and all malevolent servers are included in some $B \in \mathcal{B}$. It follows from the definitions of faulty and malevolent that, in any given execution, $B \subseteq T$.

The *benign server failure models* are the crash, omission, and crash-recovery failure models. To specify a hybrid failure model, a specific benign server failure model is selected and then the failure sets $\mathcal{T}$ and $\mathcal{B}$ are specified. If only benign server failures are tolerated, then $\mathcal{B} = \emptyset$. If all faulty servers may be malevolent, then $\mathcal{T} = \mathcal{B}$.

Comments

As stated above, if the crash-recovery failure model is specified, all servers may crash and recover. Clearly, if too many servers are crashed simultaneously, progress is not possible. However, once sufficient servers recover, progress is again possible.

As stated above, $\mathcal{T}$ is the set of sets of faulty servers that are tolerated. For a given execution, $T \in \mathcal{T}$ is the set of faulty servers. With the crash-recovery failure model for servers there may be periods of an execution during which servers not in T are crashed. So long as sufficient servers are up, potentially some in T, it is possible to make progress. In practice, the crash-recovery failure model is more general (useful) than the restriction "there being a time after which good servers are up" may suggest. During any period in which the set of crashed servers is in $\mathcal{T}$, progress is possible.

In the server failure model, send omission failures [Perry and Toueg, 1986] are not distinguished from receive omission failures [Hadzilacos, 1984]. The omission failure model is a general omission failure model [Perry and Toueg, 1986] that tolerates both send and receive omission failures.

In some failure specifications, the type and number of link failures is distinguished from the type and number of process (server) failures; we do not make such a distinction. For a given system installation, consideration of network topology and the reliability of shared network hardware (e.g., switches and routers) could inform the specification of fault sets. (Alternately, the specification of fault sets could inform network topology and physical layout.) Regardless, little is gained by directly incorporating link failure models in the resiliency models: such failures can be mapped to omission and crash-recovery failures.

There is a long history in distributed systems of considering servers that crash and recover [for example, Lampson et al., 1981]. Moreover, there are many recent formalisms [for example: Hurfin et al., 1998; Boichat and Guerraoui, 2000] other than that of Aguilera et al. [2000] for modeling servers that crash and recover. The hybrid model used by Backes and Cachin [2003] in the development of a reliable broadcast primitive tolerates Byzantine failures and processes that crash and recover.

Some failure classifications distinguish *timing faults* from *value faults*. The benign server failure models are examples of timing faults. Malevolent servers exhibit value faults. "Benign" value faults—for example, "bit-flips" during transmission or in memory—are not modeled separately from malevolent faults in the server failure models. The rationale is that components can readily be hardened against such failures independently by employing mechanisms such as checksums in the transport protocol and ECC memory.

3.1.3 Client failure models

Two client failure models may be specified in the R/W-PF:

- *Crash failures*, in which clients may crash.
- *Byzantine failures*, in which clients may exhibit arbitrary and malicious behavior.

We refer to clients that exhibit out-of-specification, non-crash failures as *malevolent*. We refer to clients that are correct or that follow their specification except for crashing as *benign*. There is no bound on the number of clients that may fail.

Comments

Fewer client failure models may be specified than server failure models. No useful versatility is gained by specifying omission, or crash-recovery failure models for clients in the R/W-PF.

An authorized malevolent client can write arbitrary values to the object. A simple data abstraction such as an atomic read/write register cannot enforce restrictions on the stored value. R/W-PF members that tolerate Byzantine failures of clients guarantee that such clients write a single value to the object consistently.

In an asynchronous system, readers cannot distinguish read-write concurrency from a client crash failure during a write operation. In a synchronous system in which only benign component failures are tolerated, readers can distinguish read-write concurrency from a crash failure during a write operation (by issuing multiple read requests separated in time by δ). The ability to distinguish crash clients from concurrent clients provides no obvious performance benefits in the R/W-PF. Even though there is the potential to take advantage of such information in benign synchronous systems, we do not.

3.2 Storage mechanisms

Each member of the R/W-PF specifies a set of storage mechanisms to employ when reading and writing objects. In some cases, these mechanisms enhance the versatility of the R/W-PF, in others these mechanisms simply improve its efficiency. Whereas the range of interesting and useful resiliency models is clear given the extensive distributed systems and fault-tolerance literature, a near endless list of mechanisms and techniques have been proposed in the context of distributed protocols. Limiting the scope to read/write objects, we have identified a set of storage mechanisms that integrate nicely in the R/W-PF:

- *Quorum system*, which dictates the sets of servers that a client can interact with to read and write an object.

– *Encoding*, which dictates how objects are encoded by a client on a write and decoded by a client on a read.

– *Witnesses*, which dictates which sets of servers must store data and metadata pertaining to an object and which sets of servers need only store metadata.

3.2.1 Quorum system

We extend the definition of a quorum system given by Malkhi and Reiter [1998] to facilitate the discussion of objects stored in different quorum systems on a common set of servers. We assume a universe $\mathcal{U}$ of servers. A quorum system employed by a R/W-PF member is based on a sub-universe $U \subseteq \mathcal{U}$, such that $|U| = n$. A *quorum system* $\mathcal{Q} \subseteq 2^U$ is a non-empty set of subsets of U, every pair of which intersect. Each $Q \in \mathcal{Q}$ is called a *quorum*. The rationale for defining quorum systems in terms of sub-universes is to allow different R/W-PF members to employ different quorum systems comprised of potentially disjoint sets of servers. This is desirable in the context of cluster-based storage.

To specify the quorum system for an R/W-PF member, either the quorum construction method must be specified, or a specific quorum system $\mathcal{Q}$ must be specified. The failure models, specifically $\mathcal{T}$ and $\mathcal{B}$, place restrictions on the construction of the quorum system. The timing model places additional restrictions on the construction of the quorum system. We develop constraints on quorum system construction, given the server failure model and timing model, in Section 5.1.

Comments

Quorum systems are a generalization of majority voting systems [Gifford, 1979; Thomas, 1979]. The construction of a quorum system affects its load and availability [Naor and Wool, 1998; Malkhi et al., 2000]. In the context of quorum systems, load has a specific meaning: *load* is the fraction of time the busiest server (process) is used, given some quorum access strategy. Although quorum constructions with ideal load and availability have been identified [Naor and Wool, 1998; Malkhi et al., 2000], quorum systems must be very large before the asymptotic nature of the analysis holds. This is especially true when tolerating multiple faults and/or Byzantine server failures.

In some sense, quorum constructions implicitly define additional sets of faulty servers that can be tolerated. For example, for some $Q \in \mathcal{Q}$, all of the servers in $U \setminus Q$ may crash (i.e., $T \subseteq U \setminus Q$). Malkhi et al. [2000] consider these faulty sets in their evaluation of the crash probability of different quorum constructions that tolerate malevolent failure sets. However, they do not explicitly construct quorums that tolerate a hybrid server failure model.

Beyond picking a quorum to access initially, the strategy also specifies how to *probe* for quorums [Peleg and Wool, 1996; Hassin and Peleg, 2001]. We do not explicitly list the quorum access strategy and probing policy in the quorum model, but these too could differentiate R/W-PF members.

From a "system's perspective", quorum construction provides different throughput–availability trade-offs. Since *load* is an overloaded term in systems, we refer to the *throughput-scalability* of quorum constructions. The experience of database practitioners suggests that it may be difficult to take advantage of quorum-based throughput-scalability. For example, Jiménez-Peris et al. [2003] conclude that a write-all read-one approach is better for a large range of database applications than a quorum-based approach. Their analysis ignores concurrency control and is based on two phase commit-based data replication with fail-stop failures in a synchronous model. In response to some criticisms of quorum-based approaches, Wool argues that quorum-based approaches are well-suited to large scale distributed systems that tolerate Byzantine failures [Wool, 1998]. Recent work (pardon the self-citation) demonstrates that quorum-based protocols can be *fault-scalable* [Abd-El-Malek et al., 2005a]: throughput of quorum-based protocols does not degrade significantly as the number of faults tolerated increases.

Operation types (e.g., read vs. write) are not considered in the theoretical evaluations of quorum constructions. For the case of read-write storage, a write-all policy (i.e., send to all servers in U) merits consideration, especially if reliability is a priority.

The nature of read/write storage lends itself to partitioning: different objects can be stored in different quorum systems. Quorum constructions for throughput-scalability are interesting only in the context of individual objects. Whereas, for cluster-based storage systems, throughput-scalability by distributing objects over a larger universe of servers is quite efficient.

The quorum-based nature of the R/W-PF localizes protocol family logic to the client, simplifying the construction of the R/W-PF. Read-write storage ought to tolerate many failures (some not benign), and so the fault-scalability of quorum-based approaches is important to the R/W-PF. Finally, the R/W-PF gets some throughput-scalability "for free" simply by employing quorum constructions that tolerate many (potentially malevolent) failures.

There are *probabilistic* quorum constructions that, in theory, provide better throughput-scalability than traditional quorum constructions [For example: Malkhi et al., 1997; Yu, 2004]. However, such constructions necessarily reduce the consistency guarantee that corresponding quorum-based protocols can achieve. There are also quorum constructions that adapt quorum size, within some range, in response to the known set of failures [Alvisi et al., 2000; Martin and Alvisi, 2004]. Such approaches assume a mechanism to detect server failures.

3.2.2 Encoding

A client *encodes* an object during a write and *decodes* an object during a read. Of primary concern for the R/W-PF is encoding objects redundantly so that they can be written to sets of servers. The fault-tolerant nature of the R/W-PF requires that objects can be decoded given responses from only a subset of servers to which they were written. As such, objects must be encoded in an erasure-tolerant manner. For encode, we refer to an object being *erasure-coded* into *fragments*.

For every quorum $Q \in \mathcal{Q}$, there are a set of *decodable sets*. Every decodable set $M \in \mathcal{M}(Q)$ is decodable so long as all fragments from M are correct. The primary encoding mechanism in the R/W-PF is an m-of-n

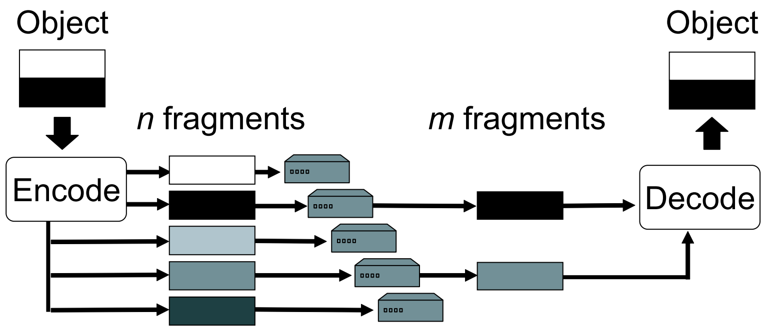


Figure 3.1. An example of an object being erasure-coded with an m-of-n threshold erasure code. In this example, $m = 2$ and $n = 5$. To illustrate stripe-fragments, the object is shown as having two halves (since $m = 2$). Any m of the n fragments can be used to decode the object.

threshold erasure code which encodes an object into n fragments such that any m is sufficient to decode. Note that $m \leq n$. For m-of-n threshold erasure codes, $\mathcal{M}(Q) = \{\forall M \subseteq 2^Q : |M| = m\}$. Figure 3.1 illustrates an object being erasure-coded. That is, correct fragments from any m servers in any quorum are sufficient to decode the object.

There are many specialized threshold erasure codes. For example, replication, Reed-Solomon codes [Berlekamp, 1968], Shamir's secret sharing [Shamir, 1979], ramp schemes [Blakley and Meadows, 1985], RAID 3/4/5/6 [Patterson et al., 1988], Rabin's information dispersal (IDA) [Rabin, 1989], and Krawczyk's short secret sharing [Krawczyk, 1994] are all threshold erasure codes. Different erasure codes provide different space-efficiencies. We define *blowup* as the ratio of the total size of all fragments to the object size[1]. The space-efficiency of an erasure code determines the network bandwidth required to read and write, as well as the disk capacity required to store encoded objects. Different erasure codes require different amounts of client computation to encode and decode. For example, replication requires effectively no client computation, whereas all other threshold erasure codes

[1]The *rate* of an erasure or error correcting code is a commonly used metric of its (space-)efficiency. Blowup is the inverse of the rate. Blowup is positively correlated with the increase in response time measured for R/W-PF members with poor space-efficiency and thus more useful for understanding the empirical results in this dissertation.

Erasure code	(m, n)	Description
Replication	$(1, n)$	Each fragment is an identical replica of the original object.
RAID 4	$(n-1, n)$	A single *parity* fragment is calculated: the object is "striped" over m fragments, all of which are XORed together to produce the parity fragment.
IDA	(m, n)	Rabin's information dispersal: the object is "striped" over m fragments and $n-m$ redundant fragments are calculated.

Table 3.2. Specification of encoding.

require client computation to calculate redundant fragments. Table 3.2 lists the threshold erasure codes that can currently be specified for an R/W-PF member.

Comments on threshold erasure codes

We use RAID 4, which constructs parity at the fragment level, in the R/W-PF rather than RAID 3, which constructs parity at the code-word level, because RAID 4 is a *systematic code*. In a systematic code, some m of the fragments are simply portions of a replica of the encoded object. For example, the "first" stripe-fragment consists of the first $\frac{1}{m}^{th}$ of the object. As such, it is possible, by reading the appropriate m fragments, to decode an object encoded with a systematic code without performing any computation.

RAID 5 specifies that parity fragments for different encoded objects are *rotated* among servers. Such a *layout policy* allows systematic reads and RAID read-modify-writes to be dispersed among all servers. Since we consider layout an orthogonal systems issue to redundancy, we include RAID 4 rather than RAID 5 in the set of erasure codes. However, a simple layout policy that rotates parity is incorporated in the implementation. Such a layout policy coupled with the RAID 4 encoding implements RAID 5. Layout policies are described in Section 7.5.

RAID 6 tolerates two erasures, whereas RAID 3/4/5 each tolerate one erasure. Whereas other RAID levels mean (roughly) the same thing to every storage engineer, RAID 6 is an enigma. So long as a scheme tolerates two erasures and does not require too much computation—ideally only performing XOR operations—then it may be "RAID 6". Regardless, all "RAID 6" schemes of which we are aware intermingle fragment layout and redundancy calculation. As such, we do not consider RAID 6. The general class of schemes that specify both layout and redundancy so that only XOR operations are required are called *array codes* [Blaum, 1987]. EVENODD [Blaum et al., 1995] and Row-Diagonal Parity [Corbett et al., 2004] are examples of array codes that tolerate two erasures. Hafner et al. [2004] survey many array codes that tolerate two erasures.

The means of generating redundancy for IDA is not explicitly specified as part of the encoding model. In Section 7.11, we describe the implementation of IDA in the prototype. Different implementations result in different compute costs depending on m.

Comments on other erasure codes

Recently, low-density parity-check (LDPC) codes [Gallager, 1963; Luby et al., 2001] have received much attention. LDPC codes compute redundant fragments via XOR of subsets of "stripes" of the object. LDPC codes trade-off being *perfect*—being able to decode given exactly as many bytes as are in the object—for reduced encode/decode computation. LDPC codes are probabilistic in nature and are best-suited to applications involving thousands of fragments. Digital Fountain [Byers et al., 1998] applied LDPC codes to multicast streams. Plank and Thomason [2004] applied LDPC codes to storage for grid-based computing. In my opinion, the probabilistic nature of the number of fragments required to decode does not lend LDPC codes to "small" storage systems of tens or even hundreds of servers. A variant of LDPC codes, rateless erasure codes [Luby, 2002], has also received some attention. Rateless erasure codes do not require the number of redundant fragments, or the exact method of constructing such fragments, to be specified *a priori*. Again, the major application is in bulk network transfer rather than fault-tolerant storage.

In addition to tolerating erasures, some threshold erasure codes provide confidentiality guarantees. For example, Shamir's secret sharing as well as ramp schemes provide information-theoretic confidentiality, whereas Krawczyk's short secret sharing provides cryptographic confidentiality. Another option for confidentiality, given some channel for managing cryptographic keys among clients, is to encrypt an object prior to erasure-coding the object.

Often, secret sharing schemes are paired with *access structures* [for example, Blundo et al., 1996]. Access structures allow specific sets of fragments to recover specific objects (secrets). Access structures can be tailored to a specific quorum system. Exploring the utility of access structures in cluster-based storage, in conjunction with access control may be interesting, however, it is outside the scope of the R/W-PF. Our focus on threshold erasure codes is due to the simplicity of integrating such erasure codes with threshold quorum constructions.

Comments on the integrity of erasure codes

As suggested by the name, erasure codes tolerate erasures, not errors: malevolent servers can corrupt fragments. If corrupt fragments are used to decode an object, then the decoded "object" is incorrect. Error correcting codes could be employed to protect against corrupt fragments, however error correcting codes are more compute intensive and less space-efficient than erasure codes. Tompa and Woll [1988] proposed an information-theoretic construction for detecting corrupt fragments in Shamir's secret sharing. However, it is even more computationally expensive and less space-efficient then error correcting codes (but it maintains confidentiality).

Gong [1989] developed *cross checksums* to detect corrupt erasure-coded fragments. A cryptographic digest of each fragment is computed, and the set of n digests are concatenated to form the *cross checksum* of the object. The cross checksum is stored with each fragment. Since the cross checksum is essentially replicated, comparing cross checksums across fragments during decode is sufficient to determine the correct cross checksum. The correct cross checksum is then used to validate the integrity of fragments before they are used to decode the object.

Krawczyk [1993] improved the space-efficiency of cross checksums by applying error correcting codes to the cross checksum; Krawczyk referred to this technique as *distributed fingerprints*. Alon et al. [2002] applied aspects of LDPC codes and error correcting codes to protect against malevolent servers in very large systems [cf., addendum Alon et al., 2004]. Krohn et al. [2004] used homomorphic hashes to detect servers sending corrupt fragments during the read and decode process for a rateless erasure code.

Malevolent clients can write arbitrary fragments to servers. For example, malevolent clients could write different object "replicas" to different servers. The value a client subsequently reads then depends on which servers the client contacts. This is unacceptable for a storage system. The value returned by a read operation can not depend on which servers reply. Martin et al. [2002] termed such an attack a *poisonous write*. Since replicated objects are *self-verifying*, a collision-resistant cryptographic hash of the object can protect against poisonous writes. For example, such a hash could be embedded in the name of the object [for example, Mazieres et al., 1999] or agreed upon by servers prior to accepting object replicas [for example, Martin et al., 2002].

Poisonous writes of erasure-coded objects are more difficult to detect and tolerate than poisonous writes of replicated objects. For example, a malevolent client can generate random fragments such that any m fragments decode to a different value. Such a poisonous write results in $\binom{n}{m}$ values being written simultaneously. Protecting against poisonous writes of erasure-coded objects could be cast as verifiable secret sharing [for example, Chor et al., 1985; Feldman, 1987; Pedersen, 1991]. However, such techniques incur unnecessary computational and space costs for encoded objects that are not confidential. Cachin and Tessaro developed verifiable information dispersal based on verifiable secret sharing techniques [Cachin and Tessaro, 2005a,b]. Such techniques require a verification phase in which servers exchange sufficient fragments to determine if all servers received a fragment consistent with a non-poisonous write. Unfortunately, the space-efficiency of verifiable information dispersal on the network is the same as replication.

We developed self-validating timestamps to protect erasure-coded objects from poisonous writes [Goodson et al., 2004a]. Self-validating timestamps combine aspects of cross checksums with self-verifying data, and rely on versioning servers to retain the latest non-poisonous version of the object. Verification of self-validating timestamps is performed at read-time by clients during decode. Section 4.8 describes self-validating timestamps in detail. Servers can lazily verify self-validating timestamps during idle time prior to a client performing a read or on demand when a read request is received [Abd-El-Malek et al., 2005b].

3.2.3 Witnesses

Figure 3.2 illustrates witnesses in the R/W-PF. We use the term *write witness* to denote a client write request that includes only a timestamp to a server, rather than a timestamp and an erasure-coded fragment (i.e., a candidate). We use the term *read witness* to denote a server read response that includes only a timestamp, rather than a timestamp and an erasure-coded fragment. A server that hosts a timestamp and an erasure-coded fragment can reply with either a candidate or a read witness. Whereas a server that is sent a write witness can only reply to read requests with a read witness. The number of write requests and read responses that can make use of write and read witnesses is limited by the timing model, server failure model, quorum system, and threshold erasure code.

Comments

Witnesses are used in replica voting protocols to avoid sending or receiving entire replicas to/from every server [Pâris, 1986]. For example, only version numbers or replica digests need be exchanged. Witnesses can reduce the network and storage bandwidth required for replica-based protocols. Depending on the context in which they have been used, witnesses have also

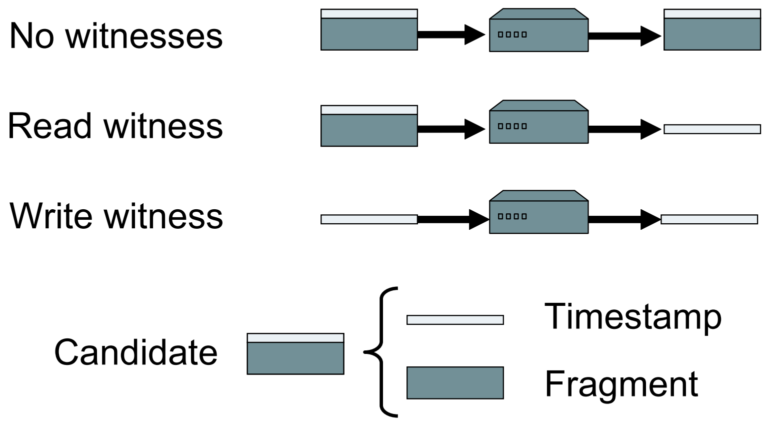


Figure 3.2. An illustration of witnesses. For a read witness, a server simply returns the timestamp of a candidate. For a write witness, a client only sends a server the timestamp of the candidate.

been called *bystanders* [Pâris, 1989], *ghosts* [van Renesse and Tanenbaum, 1988], and likely other names as well.

In the R/W-PF, self-validating timestamps [Goodson et al., 2004a] are used as witnesses. These serve as witnesses for erasure-coded fragments even in the face of malevolent components.

4 Design

4.1 Overview

At a high level, R/W-PF members proceed as follows. To perform a write operation on an object, a client first constructs a logical timestamp. Logical timestamps are used to totally order all write operations. Logical timestamps also identify erasure-coded fragments pertaining to the same write operation across a set of servers. To construct a logical timestamp, a client queries a quorum of servers for the latest candidate timestamp they host. The client identifies the highest timestamp in the quorum of responses it receives. It then constructs a higher timestamp. The client then sends each server in a quorum a write request. The write request includes the new timestamp and the erasure-coded fragment of the object. Together, the timestamp and fragment pair is called a *candidate*.

To perform a read operation on an object, a client issues read-latest requests to a quorum of servers. Once a quorum of servers replies, the client identifies the candidate with the highest timestamp in the response set. The set of server responses that share the same timestamp as the identified candidate are the *candidate set*. The client performs *classification* on the candidate set. Classification determines whether the candidate is complete, repairable, or incomplete.

If the candidate is classified complete, the fragments are decoded and *validated*. Validation protects against malevolent components. If validation is successful, the object is decoded and returned. Otherwise, the candidate is reclassified as incomplete.

If the candidate is classified repairable, it is validated. If validation is successful, the candidate is repaired. To repair a candidate, the client performing the read issues write requests so that a quorum of servers host the candidate. Once a quorum of servers host the candidate, the decoded object is returned. Otherwise, the candidate is reclassified as incomplete.

If the candidate is classified incomplete (or reclassified as incomplete), the candidate is discarded by the client. The client issues read-previous requests to a quorum of servers, requesting candidates with a lower timestamp than that of the discarded candidate. Once the client has another response set, it identifies a new candidate and classification begins anew. So that clients can traverse as many candidates as necessary to find a complete candidate, servers retain all candidates they ever accept.

4.1.1 High-level examples

This section presents a series of high-level examples to illustrate the actions a client takes in the course of read and write operations. Figure 4.1 shows the setup for these examples. This setup is based on the smallest threshold quorum construction for an asynchronous R/W-PF member that tolerates a single malevolent server (i.e., $n = 5, q = 4, r = 2, t = b = 1, m = 2$). These simple examples elide the details of tolerating such malevolent components.

Figure 4.2 illustrates a write operation. First, the client issues read time requests to a quorum of servers. Second, each server responds with the timestamp of the latest candidate in its history. The client constructs a response set and identifies "1" as the latest timestamp. It constructs a higher timestamp, "2". Third, the client sends write requests to a quorum of servers baring the timestamp "2". The details of sending erasure-coded fragments in each write request are elided. Each of the servers that receives a write request accepts the candidate sent by the client. Each server adds the accepted candidate to its history. At this point, the candidate is established. Fourth, the servers reply to the client. Once the client receives responses from a quorum of servers, it knows that its write operation is complete, and it returns.

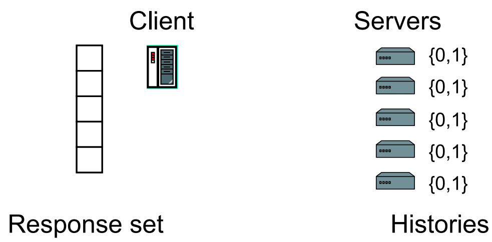
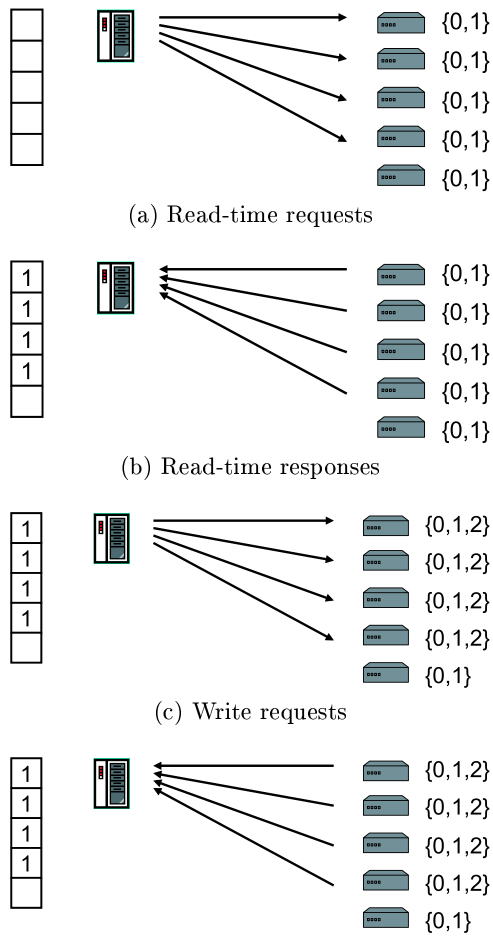


Figure 4.1. Setup for a series of examples for an R/W-PF member. A client on the left with an empty response set. Five servers on the right with their initial histories. Only timestamps of candidates are listed in each server's history.

Figure 4.2 effectively illustrates a simple read operation as well. The steps taken by a client to read a timestamp in Figures 4.2a and 4.2b are the same as in a read operation that issues read-latest requests rather than read time requests. The response set in Figure 4.2b is identical to the candidate set, and the candidate with timestamp "1" is classified complete. The client decodes the object corresponding to timestamp "1" and returns. For read operations that are not concurrent to write operations, a read operation that returns after a single round of client-server communication is the expected common case.

Figure 4.3 illustrates a read operation that is concurrent to a write operation. Notice the state of the server histories in Figure 4.3a—only the first two servers have accepted write requests for the candidate with timestamp "2". First, a client issues read-latest requests to a quorum of servers. Second, once the client receives a quorum of responses, it identifies the latest candidate. The candidate with timestamp "2" is the latest candidate. The candidate set has only two members. In this illustrative example $r = 2$ and $m = 2$ so the client classifies candidate "2" repairable. Third, the client generates fragments for those servers that do not have "2" in their histories, and then sends repair (write) requests to those servers. The third and fourth servers accept the write requests. Fourth, the client receives responses to its repair requests. The client classifies candidate "2" as complete and returns the decoded object.



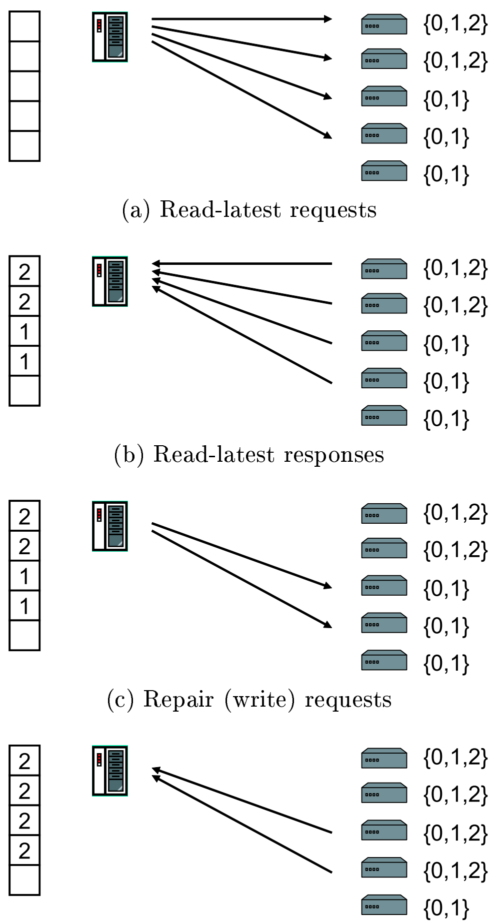
(a) Read-time requests

(b) Read-time responses

(c) Write requests

(d) Write responses

Figure 4.2. Example write operation.



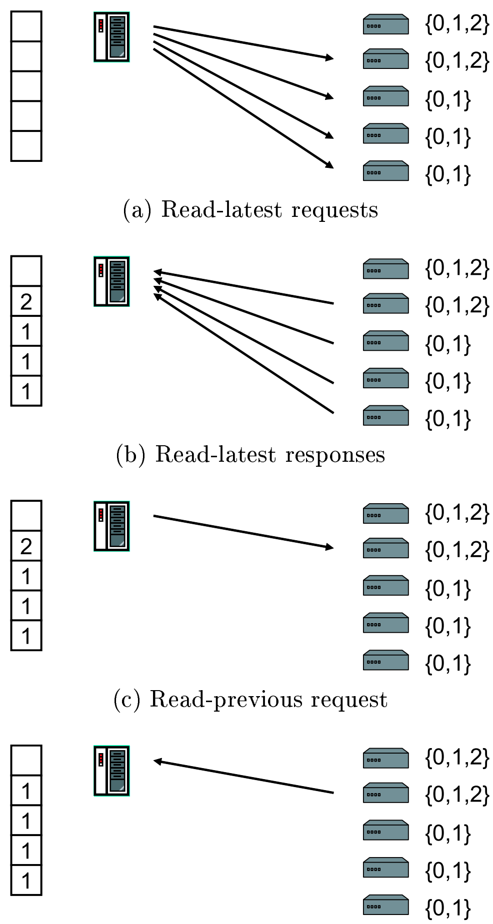
(a) Read-latest requests

(b) Read-latest responses

(c) Repair (write) requests

(d) Repair (write) responses

Figure 4.3. Example read operation that requires repair.



(a) Read-latest requests

(b) Read-latest responses

(c) Read-previous request

(d) Read-previous response

Figure 4.4. Example read operation that requires reading previous.

Figure 4.4 illustrates a different read operation that is concurrent to a write operation. This example is similar to the prior, except that the client accesses a different quorum of servers. After the client receives responses to its read-latest requests, it identifies candidate "2" as the latest. However, as Figure 4.4b shows, the candidate set has only one member. As such, the client classifies "2" incomplete. The client issues a read-previous request, requesting a candidate with an earlier timestamp than "2". After the client receives a response, it identifies a new latest candidate "1". Given the candidate set, the client classifies candidate "1" complete. The client returns the decoded object.

4.2 System model

The protocol family concept applies to systems comprised of many clients and many servers in which clients interact with sets of servers. Clients do not communicate directly with other clients and servers do not communicate directly with other servers. We assume all channels are authenticated: clients can verify server responses and servers can verify client requests. With regard to access control, clients are authorized or not, and if authorized may read or write the object.

We assume point-to-point authenticated channels with properties similar to those defined by Aguilera et al. [2000]: channels do not create messages (*no creation*), channels may experience *finite duplication*, and channels are *fair loss*. The finite duplication property ensures that if benign process p_1 sends a message to benign process p_2 only a finite number of times, then p_2 receives the message only a finite number of times. The fair loss property ensures that if benign process p_1 sends infinitely many messages to good process p_2, then p_2 receives infinitely many messages from p_1. For R/W-PF members with synchronous timing models, we further assume timeliness: repeatedly sending requests over such channel ensures that a correct server will respond within δ delay of the first request.

We assume that servers have stable storage that persists throughout the crash and recover process. Benign servers retain all requests ever accepted in persistent storage.

We assume that clients and servers are computational bound such that cryptographic primitives are effective. Specifically, we assume the existence of collision-resistant hash functions.

4.3 Notation and data structures

We present the design of the R/W-PF for an individual object (i.e., atomic read/write register). The implementation details regarding storing many objects are discussed in Chapter 7

The notation used in the pseudo-code is listed in Table 4.1. Both clients and servers make use of the well-known zero time **0** and the null value $\perp$.

Constants, data types, enumerations, and data structures used throughout the pseudo-code are defined in Figure 4.5. As used in the pseudo-code, the "flags" ASYNCHRONOUS_FLAG, MALEVOLENT_CLIENTS_FLAG, and MALEVOLENT_SERVERS_FLAG are effectively "global constants" for a given R/W-PF member. In practice, an R/W-PF specification is passed from clients to servers with each request.

The equality (=) and less than (<) operators are well-defined for timestamps. Less than is based on comparing *LogicalTime*, *ClientID*, and then *CrossChecksum* (lexicographic comparison).

4.4 Encoding

Figure 4.6 lists functions used to encode and decode objects. Server *Server* is implicitly mapped to some unique identifier in the range 1 through n.

The functions **decode** and **generate_fragments** rely on the deterministic nature of the threshold erasure code. The function **decode** is for a systematic erasure code in which the first m fragments are stripe-fragments (see line 404). The pseudo-code for function **decode** is inefficient. It calls the function **generate_fragments** (see line 403), which generates n fragments, even though only m are needed by **decode**. In fact, because of the system-

Notation	Description
VariableName	Denotes variable name.
function_name	Denotes function name.
CONSTANT_NAME	Denotes constant name.
$\langle a, b\rangle$	Angle brackets denote tuples (e.g., the pair a and b).
$\{a, b\}$	Curly braces denote a set (e.g., a set with two elements: a and b).
(a)	Parentheses are occasionally used to denote a singleton set (i.e., a set with a single element).
$a.b$	The 'dot' operator denotes members of a structure (e.g., element b of variable a); it may be applied to every member of a set.
$a \mid b$	The pipe operator denotes concatenation of a and b.
$\|a\|$	Magnitude of a; context distinguishes magnitude bars from concatentation.
$a = b$	The equality operator (e.g., true if a equals b).
$a := b$	The assignment operator (e.g., a equals b after this assignment).
$\mathbf{0}$	Well-known "zero time" for logical timestamps.
$\perp$	Well-known "null" value.

Table 4.1. Notation used in pseudo-code.

atic erasure code, if the fragments 1 through m are passed into **decode**, no computation is required.

The function **hash** provides a digest of whatever value is passed in. However, it only performs a collision resistant hash if malevolent components are tolerated by the R/W-PF member. If malevolent components are not tolerated, then **hash** is essentially a no-op.

4.5 Server

The server pseudo-code is shown in Figure 4.7. Servers expose the same interface, regardless of R/W-PF member.

A server retains candidates from write requests it accepts. A server stores accepted candidates in its history *History*. We refer to servers as being

```
/* Classifications. */
ClassificationType ∈ {INCOMPLETE, REPAIRABLE, COMPLETE}
/* Flags describing salient aspects of the resiliency model of R/W-PF. */
ASYNCHRONOUS_FLAG ∈ {TRUE, FALSE}
MALEVOLENT_CLIENTS_FLAG ∈ {TRUE, FALSE}
MALEVOLENT_SERVERS_FLAG ∈ {TRUE, FALSE}

/* Cross checksum. */
CrossChecksum ≡ Digest[U]            /* Array of digests indexed by server. */

/* Logical timestamp structure. */
Timestamp ≡ {
   LogicalTime                      /* Major component of logical time. */
   ClientID                         /* Client ID. */
   CrossChecksum                    /* Cross checksum. */
}

/* Candidate structure. A candidate is initialized to ⟨0, ⊥⟩. */
Candidate ≡ {
   Timestamp                        /* Logical timestamp. */
   Fragment                         /* Erasure-coded fragment. */
}

/* Server history. */
History ≡ {Candidate}                /* Set of accepted requests. */
```

Figure 4.5. Constants, data types, enumerations, and structures.

versioning servers because they retain each accepted candidate. Each server initializes its history to the well-known initial value for a candidate, $\langle \mathbf{0}, \perp \rangle$ (see line 700). The *History* is kept in stable storage such that it persists during a server crash and subsequent recovery.

Servers receive four different types of requests from clients: TIME_REQUEST, WRITE_REQUEST, READ_LATEST_REQUEST, and READ_PREVIOUS_REQUEST. A server responds to a read-latest request by returning the latest candidate in its history. A server responds to a read-previous request by returning the latest candidate in its history that has a timestamp less than that specified by the client. A server responds to a time request by returning the timestamp of the latest candidate in its history. If the R/W-PF tolerates malevolent clients, then a server validates the integrity of a candidate in a write request before accepting it. The check performed on line 824 protects a benign server from accepting a corrupt

```
encode(Object) :
/* Encode Object into n erasure-coded fragments. */
{Fragment_1, ..., Fragment_n} := erasure_code(Object)
return ({Fragment_1, ..., Fragment_n})

generate_fragments({Fragment_i, ..., Fragment_j})
/* Generate n fragments given m fragments. */
if (|{Fragment_i, ..., Fragment_j}| < m) then
   return (⊥)
end if
{Fragment_1, ..., Fragment_n} :=
      erasure_code({Fragment_i, ..., Fragment_j})
return ({Fragment_1, ..., Fragment_n})

decode({Fragment_i, ..., Fragment_j})
/* Perform systematic decode of fragments. */
/* (Actually only need to generate m fragments.) */
{Fragment_1, ..., Fragment_n} :=
      generate_fragments({Fragment_i, ..., Fragment_j})
Object := Fragment_1 | ... | Fragment_m
return (Object)

make_cross_checksum({Fragment_1, ..., Fragment_n})
/* Construct cross checksum given fragment set. */
for all (Server ∈ U) do
   CrossChecksum[Server] := hash(Fragment_Server)
end for
return (CrossChecksum)

hash(X):
if (MALEVOLENT_CLIENTS_FLAG ∨ MALEVOLENT_SERVERS_FLAG) then
   Digest := f(X)                         /* Collision-resistant hash f(X). */
else
   Digest := ⊥                            /* No need for cryptography. */
end if
return (Digest)
```

Figure 4.6. Functions for encoding and decoding objects.

candidate from a malevolent client. If benign servers did not validate candidates before accepting them, then malevolent clients could make benign servers look malevolent to another client. If the write candidate is well-formed, or if the R/W-PF does not tolerate malevolent clients, then the candidate is accepted. After accepting a candidate, a server sends a response to the client.

The manner in which timestamps are constructed, precludes distinct writes from having the same timestamp. As such, if a server receives the same write request multiple times, it must be for the same write. Therefore, it is safe for a server to "accept" the same candidate multiple times. This is necessary so that clients can repeatedly send the same request in an effort to establish a reliable channel to servers.

4.6 Client-server quorum RPCs

All of the quorum RPC functions have a similar structure. Requests are repeatedly broadcast to servers until a quorum of responses is received. There are more efficient quorum probing strategies than broadcasting requests to all servers, however, broadcast is the simplest to describe. Clients must repeatedly send requests to tolerate servers that crash and recovery. However, only a single response from each server is added to the response set.

R/W-PF members that specify a synchronous timing model and a crash or omission benign server failure model, may make use of timeouts. Once δ has elapsed after a client sends a request to server *Server*, if no response has been received from *Server*, then a TIMEOUT response is received. This is illustrated on lines 916-919 of **c_qprc_read_latest**.

Shown in Figure 4.8, the quorum RPC **c_qprc_read_latest** collects the latest candidate stored at each server in a quorum. Shown in Figure 4.9, the quorum RPC **c_qrpc_read_previous** collects the latest candidates stored at each server in a quorum that has a timestamp less than that specified by the client. These read RPCs allow clients to collect a partial observation of global system state and then, if need be, to traverse back in logical time. To protect against malevolent servers, clients validate the integrity of the

```
s_initialize() :
Server.History := (⟨0, ⊥⟩)                                /* Store in stable storage. */

s_receive_request() :
loop
  /* Poll for each type of request and reply accordingly. */
  if (poll_for_request(READ_LATEST_REQUEST) = TRUE) then
    LatestTimestamp := max(Server.History.Timestamp)
    LatestCandidate := (Candidate ∈ Server.History :
        Candidate.Timestamp = LatestTimestamp)            /* Singleton set. */
    reply(READ_LATEST_RESPONSE, Server, LatestCandidate)
  end if
  if (poll_for_request(READ_PREVIOUS_REQUEST) = TRUE) then
    Timestamp := receive_request()
    Prehistory := {Candidate ∈ Server.History :
        Candidate.Timestamp < Timestamp}
    LatestTimestamp := max(Prehistory.Timestamp)
    LatestCandidate := (Candidate ∈ Prehistory :
        Candidate.Timestamp = LatestTimestamp)            /* Singleton set. */
    reply(READ_PREVIOUS_RESPONSE, Server, LatestCandidate)
  end if
  if (poll_for_request(TIME_REQUEST) = TRUE) then
    LatestTimestamp := max(Server.History.Timestamp)
    reply(TIME_RESPONSE, Server, LatestTimestamp)
  end if
  if (poll_for_request(WRITE_REQUEST) = TRUE) then
    ⟨Timestamp, Fragment⟩ := receive_request()
    if (MALEVOLENT_CLIENTS_FLAG) then
      if (hash(Fragment) ≠ Timestamp.CrossChecksum[Server]) then
        goto loop                    /* Ignore requests from malevolent clients. */
      end if
    end if
    /* Accept candidate and store it in local history. */
    Server.History := Server.History ∪ (⟨Timestamp, Fragment⟩)
    reply(WRITE_RESPONSE, Server)
  end if
end loop
```

Figure 4.7. Pseudo-code for server *Server*.

```
c_qprc_read_latest() :
900: ResponseSet := ∅
901: repeat
902:    for all (Server_i ∈ U \ ResponseSet.Server) do
903:       send(READ_LATEST_REQUEST, Server_i, Timestamp)
904:    end for
905:    if (poll_for_response(READ_LATEST_RESPONSE) = TRUE) then
906:       ⟨Server, Response⟩ := receive_response()
907:       if (Server ∉ ResponseSet.Server) then
908:          Digest = hash(Response.Fragment)
909:          if (Digest ≠ Response.CrossChecksum[Server]) then
910:             /* Ignore responses from malevolent server. */
911:          else
912:             ResponseSet := ResponseSet ∪ (⟨Server, Response⟩)
913:          end if
914:       end if
915:    end if
916:    if (¬ASYNCHRONOUS_FLAG ∧ (elapsed(δ) = TRUE)) then
917:       ResponseSet := ResponseSet ∪
918:             {⟨Server, TIMEOUT⟩ : Server ∈ U \ ResponseSet}
919:    end if
920: until (∃Q ∈ 𝒬 : Q ⊆ ResponseSet)
921: return (ResponseSet)
```

Figure 4.8. Quorum RPC **c_qprc_read_latest** for client *ClientID*.

server response before adding it to the response set (see lines 909 and 1009). In the case of reading back in logical time, the client also makes sure the candidate has a timestamp less than that specified (see line 1012).

The quorum RPC **c_qrpc_read_time**, shown in Figure 4.10, collects the latest timestamp stored at each server in a quorum. Once a quorum of responses is received, the logical component of the largest timestamp in the response set is identified. This logical time is incremented and returned. The quorum RPC **c_qrpc_write** issues write requests to servers until a quorum of servers have replied. This implies that a quorum of servers, less any malevolent servers in the quorum, have accepted the candidate sent in the write request.

```
c_qrpc_read_previous(Timestamp) :
ResponseSet := ∅
repeat
   for all (Server_i ∈ U \ ResponseSet.Server) do
      send(READ_PREVIOUS_REQUEST, Server_i, Timestamp)
   end for
   if (poll_for_response(READ_PREVIOUS_RESPONSE) = TRUE) then
      ⟨Server, Response⟩ := receive_response()
      if (Server ∉ ResponseSet.Server) then
         Digest = hash(Response.Fragment)
         if (Digest ≠ Response.CrossChecksum[Server]) then
            /* Ignore responses from malevolent server. */
         else
            if (Response.Timestamp < Timestamp) then
               ResponseSet := ResponseSet ∪ (⟨Server, Response⟩)
            end if
         end if
      end if
   end if
   if (¬ASYNCHRONOUS_FLAG ∧ (elapsed(δ) = TRUE)) then
      ResponseSet := ResponseSet ∪
         {⟨Server, TIMEOUT⟩ : Server ∈ U \ ResponseSet}
   end if
until (∃Q ∈ 𝒬 : Q ⊆ ResponseSet)
return (ResponseSet)
```

Figure 4.9. Quorum RPC **c_qrpc_read_previous** for client *ClientID*.

4.7 Client

Pseudo-code for client functions is listed in Figure 4.11. The majority of protocol logic resides on the client.

The **c_write** function consists of determining the greatest logical time by performing a **c_qrpc_read_time** quorum RPC, encoding the object to store into erasure-coded fragments, constructing the cross checksum, constructing the timestamp for the write candidate, and then issuing write requests to servers by performing a **c_qrpc_write** quorum RPC. The cross checksum couples the value of the object being written with its encoding and with the timestamp.

The **c_read** function iteratively identifies and then classifies candidate sets until either a complete or repairable well-formed candidate set is found.

```
c_qrpc_read_time() :
ResponseSet := ∅
repeat
   for all (Server_i ∈ U \ ResponseSet.Server) do
      send(TIME_REQUEST, Server_i)
   end for
   if (poll_for_response(TIME_RESPONSE) = TRUE) then
      ⟨Server, Timestamp⟩ := receive_response()
      if (Server ∉ ResponseSet.Server) then
         ResponseSet := ResponseSet ∪ ⟨(Server, Timestamp⟩)
      end if
   end if
   if (¬ASYNCHRONOUS_FLAG ∧ (elapsed(δ) = TRUE)) then
      ResponseSet := ResponseSet ∪
            {⟨Server, TIMEOUT⟩ : Server ∈ U \ ResponseSet}
   end if
until (∃Q ∈ 𝒬 : Q ⊆ ResponseSet)
return (max(ResponseSet.Timestamp.LogicalTime) + 1)

c_qrpc_write(Timestamp, {Fragment_1, ..., Fragment_n}) :
ResponseSet := ∅
repeat
   for all (Server_i ∈ U \ ResponseSet.Server) do
      send(WRITE_REQUEST, Server_i, ⟨Timestamp, Fragment_i⟩)
   end for
   if (poll_for_response(WRITE_RESPONSE) = TRUE) then
      ⟨Server⟩ := receive_response()
      if (Server ∉ ResponseSet.Server) then
         ResponseSet := ResponseSet ∪ (⟨Server⟩)
      end if
   end if
   if (¬ASYNCHRONOUS_FLAG ∧ (elapsed(δ) = TRUE)) then
      ResponseSet := ResponseSet ∪
            {⟨Server, TIMEOUT⟩ : Server ∈ U \ ResponseSet}
   end if
until (∃Q ∈ 𝒬 : Q ⊆ ResponseSet)
return ()      /* Could return ResponseSet to track which servers replied. */
```

Figure 4.10. Quorum write RPCs for client *ClientID*.

The term *candidate set* refers to the set of candidates in the quorum of responses received from servers that have the same specified logical timestamp (usually the highest logical timestamp). Once such a candidate set is found, its fragments are decoded, and the object is returned.

The read begins by performing a **c_qprc_read_latest** quorum RPC. The response set returned from **c_qprc_read_latest** is used to identify the candidate set. Next, the candidate set is classified. The rules for classifying a candidate set as incomplete, repairable, or complete differ among R/W-PF members. The function **classify**, which returns one of INCOMPLETE, REPAIRABLE, COMPLETE, is defined in Figure 4.12. It closely follows constraint 5.11, developed in Section 5.1. For asynchronous R/W-PF members, and synchronous members with the crash-recovery benign server failure model, the set *TimeoutSet* will always be empty.

If the candidate set is classified as complete or repairable, the client determines if the candidate set is well-formed. The client calls **c_validate** which constructs a cross checksum for the candidate set. The pseudo-code for **c_validate** is in Figure 4.12. If the cross checksum for the candidate set differs from the cross checksum in the timestamp for the candidate set, then this is a *poisonous candidate set* (i.e., the candidate in the candidate set corresponds to a poisonous write by a malevolent client). We refer to this method of protecting against malevolent clients as self-validating timestamps and discuss it in more detail in Section 4.8. If an R/W-PF member does not tolerate malevolent clients, then **c_validate** is effectively a no-op; it returns TRUE immediately. If the candidate set passes validation, then the object can be decoded from the fragments in the candidate set and be returned.

If validation of a repairable or complete candidate set fails, or the candidate set is classified as incomplete, then the client reads back in logical time. The client calls the **c_qrpc_read_previous** quorum RPC to read back in logical time. Candidate classification begins again with the new (refreshed) response set.

The read returns a pair: a logical timestamp and an object. The client likely only makes use of the object. Including the logical timestamp in the return value simplifies discussion of return values.

```
c_write(Object) :
LogicalTime := c_qrpc_read_time()
{Fragment_1, ..., Fragment_n} := encode(Object)
CrossChecksum := make_cross_checksum({Fragment_1, ..., Fragment_n})
Timestamp := ⟨LogicalTime, ClientID, CrossChecksum⟩
c_qrpc_write(Timestamp, {Fragment_1, ..., Fragment_n})
return

c_read() :
ReadResponseSet := c_qprc_read_latest()
loop
  CandidateTimestamp := max(ReadResponseSet.Timestamp)
  CandidateSet := {Candidate ∈ ReadResponseSet :
      Candidate.Timestamp = CandidateTimestamp}
  if (classify(CandidateSet, ReadResponseSet) = INCOMPLETE) then
    /* Must read back in time. */
  else if (c_validate(CandidateSet)) then
    if (classify(CandidateSet, ReadResponseSet) = COMPLETE) then
      Object := decode(CandidateSet.Fragment)
    else
      /* classify(CandidateSet, ReadResponseSet) = REPAIRABLE */
      {Fragment_1, ..., Fragment_n} := generate_fragments(CandidateSet)
      c_qrpc_write(CandidateTimestamp, {Fragment_1, ..., Fragment_n})
      Object := decode({Fragment_1, ..., Fragment_n})
    end if
    return (⟨CandidateTimestamp, Object⟩)
  end if
  /* INCOMPLETE or validation failed. */
  ReadResponseSet := c_qrpc_read_previous(CandidateTimestamp)
end loop
```

Figure 4.11. Client pseudo-code for read and write operations.

4.8 Self-validating timestamps

As shown in the pseudo-code, a number of steps are taken to protect against malevolent components. Cross checksums are employed to protect against malevolent servers [Gong, 1989]. Cross checksums are the set of n collision-resistant hashes of the n erasure-coded fragments. A client can compare the hash of a fragment returned to it with the corresponding hash in the cross checksum. Replicating the cross checksum at all servers ensures that the client can identify a correct copy of the cross checksum to which to compare a fragment's hash. Cross checksums prevent a malevolent server

```
classify(CandidateSet, ReadResponseSet) :
S := {CandidateSet.Server}
TimeoutSet := {⟨Server, Response⟩ ∈ ReadResponseSet :
      Response = TIMEOUT}
S_t := {Server ∈ TimeoutSet.Server}
if (∃Q ∈ 𝒬 : Q ⊆ S ∪ S_t) then
   return (COMPLETE)
else if ((∀Q ∈ 𝒬 : Q ⊈ S ∪ S_t) ∧ (∃Q ∈ 𝒬, R ∈ ℛ(Q) : R ⊆ S)) then
   return (REPAIRABLE)
end if
return (INCOMPLETE)

c_validate(CandidateSet)
if (MALEVOLENT_CLIENTS_FLAG = TRUE) then
   CandidateCrossChecksum := CandidateSet.Timestamp.CrossChecksum
   {Fragment_1, ..., Fragment_n} :=
         generate_fragments(CandidateSet.Fragment)
   ValidatedCrossChecksum :=
         make_cross_checksum({Fragment_1, ..., Fragment_n})
   if (ValidatedCrossChecksum ≠ CandidateCrossChecksum) then
      return (FALSE)
   end if
end if
/* Candidate is well-formed. */
return (TRUE)
```

Figure 4.12. Pseudo-code for client functions called by **c_read**.

from undetectably corrupting the integrity of a fragment it hosts.

Self-validating timestamps[1] are employed to protect against malevolent clients [Goodson et al., 2004a]. Self-validating timestamps are an extension of cross checksums.

The cross checksum is part of a logical timestamp (see line 112). Embedding the cross checksum in the logical timestamp makes it impossible (assuming collision-resistant hash functions) to write different erasure-coded fragments to some server that have the same logical timestamp. Servers validate the candidates sent to them by clients (see line 824). Such validation ensures that benign servers only accept fragments that match their entry

[1]In Goodson et al. [2004a], we referred to aspects of self-validating timestamps as *validating timestamps*, *storage-node (server) verification*, and *validated cross checksums*, but did not name the overall technique.

in the cross checksum. These two measures ensure that at most some set of $B \in \mathcal{B}$ servers respond to reads with requests that lack integrity.

The final step for self-validating timestamps occurs during a read. Once a repairable or complete candidate set is identified, it is validated (see **c_validate** in Figure 4.11). Validation consists of generating the cross checksum for the fragments that are in the candidate set, and comparing it to the cross checksum that is actually in the logical timestamp. If the cross checksum of the candidate set does not match the cross checksum in the logical timestamp, then a malevolent client performed a poisonous write. All benign clients detect that the candidate set for the given timestamp is due to a poisonous write; this is true regardless of which m fragments are used to generate the other $n - m$ fragments. Clients treat candidates that are not well-formed like they do incomplete candidates and read back in logical time.

5 Constraints and classification

Although the R/W-PF offers much versatility, there are constraints on R/W-PF members. Given the timing model, server failure model, and erasure code, there are constraints on quorum construction and on witnesses. The client failure model does not affect the constraints. The constraints could be formulated in a somewhat different manner, by selecting different "dependent" parameters. For example, given the resiliency models and quorum constructions, the erasure code and witnesses are constrained. Classification of a candidate, a fundamental step in R/W-PF members, is tightly coupled to the quorum construction. As such, the rules for classifying a candidate are presented in this chapter. This chapter also presents specific constraints and classification rules for threshold quorums that are used in PASIS, the prototype implementation.

5.1 Constraints

5.1.1 Asynchronous timing model

Under the asynchronous timing model, the crash-recovery, omission, and crash benign server failure models are indistinguishable. As such, the benign server failure model does not affect quorum construction or classification.

Live quorum exists

To guarantee that a live quorum exists in the asynchronous timing model,

$$\forall T \in \mathcal{T}, \exists Q \in \mathcal{Q} : Q \subseteq U \setminus T \tag{5.1}$$

This constraint ensures that even if all of the faulty servers in an execution do not reply to requests, there still exists a quorum of live servers that do respond.

Byzantine masking quorum

Malkhi and Reiter [1998] define the intersection constraint for Byzantine masking quorums:

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B_i, B_j \in \mathcal{B} : (Q_i \cap Q_j) \setminus B_i \not\subseteq B_j. \tag{5.2}$$

This constraint ensures that all quorums intersect so that there are sufficient benign servers in the intersection to "out vote" any malevolent servers in the intersection.

Repairable sets

If an erasure code is used such that the benign servers in the intersection due to the masking quorum are a decodable set, then the masking quorum is sufficient. However, to include $\mathcal{M}$, the decodable sets, in the specification of a R/W-PF member, we define repairable sets for every quorum. Each quorum $Q \in \mathcal{Q}$, in conjunction with $\mathcal{M}$, defines a set of *repairable sets* $\mathcal{R}(Q) \subseteq 2^Q$. Essentially, a repairable set is sufficient to "out vote" any malevolent responses and to decode the erasure-coded object:

$$\mathcal{R}(Q) = \{R \in 2^Q : (\forall B \in \mathcal{B}, R \not\subseteq B) \wedge (\exists M \in \mathcal{M}(Q), M \subseteq R)\} \tag{5.3}$$

Note that some repairable sets are subsets of other repairable sets.

Byzantine masking m-quorum

Constraint (5.2) is modified to accommodate erasure-coded objects, by incorporating repairable sets. In so doing, we define *Byzantine masking m-quorums*:

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall B \in \mathcal{B}, \exists R \in \mathcal{R}(Q_i) : (Q_i \cap Q_j) \setminus B \supseteq R. \tag{5.4}$$

The concept of an *m-quorum*[1] was developed simultaneously by Frølund et al. [2004]. However, Frølund et al. considered only a benign failure model in their development of m-quorums.

Witnesses

Additional constraints are required for read and write witnesses. Read witnesses allow a subset of a quorum of servers to reply with a witness (i.e., a logical timestamp). Write witnesses allow a subset of a quorum of servers to receive only a witness, rather than a witness and a fragment.

For read witnesses, a quorum Q of replies can be separated into a subset of servers Q_r that replied with read witnesses and a subset of servers $Q \setminus Q_r$ that replied with candidates (i.e., timestamp and fragment pairs). As such, to decode a response,

$$\exists M \in \mathcal{M}(Q) : Q \setminus Q_r \supseteq M. \tag{5.5}$$

The constraint on read witnesses is not important for safety, since if too few fragments are requested, additional fragments can be read.

For write witnesses, a quorum Q of requests can be separated into a subset of servers Q_w that are sent write witnesses and a subset of servers $Q \setminus Q_w$ that are sent fragments. The constraint on write witnesses is important for safety, since if too few fragments are stored, it may not be possible to read the object. To ensure that an object can always be decoded, sufficient fragments must be in the intersection between the quorum written to and the quorum read from to decode. Thus for some quorum $Q \in \mathcal{Q}$,

$$\forall Q_i \in \mathcal{Q}, \forall B \in \mathcal{B}, \exists M \in \mathcal{M}(Q) : ((Q \setminus Q_w) \cap Q_i) \setminus B \supseteq M. \tag{5.6}$$

Note that write witnesses are necessarily more restricted than read witnesses: a server that hosts only a write witness can only respond with a read witness.

[1] Frølund et al. coined the term *m-quorum*. We originally referred to such quorums, as "quorums that are big enough to allow data to be erasure-coded," which lacks elegance.

Classification

A candidate is *established* if all of the benign servers in a quorum have the candidate in their history. The intuition behind established candidates is that any read that begins after a candidate is established will return the established candidate, or some other established candidate with a higher timestamp.

With perfect global information, it would be possible to observe directly whether or not a candidate is established. Candidates are classified as *complete*, *repairable*, or *incomplete*. The constraints on the quorum system, in conjunction with *classification rules*, ensure that established candidates are classified as repairable or complete. Repairing a candidate, i.e., completing the write that generated the candidate, is a fundamental aspect of the R/W-PF.

Given a set of server responses S that share the same candidate, the classification rules for that candidate are as follows:

$$\mathbf{classify}(S) = \begin{cases} \textsc{complete} & \text{if } \exists Q \in \mathcal{Q} : Q \subseteq S, \\ \textsc{repairable} & \text{if } (\forall Q \in \mathcal{Q} : Q \not\subseteq S) \wedge \\ & (\exists Q \in \mathcal{Q}, R \in \mathcal{R}(Q) : R \subseteq S), \\ \textsc{incomplete} & \text{otherwise.} \end{cases} \tag{5.7}$$

Remember that the repairable sets $\mathcal{R}(Q)$ for quorum Q define the intersection property between Q and other quorums which includes a subset of benign servers that are a decodable set. Therefore, the classification rule is inter-related with the quorum system construction and the erasure coding scheme.

5.1.2 Synchronous timing model

Synchronous members of the R/W-PF may wait for responses from all servers. However, some responses may simply be TIMEOUT. The effect of timeout responses on classification and on system constraints depends on the benign server failure model (i.e., crash, omission, or crash-recovery).

For the crash-recovery benign server failure model, quorum system constraints and classification are the same as for the asynchronous timing model. Given the crash-recovery failure model for benign servers, requests to servers that timeout are retried until a quorum of responses, none of which are TIMEOUT, is received.

A distinguishing feature of the omission and crash benign server failure models, relative to the crash-recovery server failure model, is that only a subset of the servers may ever crash. That is, in an execution in which a set of servers T are faulty, $U \setminus T$ servers are always-up. There are no benign servers that crash and then recover (i.e. that are eventually-up). As such, in the synchronous timing model, it is possible to wait for responses from more than a quorum of servers. For example, once δ has elapsed, responses, even if some are TIMEOUT, are received from all servers to which requests were sent. In such cases, read classification has more information with which to classify candidates.

Live quorum exists

For omission and crash benign server failure models, it is possible to use a smaller universe for R/W-PF members with a synchronous timing model than with an asynchronous timing model. Since all servers can respond, albeit some set of servers $T \in \mathcal{T}$ with TIMEOUT, the existence of a live quorum requires only that

$$\forall Q \in \mathcal{Q} : Q \subseteq U. \qquad (5.8)$$

Comparing (5.8) with (5.1) implies that synchronous R/W-PF members that tolerate crash or omission benign server failures require smaller quorums than asynchronous members and members that tolerate crash-recovery benign server failures.

Byzantine non-blocking m-quorum

For the omission and crash benign server failure models, a candidate is established if all of the *correct* servers in a quorum have the candidate in their history. The rationale for defining established candidates in terms of correct servers for synchronous R/W-PF members, is so that it is a *pure* property [Charron-Bost et al., 2000]: a candidate is established (or not) irrespective of the set of servers which are currently failed.

This definition of established leads to the following quorum intersection constraint:

$$\forall Q_i, Q_j \in \mathcal{Q}, \forall T \in \mathcal{T}, \exists R \in \mathcal{R}(Q_i) : (Q_i \cap Q_j) \setminus T \supseteq R. \tag{5.9}$$

This constraint is subtly different than (5.4). Since responses from all servers in a quorum are expected, faulty servers are "subtracted" from the intersection, rather than malevolent servers. Synchronous Byzantine quorum systems [Bazzi, 2000] permit non-blocking quorum access [Bazzi, 1999, 2001]. As such, quorum probing is unnecessary for the synchronous R/W-PF members, because responses from a quorum of servers is guaranteed, albeit some may be timeout responses.

Witnesses

For read witnesses in synchronous members, the constraint for asynchronous members, (5.5), applies. For write witnesses, the constraint for asynchronous members, (5.6), does not apply. The constraint on write witnesses must account for the non-blocking nature of synchronous quorums:

$$\forall Q_i \in \mathcal{Q}, \forall T \in \mathcal{T}, \exists M \in \mathcal{M}(Q) : ((Q \setminus Q_w) \cap Q_i) \setminus T \supseteq M. \tag{5.10}$$

Classification

Given that no more than $T \in \mathcal{T}$ servers are faulty in an execution, a client is guaranteed to receive responses from a quorum Q of servers, of which no more than $T \subset Q$ are TIMEOUT. Given a set of server responses S that share the same candidate and a set of server responses S_t that are TIMEOUT, the classification rules for that candidate are as follows:

$$\mathbf{classify}(S, S_t) = \begin{cases} \text{COMPLETE} & \text{if } \exists Q \in \mathcal{Q} : Q \subseteq S \cup S_t, \\ \text{REPAIRABLE} & \text{if } (\forall Q \in \mathcal{Q} : Q \not\subseteq S \cup S_t) \wedge \\ & (\exists Q \in \mathcal{Q}, R \in \mathcal{R}(Q) : R \subseteq S), \\ \text{INCOMPLETE} & \text{otherwise.} \end{cases} \tag{5.11}$$

Fail-stop servers and repairable sets

A server that experiences a fail-stop failure crashes in such a manner that it takes no further action and its failure is detectable [Schneider, 1984]. In a synchronous system, a TIMEOUT response from a server can be used to transform crash failures into fail-stop failures. Although clients can detect that a server has failed, malevolent servers may appear crashed to some clients and alive to other clients.

If sufficient TIMEOUT responses are received, it is possible to deduce that some non-timeout responses are *not* from malevolent servers. It is thus possible to formulate the repairable sets as a function of the timeout responses received. Given the set of server responses S_t that are TIMEOUT, (5.3) becomes,

$$\begin{aligned} \mathcal{R}(Q, S_t) = \{R \in 2^Q : & (\exists M \in \mathcal{M}(Q), M \subseteq R) \wedge \\ & (\forall B \in \mathcal{B}, \exists T \in \mathcal{T}, B \subseteq T, R \not\subseteq (T \setminus S_t) \cap B)\} \end{aligned} \tag{5.12}$$

If $\exists B \in \mathcal{B} : S_t \subseteq B$, then timeouts do not change the repairable sets. However, each additional timeout response beyond this reduces the number of responses in S that may be from a malevolent server. For classification, $\mathcal{R}(Q, S_t)$ could replace $\mathcal{R}(Q)$ in (5.11). However, we have not identified any tangible benefits to doing so. We include these details though, because they suggest that, in other protocol families, there may be substantive differences between the crash/fail-stop and omission benign server failure models.

5.2 Threshold quorum systems

This section reproduces the general constraints developed Section 5.1, but specialized for threshold quorum systems. It also describes the threshold quorum construction used in the prototype implementation.

Threshold quorum systems are parameterized as follows:

- all quorums $Q \in \mathcal{Q}$ are of size q;
- all repairable sets $R \in \mathcal{R}(Q)$ are at least of size r;
- all faulty server sets $T \in \mathcal{T}$ are of size t;
- all malevolent server sets $B \in \mathcal{B}$ are of size b and $b \leq t$;
- all decodable sets $M \in \mathcal{M}(Q)$ are of size m and $m \leq r$;
- and the universe of servers U is of size n.

5.2.1 Asynchronous timing model

For threshold quorums, the existence of a live quorum guaranteed by (5.1), becomes,

$$q + t \leq n. \tag{5.13}$$

From (5.3), it follows that any subset of a quorum that share the same candidate and that is at least of size r is repairable:

$$r = \mathbf{max}(b + 1, m). \tag{5.14}$$

And so, from (5.4), threshold Byzantine masking m-quorums must obey the constraint,

$$2q - n \geq b + r. \tag{5.15}$$

Letting q_r be the number of read witness responses, then, from (5.5), it follows that,

$$q - q_r \geq m. \tag{5.16}$$

Letting q_w be the number of write witnesses written, then, from (5.6) and (5.15),

$$r - q_w \geq m. \tag{5.17}$$

From (5.13) and (5.15), it follows that $2q - r - b \geq n \geq q + t$, and thus that,

$$r + t + b \leq q. \tag{5.18}$$

And, from (5.13) and (5.18), it follows that,

$$r + 2t + b \leq n. \tag{5.19}$$

Finally, from (5.7), it follows that the classification rule for a set of responses S, is,

$$\textbf{classify}(S) = \begin{cases} \text{COMPLETE} & \text{if } q \leq |S|, \\ \text{REPAIRABLE} & \text{if } r \leq |S| < q, \\ \text{INCOMPLETE} & \text{otherwise.} \end{cases} \tag{5.20}$$

Threshold quorum construction

Consider a threshold quorum system parameterized by Δ. The intention of this construction is to provide throughput-scalability. The (Δ, t, b, m)-threshold quorum construction is as follows:

$$\begin{aligned} r &= \mathbf{max}(m, b+1); \\ q &= \Delta + t + b + r; \\ n &= 2\Delta + 2t + b + r; \\ q_r &= q - m; \\ q_w &= \mathbf{max}(b + 1 - m, 0). \end{aligned} \tag{5.21}$$

Notice that, for some fixed t, b, and m, as Δ increases, the per-server load $\frac{q}{n}$ asymptotically approaches $\frac{1}{2}$. Table 5.1 lists example (Δ, t, b, m)-threshold quorum constructions, based on (5.21). This threshold quorum construction is simply a majority voting construction.

Δ	t	b	m	r	q	n	q_r	q_w
0	1	1	1	2	4	5	3	1
0	1	1	2	2	4	5	2	0
0	1	1	3	3	5	6	2	0
0	2	1	1	2	5	7	4	1
0	2	1	2	2	5	7	3	0
0	2	1	3	3	6	8	3	0
1	1	1	1	2	5	7	4	1
1	1	1	2	2	5	7	3	0
1	1	1	3	3	6	8	3	0
2	3	3	1	4	12	17	11	3
2	3	3	2	4	12	17	10	2
2	3	3	3	4	12	17	9	1
2	3	3	4	4	12	17	8	0
2	3	3	5	5	13	18	8	0

Table 5.1. Example (Δ, t, b, m)-threshold quorums for asynchronous timing model.

5.2.2 Synchronous timing model

The synchronous timing model with crash-recovery benign server failure model uses the threshold quorum construction (5.21) for the asynchronous timing model.

The existence of a live quorum given by constraint (5.8), for omission and fail-stop benign server failure models, becomes,

$$q \leq n, \tag{5.22}$$

and the Byzantine non-blocking m-quorum constraint (5.9) becomes,

$$2q - n \geq t + r. \tag{5.23}$$

From (5.22) and (5.15), it follows that $2q - r - t \geq n \geq q$, and thus,

$$t + r \leq q \leq n. \tag{5.24}$$

Previously, Goodson et al. [2004b] identified a looser bound, $2t + 1 \leq n$, for such R/W-PF members[2]. Whereas, for $m \leq b + 1$, (5.24) reduces to $t + b + 1 \leq n$.

For the crash and omission benign server failure model, classification of a candidate given a set S of responses that share the same candidate, and a set of S_t of timeouts, is as follows:

$$\mathbf{classify}(S, S_t) = \begin{cases} \text{COMPLETE} & \text{if } q - |S_t| \leq |S|, \\ \text{REPAIRABLE} & \text{if } r \leq |S| < q - |S_t|, \\ \text{INCOMPLETE} & \text{otherwise.} \end{cases} \tag{5.25}$$

The general constraint differs for write witnesses in (5.6), the asynchronous timing model, and (5.10), the synchronous timing model. However, for threshold quorums, witnesses are constrained like in the asynchronous timing model (cf. (5.16) and (5.17)). The salient differences are captured by the differences in the bounds on quorum size.

Analogous synchronous threshold quorum constructions to those given for asynchronous R/W-PF members are as follows. The (Δ, t, b, m)-threshold quorum construction for the synchronous timing model is as follows:

$$\begin{aligned} r &= \mathbf{max}(m, b+1); \\ q &= \Delta + t + r; \\ n &= 2\Delta + t + r; \\ q_r &= q - m; \\ q_w &= \mathbf{max}(b + 1 - m, 0). \end{aligned} \tag{5.26}$$

[2] The notation in the cited paper uses N, not n, for the universe size. Moreover, the cited paper considers only constraints for $m \leq b + 1$.

6 Correctness

The client failure model for an R/W-PF member dictates the safety guarantee achieved. The desired safety property for the R/W-PF is *linearizability* of read and write operations. Operations are linearizable if their return results are consistent with an execution in which each operation is performed instantaneously at a distinct point in time between its start time and its completion time. Members of the R/W-PF that do not tolerate malevolent clients achieve linearizability as originally defined Herlihy and Wing [1990].

It is necessary to adapt linearizability for members of the R/W-PF that tolerate malevolent clients. The adaptations necessary to interpret linearizability in our context arise from the fact that malevolent clients may not follow the R/W-PF member specification. We exclude reads by malevolent clients from the set of linearizable operations. Poisonous writes by malevolent clients are ignored by benign clients. As such, only non-poisonous writes that complete are included in the set of linearizable operations.

Write operations by malevolent clients do not have a well-defined start time. We consider that all writes by malevolent clients begin at the start of the execution. As such, writes by malevolent clients are concurrent to all operations that begin before they complete.

For all R/W-PF members, reads and writes performed by correct clients are wait-free [Herlihy, 1991; Jayanti et al., 1998]. Informally, achieving wait-freedom means that each client can complete its operations in finitely many steps regardless of the actions performed or failures experienced by other clients. For a formal definition of wait-freedom, see Herlihy [1991]. The liveness guarantee of wait-freedom requires unbound storage capacity.

6.1 Proof of safety

Before defining the duration of write operations, it is necessary to define what it means for a server to *accept* a write request.

Definition 6.1.1 (*accept*). Server *Server*, *accepts* a write request with fragment *Fragment* and timestamp *Timestamp* only if the hash of the fragment (**hash**(*Fragment*)) equals the corresponding entry in the cross checksum ($Timestamp.CrossChecksum[Server]$).

See line 824 of function **s_receive_request** in Figure 4.7 for more details about server logic. Remember that **hash** is a collision-resistant hash function, if the R/W-PF member tolerates malevolent components. An accepted write request is stored, as a candidate, in the history of a benign server. A benign server retains its history over the course of a crash and subsequent recovery.

Definition 6.1.2 (*write begin,* $w_{Timestamp}$). A write operation $w_{Timestamp}$, *begins* when a benign client invokes the **c_write** operation locally that issues a write request bearing timestamp *Timestamp*.

We use the label $w_{Timestamp}$ as a shorthand for the write operation with timestamp *Timestamp* for the remainder of this section.

Since malevolent clients may not follow their specification, it is difficult to state when a write operation by such a client begins. We avoid making such a statement. Instead, we assume that all write operations by malevolent clients begin at the start of the execution.

It may be possible to state when a write operation by a malevolent client is in progress. For example, "if a benign server accepts a write request from a malevolent client, then ...". However, such definitional deftness is unneeded. A definition of when write operations complete is sufficient.

Definition 6.1.3 (*established*). For the asynchronous timing model and for the synchronous timing model with crash-recovery benign server failure model, a candidate $\langle Timestamp, Fragment\rangle$ is established once all the *benign* servers in a quorum have accepted a write request with timestamp

Timestamp. Whereas, for the synchronous timing model with crash/fail-stop or omission benign server failure model, a candidate $\langle Timestamp, Fragment\rangle$ is established once all the *correct* servers in a quorum have accepted a write request with timestamp *Timestamp*.

Definition 6.1.4 (*well-formed***).** A write operation $w_{Timestamp}$ and the corresponding candidate are *well-formed* if the timestamp is self-validating. That is, the function **c_validate** returns TRUE for any decodable set of the fragments sent with timestamp *Timestamp* that could be accepted at the corresponding benign servers.

Definition 6.1.5 (*write complete***).** A write operation $w_{Timestamp}$ *completes* once a well-formed candidate with timestamp *Timestamp* is established.

Definition 6.1.5 applies to write operations by benign clients as well as by malevolent clients. A poisonous write by a malevolent client does not complete, since it is not well-formed.

Because return values of reads by malevolent clients obviously need not comply with any correctness criteria, we disregard read operations by malevolent clients in reasoning about linearizability, and define the duration of reads only for those executed by benign clients only.

Definition 6.1.6 (*read begin***).** A read operation executed by a benign client *begins* when the client invokes **c_read** locally.

Definition 6.1.7 (*read complete***).** A read operation executed by a benign client *completes* when the invocation **c_read** returns $\langle timestamp, value\rangle$.

Clearly, a benign client that crashes during a read operation does not complete the read.

Lemma 6.1.8. *Let c be a benign client. If c performs a read operation that returns* $\langle Timestamp, v\rangle$*, then the candidate* $\langle Timestamp, v\rangle$ *is established and well-formed.*

Proof. Only an established candidate is classified as complete, therefore $\langle Timestamp, v\rangle$ is established. See classification rules (5.7) and (5.11) in

Chapter 5. Only a candidate with a self-validating timestamp is returned from a benign client, therefore $\langle Timestamp, v\rangle$ is well-formed. □

Lemma 6.1.9. *Let c_1 and c_2 be benign clients. If c_1 performs a read operation that returns $\langle Timestamp_1, v_1\rangle$, c_2 performs a read operation that returns $\langle Timestamp_2, v_2\rangle$, and $Timestamp_1 = Timestamp_2$, then $v_1 = v_2$.*

Proof. Lemma 6.1.8 proves that $\langle Timestamp_1, v_1\rangle$ and $\langle Timestamp_2, v_2\rangle$ are established and well-formed. Since $Timestamp_1 = Timestamp_2$ (and thus $Timestamp_1.CrossChecksum = Timestamp_2.CrossChecksum$), v_1 and v_2 must be the same (assuming the existence of collision-resistant hash functions). □

Definition 6.1.10 (*precedes*, $\rightarrow$). Let o_1 denote an operation that completes (a read operation by a benign client, or a write operation), and let o_2 denote an operation that begins (a read or write by a benign client). o_1 *precedes* o_2 if o_1 completes before o_2 begins. The precedence relation is written as $o_1 \rightarrow o_2$.

Lemma 6.1.11. *If $w_{Timestamp} \rightarrow w_{Timestamp'}$, then $Timestamp < Timestamp'$.*

Proof. By definition, $w_{Timestamp}$ completes and so there exists a candidate $\langle Timestamp, v\rangle$ that is well-formed and established. Since $w_{Timestamp} \rightarrow w_{Timestamp'}$, and because of the quorum intersection constraints, the quorum RPC **c_qrpc_read_time** for $w_{Timestamp'}$ receives at least one TIME_RESPONSE response from a benign server that hosts a candidate with timestamp $Timestamp$ (i.e., from a benign server that accepted $w_{Timestamp}$. The quorum intersection constraints of interest are (5.4) and (5.9) in Chapter 5. As such, $w_{Timestamp'}$ observes some timestamp greater than or equal to $Timestamp$ and constructs $Timestamp'$ to be greater than $Timestamp$. See **c_qrpc_read_time** in Figure 4.10 and **c_write** in Figure 4.11 for more details of timestamp construction.

□

A malevolent server can return a logical timestamp greater than that of the preceding write operation; however, this still advances logical time and Lemma 6.1.11 holds. Timestamps and malevolent components are discussed in Section 7.8.

Observation 6.1.12. Timestamp order is a total order on write operations. The timestamps of write operations by benign clients respect the precedence order among writes.

Definition 6.1.13 ($r_{Timestamp}$). Let $v_{Timestamp}$ denote the value written by $w_{Timestamp}$. We use $r_{Timestamp}$ to denote a read operation by a benign client that returns $\langle Timestamp, v_{Timestamp}\rangle$. By Lemma 6.1.8, $w_{Timestamp}$ is established and well-defined.

Lemma 6.1.14. *If $w_{Timestamp}$ is complete, and if $w_{Timestamp} \rightarrow r_{Timestamp'}$, then $Timestamp \leq Timestamp'$.*

Proof. Since $w_{Timestamp}$ is complete, there exists an established well-formed candidate $\langle Timestamp, v\rangle$. By Lemma 6.1.8, read operations only return values from complete write operations. As such, $r_{Timestamp'}$ must either return the value with timestamp $Timestamp$ or a value with a greater timestamp. Therefore, $Timestamp \leq Timestamp'$. $\square$

Observation 6.1.15. It follows from Lemma 6.1.14, that for any read $r_{Timestamp}$, either $w_{Timestamp} \rightarrow r_{Timestamp}$ and $w_{Timestamp}$ is the latest complete write that precedes $r_{Timestamp}$, or $w_{Timestamp} \not\rightarrow r_{Timestamp}$ and $r_{Timestamp} \not\rightarrow w_{Timestamp}$ (i.e., $w_{Timestamp}$ and $r_{Timestamp}$ are concurrent).

Observation 6.1.16. It also follows from Lemmas 6.1.8 and 6.1.14 that if $r_{Timestamp} \rightarrow r_{Timestamp'}$, then $Timestamp \leq Timestamp'$. As such, there is a partial order $\prec$ on read operations by benign clients defined by the timestamps associated with the values returned (i.e., of the write operations read). More formally, $r_{Timestamp} \prec r_{Timestamp'} \iff Timestamp < Timestamp'$.

Since Lemma 6.1.11 ensures a total order on write operations, ordering reads according to the timestamps of the write operations whose values they return yields a partial order on read operations. Lemma 6.1.14 ensures that this partial order is consistent with precedence among reads. Therefore, any way of extending this partial order to a total order yields an ordering of reads that is consistent with precedence among reads. Thus, Lemmas 6.1.11 and 6.1.14 guarantee that this totally ordered set of operations is consistent with precedence. This implies the natural extension of linearizability to R/W-PF members that tolerate malevolent clients (i.e., ignoring reads by malevolent clients and assuming writes by malevolent clients begin at the start of the execution). In particular, it implies linearizability as originally defined by Herlihy and Wing [1990] if all clients are benign.

6.2 Proof of liveness

Lemma 6.2.1. *All quorum RPCs eventually return.*

Proof. The quorum RPCs are **c_qprc_read_latest**, **c_qrpc_read_previous**, **c_qrpc_read_time**, and **c_qrpc_write**. For the asynchronous timing model and for the crash-recovery benign server failure model with synchronous timing model, there is at least one quorum comprised exclusively of good servers. See Section 3.1.2 for a definition of *good* and constraint (5.1) for the constraint on quorum system construction that ensures this property. Quorum probing and repeated sends of requests therefore ensures a quorum of responses will be received, thus allowing the quorum RPC to return eventually. For the crash and omission benign server failure models with synchronous timing model, because of the non-blocking quorums employed, a quorum of responses will be received, of which $Q \setminus T$ are from correct servers; the reminder may be from malevolent servers and/or TIMEOUT. Such a quorum of responses ensures that the quorum RPC returns. □

Lemma 6.2.2. *A write operation by a correct client completes.*

Proof. A write operation by a correct client performs two quorum RPCs: one to retrieve the latest timestamp (**c_qrpc_read_time**) and one to write the value (**c_qrpc_write**). Lemma 6.2.1 proves that both quorum RPCs return. For a write operation to complete it must establish a well-formed candidate. A correct client constructs the self-validating timestamp correctly, therefore the resulting candidate is well-formed. For the asynchronous timing model and crash-recovery benign server failure model with synchronous timing model, a quorum of responses to the **c_qrpc_write** quorum RPC, implies that all of the benign servers in a quorum accepted the candidate and it is therefore established. For the crash and omission benign server failure models with synchronous timing model, a quorum of responses to the **c_qrpc_write** quorum RPC, implies that all of the correct servers in a quorum accepted the candidate and it is therefore established. □

Lemma 6.2.3. *A read operation by a correct client completes.*

Proof. A read operation by a correct client performs one quorum RPC to read the latest candidate (**c_qprc_read_latest**) and, if needed, additional quorum RPCs to read candidates with earlier timestamps (**c_qrpc_read_previous**) and/or a quorum RPC to repair a candidate (**c_qrpc_write**). Lemma 6.2.1 proves that all such quorum RPCs return.

A read returns once it identifies a well-formed candidate that it classifies as complete. (A read operation that identifies a well-formed candidate that it classifies as repairable performs a **c_qrpc_write** and the candidate is then classified complete.) A read operation reads candidates with earlier timestamps given a candidate it classifies as incomplete or not well-formed. At the start of the system execution, there exists a well-formed established candidate $\langle \mathbf{0}, \perp \rangle$. There can only be a finite number of candidates with higher timestamps then $\mathbf{0}$, and so the read operation eventually completes. □

The R/W-PF members achieve a strong liveness property, namely wait-freedom [Herlihy, 1991; Jayanti et al., 1998]. Informally, each operation by a correct client completes with certainty, even if all other clients fail, provided that client, server, and network failures are within the resiliency model specified. Wait-freedom was originally defined for an asynchronous timing model. However, the definition applies in the synchronous timing model without modification. Note that write operations require a finite number of client steps (two) to complete, whereas read operations require a finite but unbounded number of client steps. Given a finite number of clients in the system, the number of steps required to complete a read operation could be bound (see Section 7.7.1 for more detailed discussion).

To achieve wait-freedom R/W-PF members with an asynchronous timing model require unbound storage capacity. Servers must retain the candidate for *every* request they accept indefinitely. In practice, such versions are garbage collected (see Section 7.7). This weakens the liveness guarantee achieved.

Garbage collection removes well-formed established candidates once later well-formed established candidates exist. "Slow" clients that perform read operations concurrent to write operations and garbage collection may reach the earliest candidate still in each server's history without classifying any candidate as complete. Such a read operation could either abort or retry; a retried read may also not complete.

With bound storage capacity, R/W-PF members with an asynchronous timing model achieve obstruction-freedom [Herlihy et al., 2003]. Lemma 6.2.3 relies on all well-formed established candidates existing indefinitely. For example, the initial candidate, $\langle \mathbf{0}, \perp \rangle$ is relied upon to ensure read operations complete.

So long as a client performs a read operation in isolation (i.e., no other concurrent read or write operations by other clients take any steps) or no server performs garbage collection during the read operation, the read operation completes. For R/W-PF members with a synchronous timing model, so long as garbage collection delays δ before deleting "unneeded" candidates from its history, read operations complete.

6.3 Discussion of malevolent components

The linearizability provided by R/W-PF members that tolerate malevolent clients does not guarantee that such clients write valid object values. Linearizability ensures that malevolent clients write a single value to the object (i.e., poisonous writes are masked). However, malevolent clients can write arbitrary values to the object. Read/write objects do not provide stronger semantics.

A replicated state machine [Lamport, 1978; Schneider, 1990] ensures that object values transition based on specified functions. Moreover, a replicated state machine may expose a narrow interface that limits the manner in which a client may manipulate an object's value. Replicated state machines require an atomic broadcast primitive. Read/write objects provide weaker semantics than replicated state machines and so malevolent clients may modify the object value arbitrarily.

An application that stores data using R/W-PF objects can perform integrity or semantic checks on object values. Such checks could provide some protection from malevolent clients. The comprehensive versioning performed by R/W-PF members also provides an audit trail of client actions that may be useful in detecting aberrant values written by malevolent client.

The wait-freedom guarantee with unbounded storage capacity and the obstruction-freedom with bounded storage capacity does not guarantee good performance. Malevolent clients can perform intentionally incomplete writes that require additional round trips by subsequent correct clients to read. Server verification of writes can provide a bound on the number of such writes a malevolent client performs (see Section 7.7.1 for more discussion). Malevolent clients and servers can make timestamps extremely large thus incurring additional network and storage costs. Protecting against such timestamp attacks is discussed in Section 7.8. Malevolent components may even perform denial of service attacks on specific objects by issuing an inordinate number of requests to specific servers.

The Byzantine failure model is characterized by a lack of assumptions about actions faulty components may take, whereas estimating performance in the face of malevolent components requires assumptions about the frequency and type of attacks. Such models of malevolent components are outside of the scope of this work; the performability of Byzantine fault-tolerant systems seems to be an open research topic. In practice, intrusion tolerance techniques based on statistical models of "normal" behavior may be necessary to provide timeliness in the face of malevolent components.

7 Prototype design and implementation

This chapter describes salient aspects of the prototype system implementation of the R/W-PF. The prototype system implementation is called PASIS[1]. PASIS is implemented in C/C++ and runs on the Linux operating system. Some mechanisms and optimizations in the implementation that are not described in the design of the R/W-PF given in Chapter 4 are described in this chapter.

7.1 Client and server implementation

The majority of the client implementation resides in a client library. The client library routines are implemented such that different objects may have different universes of servers and different R/W-PF membership. For the sake of evaluating the prototype implementation, a client program, that links with the client library, was developed to read and write objects with different R/W-PF membership.

Servers expose the same interface, regardless of the protocol member being employed—write and read requests for all R/W-PF members are serviced by a single interface. In PASIS, the interface given for servers in Figure 4.7 is extended to include an object ID.

Each write request that is accepted results in a fragment being stored. A fragment is indexed by its object ID and by its logical timestamp. To be clear, fragments are not over-written, every "version" is retained. If a server receives a write request for an object it does not yet host locally, it creates the object automatically. Upon creation, the server inserts $\langle \mathbf{0}, \perp \rangle$ into

[1] In Wylie et al. [2000] PASIS was an acronym—it is now simply a state of mind.

its local history of the object. The server then processes the write request normally. If a server receives a read request for an object it has not created locally, it simply replies with $\langle \mathbf{0}, \perp \rangle$. Object creation is discussed further in Section 7.4.

Servers expose additional interfaces that allow clients to specify the number of timestamp entries from the local history to return with read requests. In the implementation, a server includes the two latest timestamps in its reply to a read-latest request, as well as the fragment corresponding to the latest timestamp (witnesses are discussed in Section 7.5). In a failure-free execution, isolated read operations identify a complete candidate in a single phase of client-server communication. If there is read-write concurrency, the additional timestamp from server histories provides more information with which to classify a complete candidate. If a client classifies a candidate as complete from this additional information, the client may still have to retrieve fragments to actually decode the object. Servers support an interface for reading a specific fragment by its timestamp. Clients can also specify the number of timestamp entries from the history to return in a read-previous request.

For correctness in the crash-recovery benign server-failure model, a server must store a candidate it accepts in stable storage before responding to the client. Disks are slow relative to many other components in computer systems. Waiting for candidates to be stored to disk is not desirable. If servers employ non-volatile RAM, then servers can respond much more quickly. Volatile RAM that is battery-backed can serve as non-volatile RAM. Computers with uninterruptible power supplies (UPS) that store their memory contents to disk when they switch to the backup power also have non-volatile RAM.

The PASIS server is based on the Comprehensive Versioning File System (CVFS) code base [Soules et al., 2003]. The CVFS code base is based on the Self-Securing Storage Systems (S4) code base [Strunk et al., 2000]. The PASIS server has two modules: the back end and the front end. The back end server implementation uses a log-structured organization to reduce the cost of comprehensive data versioning [Rosenblum and Ousterhout, 1992]. The back end server has not been significantly modified from CVFS for PASIS.

The PASIS front end server is heavily modified from CVFS for PASIS. The PASIS front end exports the PASIS server interface rather than an NFS interface. In addition, there are specialized caches for object histories in the front end that include only timestamps.

In the code base, the client library is called PASISIO (in the Ursa Minor code branch, it is called SSIO). The client program is called PASISIO_BENCHMARK. The PASIS server front end program is called PASIS_FE_S4 and the back end is called S4_DRIVE.

7.2 Reliable channels

Client-server communication is implemented over TCP/IP sockets via remote procedure calls (RPCs). For R/W-PF members with a synchronous timing model, a timeout δ is specified. In the prototype implementation, δ is set at compile time and has a default value of 1 s. A short timeout facilitates experimentation and demonstrating the system; in practice δ should be set to be tens of seconds. However, δ is currently also used for the retry frequency for quorum probing. The PASIS prototype could be extended to allow the timeout to be specified on a per-object basis (and potentially even per-access basis), and to have different parameters for δ and for quorum probing.

Using TCP/IP for client-server communication in the implementation is an imperfect match for the asynchronous timing model and for the crash-recovery benign server failure model. TCP/IP uses timers to control the flow of network traffic and to define a period after which it stops retransmitting packets. The former use of a timer is safe, since it is a performance hint. The latter use of a timer is problematic for asynchronous R/W-PF members. There is a background process in PASIS that attempts to re-establish connections to servers that timeout once a second. This mechanism is relied upon by asynchronous R/W-PF members to retry indefinitely. However, such an implementation, at best, only meets a partially synchronous timing model [Dwork et al., 1988]. Only during periods of relative synchrony will responses be received.

To handle the crash-recovery benign server failure model, reliable channels are achieved by repeatedly sending requests to servers until sufficient responses are received. TCP/IP retransmits dropped packets, and so implements a reliable unidirectional channel in the crash-recovery failure model. Client-server RPCs, as implemented, retry sending packets until TCP indicates that all packets comprising a request have been delivered. If the server does not send a response to the request (via a separate RPC), the client does not resend a request until a timeout expires. The PASIS prototype wraps TCP/IP with another level of timeouts and retries to implement the reliable channel.

The channels implemented by the PASIS prototype are close enough to the timing and failure models that empirical measurements of PASIS are meaningful. However, there is room for additional engineering in the design and development of a network stack that meets the needs of all members of the R/W-PF while providing stable throughput and good responsiveness. Specifically, full compliance with the asynchronous timing model and a more refined timeout and retry policy would improve the prototype implementation.

7.2.1 Quorum remote procedure calls

Quorum-based techniques are often advocated for their ability to disperse load among servers and their throughput-scalability [for example: Naor and Wool, 1998; Wool, 1998; Malkhi et al., 2000]. In storage systems, however, there are performance benefits gained by locality of access. In the context of the R/W-PF, locality of quorum access may be beneficial: a read operation serviced by the same quorum of servers that serviced the most recent write operation will likely complete in a single round of communication. We implement quorum access locality by mapping object IDs to *preferred quorums*.

An access strategy based on preferred quorums does not directly offer the load dispersal or throughput-scalability associated with quorum systems. If many objects are accessed over the entire universe of servers, load can be balanced among the servers. Indeed, we do not expect quorum access strategy to be used to disperse load or provide throughput-scalability. Because different objects can be placed in different quorum systems that do not share servers, we expect the placement of objects on servers to be the primary means of balancing load among servers.

The PASIS prototype ignores corrupt responses from malevolent servers. This matches the client pseudo-code for **c_qprc_read_latest** in Figure 4.8 and for **c_qrpc_read_previous** in Figure 4.9, that ignores replies with corrupt fragments (cf. lines 909 and 1009). By ignoring the response, the client effectively transforms the malevolent fault to a crash. For asynchronous timing models, additional servers are probed, and for synchronous timing models, such servers return TIMEOUT once δ expires.

If the client "detects" that a server is malevolent, it could safely stop sending requests to that server in the future. Moreover, it could log the information for the purposes of system administration (although if clients may be malevolent, such information may not be trustworthy). The PASIS prototype implements neither of these features.

The quorum probing policy implemented in PASIS is fairly simple. Requests are initially sent to the servers in an object's preferred quorum. Preferred quorums are a deterministic function of object ID. If a server in the preferred quorum does not respond in a timely manner (i.e., δ), requests are broadcast to all servers in the object's universe. For asynchronous R/W-PF members, this means that probing is retried every δ. For synchronous R/W-PF members, this means that a TIMEOUT response is assigned to any pending server requests once δ has elapsed.

7.3 Authorized channels and access control

Clients and servers have pair-wise shared secrets. We assume some infrastructure is in place to distribute shared secrets among clients and servers.

The PASIS prototype supports an existing Kerberos [Steiner et al., 1988] infrastructure.

Each RPC request and response is authenticated. Authenticated channels for R/W-PF members that tolerate malevolent components incur some additional compute cost because an HMAC [Bellare et al., 1996] is used. For such R/W-PF members, we rely on the hash in the timestamp to vouch for the fragment. This means that each byte of the request is only hashed once. For R/W-PF members that do not tolerate malevolent components, the HMAC is replaced with the client ID.

Clients are either authorized, or not. If authorized, clients can perform reads and writes of an object. Access control is considered orthogonal and so is not implemented. However, we note that requiring that a client be able to repair a candidate during the course of a read operation could complicate access control: should a client authorized only to read an object be able to repair the object? If read and write privileges are differentiated, then a client that can only perform read operations can not perform repair and thus may not be able to complete a read operation.

To limit client access, but retain strong liveness guarantees, additional mechanisms may be employed. Two such mechanisms are servers digital signing the candidates they return in response to client read requests or servers being authorized to perform repair on behalf of under-privileged clients. The former allows a client to prove to a server that there exists a repairable candidate at some timestamp. This is similar to the use of authenticators with replica histories in the Query/Update protocol [Abd-El-Malek et al., 2005a]. If malevolent clients are tolerated, then this approach is only effective for replicated objects. The latter authorizes servers to contact other servers to directly determine if a repairable candidate indeed exists at the timestamp sent by the client. This is similar to on-demand verification which is discussed in Section 7.7.1.

7.4 Object metadata

For simplicity and modularity of the implementation, metadata as to where objects are stored is assumed to be the responsibility of another component (e.g., the client application, a directory service, etc.). In PASIS, all clients have full knowledge of the entire universe of servers $\mathcal{U}$ and of every object they read and write.

Every object is uniquely identified by an object ID. For an object, its universe of servers U, R/W-PF member, and object ID is static and set at object creation time. The client that performs the first write operation to an object creates the object and determines its universe, R/W-PF membership, and object ID.

Malevolent client actions during create are not tolerated by the PASIS prototype. For example, a malevolent client could re-use an object ID and create the "same" object with different R/W-PF membership at different servers. Extending the prototype so that object names are self-verifying—for example, by including the universe, membership, and object ID in the object name—would protect against malevolent object creates. However, protecting against malevolent object creation could be done, in some other fashion, by another component in a complete storage system (e.g., a volume manager or directory service). We did not implement any specific mechanisms in the prototype.

The PASIS prototype does, however, include a structure that lists the object's universe and its R/W-PF membership in each write request. This structure, because it describes the object's universe, is similar to a *linkage record* [Amiri et al., 1999]. Linkage records, as used by Amiri et al., list the set of servers that accepted the most recent write.

7.5 Quorums, witnesses, and the universe

In the PASIS prototype each server in the universe $\mathcal{U}$ has an ID. As well, each server in a sub-universe (i.e., an object's universe) has an ID. Fragments written by a client have an index in the range $(1, \ldots, q, \ldots, n)$. The fragments with indices $1, \ldots, q$ are stored in the preferred quorum. The PASIS prototype supports read and write witnesses. Timestamps serve as witnesses in the implementation.

Read witnesses are selected by the client at the time of a read operation. The client sets a flag in each request it sends indicating whether the server should respond with a candidate (timestamp and fragment pair) or just a timestamp. Clients select the servers that host fragments with indices $1, \ldots, m$ to return candidates and the remaining $q - m$ to return read witnesses. For erasure-coded data, the "first" m servers correspond to those that host stripe-fragments. For R/W-PF members that support only client crash failures these fragments require no computation to decode. Otherwise, if the self-validating timestamp must be checked, it does not matter which m servers return fragments.

Whereas each client can select which servers act as read witnesses independently, the set of servers that host only write witnesses (timestamps) are specified as part of the R/W-PF member. This is necessary to ensure correctness for R/W-PF members that tolerate malevolent clients. If clients independently select which servers host write witnesses, a malevolent client could select too many servers for hosting write witnesses. Subsequently, some clients may complete a read operation of the candidate so "established", while others may not. As such, linearizability would be compromised. This is a variant of a poisonous write. To protect against malevolent clients, in the implementation, the only servers that can host write witnesses are those servers that host fragments with the indices $q - q_w + 1, \ldots, q$. Entries in the cross checksum for such fragments are assigned the null value $\perp$.

The solution implemented limits the ability to employ write witnesses if additional quorums are probed. A solution that allows clients to select the servers that host write witnesses at the time of the write operation is to embed a write witness flag (bit) in the cross checksum. The client cannot change the set of servers that host write witnesses after it has constructed the timestamp (cross checksum). However, this at least allows a client to select servers that it believes are up to host write witnesses; in the face of server failures, this may be useful.

Many quorum constructions can be used in conjunction with the R/W-PF [for example: Malkhi and Reiter, 1998, 2000]. We have implemented the threshold (majority) quorum constructions described in Section 5.2. We have also implemented aspects of recursive threshold quorum constructions based on these threshold quorums [Malkhi and Reiter, 2000]. The recursive threshold quorums are sufficient for evaluating common case performance, but quorum probing is not implemented. (The probing implementation exclusively works with threshold quorums currently because it simply probes for a specific number of responses, without consideration of whether the set of responses is from a quorum.) To handle general quorum constructions, the quorum probing implementation must be revised to consider the quorum membership servers that respond.

For a single object, only its U of servers matters. However, since many objects are stored in a storage system, there are interesting questions of layout policy. This is true for a set of objects that share the same U and for sets of objects that span all servers $\mathcal{U}$.

Consider RAID 5 which "rotates" a parity fragment among a fixed set of servers. To achieve a similar layout using the R/W-PF, consider a set of n objects with the same U and with server IDs in the range $(1, \ldots, n)$. Each object stores its fragment with index 1 on a different server, and then its fragment with index 2 on the next server, with the n^{th} server wrapping around to the first. Given the access policy for reads, that selects the "first" m servers to return candidates, this layout disperse the load among all the servers in the universe (assuming a uniform object load).

This approach to "rotating parity" extends to the case of multiple objects with distinct overlapping universes. To be concrete, consider 10 objects, each with its own sub-universe of 5 servers, in an entire universe of 10 servers. If each object's universe consists of 5 "consecutive" servers and starts with a different server, then each of the 10 servers hosts 5 objects. Moreover, the load is dispersed among all 10 servers (again, assuming uniform object load). There are similarities between this layout policy and chained declustering [Lee and Thekkath, 1996; Thekkath et al., 1997]. The prototype implementation supports these rotation layout policies.

Although $\mathcal{M}$ is general enough to describe RAID schemes that conflate layout and redundancy (i.e., RAID 1/0, RAID 5, and RAID 6), such specifications unnecessarily complicate quorum construction. The prototype implementation treats the layout of erasure-coded fragments as a separate task. This works well for m-of-n threshold erasure codes and results in a flexible system design.

Some researchers have considered the merit of random layout policies versus partition layout policies. That is, should the number of distinct universes be minimized? or, maximized? The former decreases the likelihood that failures beyond the minimum tolerated result in data loss, but it increases the amount of data lost if such an event occurs. Douceur and Wattenhofer [2001] considered such a layout problem in the context of a decentralized file system, and van Renesse and Schneider [2004] considered a similar layout problem in the context of establishing Chain Replication server groups. The PASIS prototype can be configured to implement many layout policies. Layouts that rotate redundant fragments can be specified as part of the R/W-PF member. However, random layouts must be specified in configuration files.

7.6 Extensions for experimentation

The PASIS prototype includes extensions to facilitate testing and evaluation. Much of the code is instrumented and the data collected is printed out once programs complete. In instrumenting the code, we have avoided unnecessary slowdown of the code path. We use the Pentium register counter (RDTSC) to measure the duration of various events in the system. We also avoid instrumenting inner-most loops of compute-intensive functions (e.g., encoding and decoding functions).

Features are added to both the client and server code so that faults can be injected. To experiment with crash servers, servers can be forced to "crash" with a SIG INT command. Servers that receive such a signal shut down cleanly: they close open connections and they print out statistics. By closing the connections, client-side timeout code is circumvented. As such, this is more like a fail-stop failure than a crash failure.

Servers have command line options for omission faults and malevolent faults. Servers run with omission faults enabled respond as though they accept write requests but never insert them into their history. Such servers return $\langle \mathbf{0}, \perp \rangle$ in response to every read request. Servers run with malevolent faults enabled accept write requests but respond to all read requests with a random fragment. Such a fragment does not match the cross checksum in the timestamp. Clearly, such command line options should not be part of a production version of the R/W-PF.

Clients have command line options for stuttering failures and for poisonous writes. A client specified to stutter performs writes that do not complete. The command line option allows the type of stutter (incomplete or repairable) and the number of such writes to be specified. A client specified to perform poisonous writes generates fragments by randomly filling in the fragment buffer. Such a client "correctly" constructs the timestamp, including the cross checksum. The number of poisonous writes to perform is specified on the command line.

7.7 Garbage collection

Garbage collection is necessary to prevent capacity exhaustion at servers. Servers can delete a candidate from their local history, if the candidate has a timestamp that is less than that of the well-formed established candidate with the highest timestamp—referred to as the *latest complete candidate*. However, a server, in isolation, cannot determine which candidates in its local history meet this requirement. To determine the timestamp of the latest complete candidate, servers perform a read operation.

Garbage collection is implemented in the current prototype and it requires no additional RPCs. There are optimizations in the interface though to make garbage collection more efficient. For R/W-PF members that do not tolerate malevolent clients, there is no need to read candidates. Servers can perform garbage collection by exclusively reading timestamp histories.

The cost of sending garbage collection requests is amortized by batching such requests together: a range of object IDs is specified in the read request. For R/W-PF members that do not tolerate malevolent clients or servers, only a single server need perform a read; it can then *notify* other servers of the timestamp of the latest complete candidate. The *responsibility* for garbage collecting specific object IDs is distributed among servers. This reduces the potential for redundant work being done by servers.

In the prototype, garbage collection is triggered during idle time and on-demand by memory exhaustion. A simple idle-time detector [Golding et al., 1995] is used to detect idle periods, during which garbage collection is initiated. If the front end is low on memory, it must either move candidates from memory to disk, or garbage collect candidates in memory. The front end maintains a *high write count* table that tracks how many candidates (versions) each object has in memory. Such objects are prioritized by garbage collection, because they free up the most capacity when successfully garbage collected.

Garbage collection requires server-to-server communication. Most quorum-based protocols avoid such communication. However, the alternatives are to rely on a trusted "client" that identifies versions that are safe to garbage collect or to introduce verifiable information dispersal techniques that are not network-efficient. Garbage collection requests ought to be amortized over many versions and potentially many objects. Moreover, garbage collection can be scheduled during otherwise idle periods. For these reasons, server-to-server communication seems like the right approach in practice.

7.7.1 Lazy verification

For R/W-PF members that tolerate malevolent clients, determining the latest complete candidate is more cumbersome. Servers must perform normal read operations so that they can see if the timestamp is self-validating. However, the PASIS prototype implements *lazy verification* for such R/W-PF members [Abd-El-Malek et al., 2005b]. Laziness refers to the fact that servers do not verify write operations until there is idle time or memory exhaustion.

For R/W-PF members that tolerate malevolent clients and servers, multiple servers are assigned responsibility for each object ID. For threshold quorums, $b+1$ servers are assigned primary responsibility, and b additional servers secondary responsibility. So long as the primary servers send notification messages in a timely fashion that are consistent with one another, the secondary servers do nothing. Otherwise, secondary servers also perform verification, until $b+1$ matching notify messages are sent. This *co-operation* reduces the expected number of messages for verification, relative to each server performing verification independently.

Once a candidate is verified, candidates with prior timestamps can be garbage collected. Moreover, servers set a flag in subsequent read responses indicating that a candidate has been verified. If a client classifies a candidate as complete and observes that at least $b+1$ servers have flagged the candidate as verified, the client concludes that the candidate is well-formed. This reduces the client compute for decoding such candidates. Servers also garbage collect candidates that are not well-formed; this potentially reduces the number of clients that perform verification on the candidate. Finally, to bound the number of poisonous writes in the system at any one time, or the number of intentionally incomplete candidates, servers can be set to limit the number of unverified candidates on a per client basis.

Note that erasure codes that provide confidentiality from servers (e.g., secret sharing or short secret sharing) cannot be garbage collected using these techniques: to perform garbage collection, some servers must decode the object. Some other trusted entity is necessary to perform garbage collection of such objects and maintain server confidentiality.

7.8 Timestamps and chronological chicanery

The PASIS prototype reserves 8 bytes for $Timestamp.LogicalTime$. In theory, 2^{64} write operations is a lot. In practice, malevolent clients can generate arbitrary-sized values for $Timestamp.LogicalTime$ and exhaust the timestamp space. Moreover, malevolent servers can reply to correct clients with arbitrary-sized timestamps, forcing correct clients to write arbitrary-sized timestamps. Even if *Timestamp.LogicalTime* was of some large integer type that did not have a fixed size, arbitrary-sized timestamps are still problematic. Arbitrary-sized timestamps impact network- and space-efficiency; such timestamps could be many factors larger than the fragments being stored.

Bazzi and Ding [2004] identified a solution to arbitrary-sized timestamps in Byzantine fault-tolerant storage systems: *non-skipping timestamps*. To implement such non-skipping timestamps in the R/W-PF an additional quorum RPC that establishes the timestamp before the write is necessary. Moreover, to tolerate malevolent clients, servers would need to digitally sign timestamps returned to clients in response to **c_qrpc_read_time** requests. An additional phase of communication and the use of asymmetric cryptography are not desirable. Cachin and Tessaro [2005c] identify a similar technique for constructing timestamps that exhibit bound growth via a threshold signature scheme.

The PASIS implementation does not implement any mechanisms to protect against arbitrary-sized timestamps. However, we believe that the techniques implemented for lazy verification (See Section 7.7.1) with a minor modification can provide protection efficiently. Bazzi and Ding [2004] ignore the b highest responses to read timestamp requests when constructing the timestamp. This ensures that malevolent servers cannot force a correct client to construct an arbitrary-sized timestamp. A minor modification of **c_qrpc_read_time** can provide this feature. To protect against malevolent clients, servers could perform timestamp verification on-demand if they receive a write request with "too large" a timestamp. The concept of "too large" could be based simply on the highest timestamp in a server's local history. Allowing the timestamp element *Timestamp.LogicalTime* to increment by more than 1 at a time may avoid additional, unnecessary, server messages during periods of contention.

Malevolent servers can also attack the performance of read operations via the timestamp. A malevolent server can respond to a read-latest request with a forged candidate that has a high timestamp. In the pseudo-code for **c_read**, a client issues read-previous requests if the candidate set is classified incomplete. The malevolent server can then respond with a forged candidate that has an incrementally smaller timestamp. Because timestamps include cryptographic digest in the low bits, a malevolent server has a large timestamp range with which to forge candidates.

A client can protect itself from this attack by selecting the timestamp for read-previous requests differently. Instead of basing the read-previous timestamp on the candidate classified incomplete, the client bases the timestamp on another candidate in the response set. As is done for non-skipping timestamps, the b highest timestamps are ignored. This is safe, because the candidates corresponding to such timestamps are classified incomplete. Responses to such read-previous requests actually return candidates with timestamps *equal* to or less than the specified timestamp.

7.9 Synchronized clocks

Loosely synchronized clocks can often be used as performance hints in distributed systems [Liskov, 1991]. To reduce the number of quorum RPCs a write operation that synchronous R/W-PF members require, such members use loosely synchronized clocks to generate timestamps for writes. However, the use of loosely synchronized clocks weakens the linearizability guarantee.

In the synchronous timing model, client clocks may be synchronized using the Network Time Protocol [Mills, 1992, 1995]. Given loosely synchronized client clocks, there is no need to perform a **c_qrpc_read_time** quorum RPC. Therefore, line 1300 of the of the function **c_write** in Figure 4.11 becomes $LogicalTime$:= `C_LOCAL_READ_TIME`().

In practice, client clocks drift—each client's local clock may progress at (slightly) different rates—which is why they must be continuously synchronized, and why they are considered only *loosely* synchronized. As such, at any time, every client clock may exhibit a different amount of drift relative to global time. We define clock skew τ to be the maximum difference between the local clocks for any two correct clients. Note that τ is not directly observable; it is an assumption about the efficacy of the clock synchronization protocol. Because of clock skew, the definition of when a write operation begins (cf. Definition 6.1.2) is modified.

Definition 7.9.1 (*synchronized write begin*). A write operation $w_{Timestamp}$, that uses the function `C_LOCAL_READ_TIME` to determine its logical timestamp, *begins* τ before when a benign client invokes the **c_write** operation locally that issues a write request bearing timestamp *Timestamp*.

Revising the begin time weakens the safety guarantee. Consider two clients, A and B, whose clocks differ in that A's clock is τ less than B's clock. It is necessary to extend the begin time of a write operation to accommodate the case when B invokes a write operation less than τ before A invokes a write operation. If B establishes a candidate before A invokes its write operation, then B's write operation should precede that of A. Lemma 6.1.11 proved that the timestamp order is a total order on write operations. However, A's write operation is ordered before B's write operation. Extending the begin time of the write operation into the past, as Definition 7.9.1 does, "fixes" this problem by allowing write operations that begin within τ of one another to be linearized.

In many systems, such as cluster-based storage in a high-speed LAN, clock skew is expected to be much less than one millisecond. Weakening linearizability in this fashion, halves the number of phases of client-server communication for write operations. The "right" decision depends on the object/system in question. If the R/W-PF is being used in the context of a system that serializes access to objects, i.e., its failure-atomicity is utilized, but not its concurrency atomicity, then using a local clock to construct timestamps may not weaken the linearizability guarantee. The implementation supports either using local clocks or using a quorum RPC on a per-object basis (i.e., it is specified as part of R/W-PF membership).

As discussed in Section 7.8, malevolent clients can write "into" the future. Timestamps based on local clocks introduce another concern regarding malevolent clients. Correct clients that base timestamps on the local clock will write behind candidates with arbitrarily high timestamps.

To protect against such a malevolent fault, a correct client can ignore (classify as incomplete) a candidate with a timestamp that is τ greater than its local clock. If servers, as well as clients, have synchronized clocks, then servers can reject write requests that are "in the future". Given this feature, clients do not need to perform additional logic to classify "future" writes as incomplete. Note that a server or client with an unsynchronized clock (i.e., its drift is more than τ from real time) is malevolent.

7.10 Encode and decode

The IDA implementation and a number of cryptographic primitives are based on publically available code collected in the Crypto++ library [Wei Dai, 2005]. In PASIS, the erasure code implementation is separated from the specification of m and n. However, there are default erasure codes selected depending on m and n: if $m = 1$, replication is used; if $m = n - 1$, RAID 4 is used; otherwise, IDA is used. These defaults can be over-ridden, so that, for example, IDA is used with $m = 1$. For IDA, the implementation of the Galois Field used may also be specified (see Section 7.11).

For replicated objects, cross checksums are constructed in a special manner. Since replicated data is self-verifying, it is sufficient for a single collision resistant hash to be used in lieu of one for each fragment. This reduces the computation and space required to protect against poisonous writes.

Beyond the IDA implementation, Shamir's secret sharing [Shamir, 1979], Krawczyk's short secret sharing [Krawczyk, 1994], and Blakley's ramp schemes [Blakley and Meadows, 1985] are also implemented. Various ciphers are implemented for use with short secret sharing: DES, 3DES, and AES. The DES and 3DES implementations are based on the Crypto++ library and the AES implementation is based on publically available code [Gladman, 2004a]. A pseudo random number generator based on ANSI X9.17C is implemented for use with secret sharing and is based on the Crypto++ library.

Various cryptographic digests are implemented for constructing cross checksums: MD5 [Rivest, 1992] and SHA1 [NIST, 1995], as well as some other SHA variants. We note that MD5 is showing its age [Wang et al., 2004], however, it is useful for comparing performance trends across cryptographic primitives. The implementation of MD5 is based on publically available code [Rivest, 1992]. The implementation of SHA is also based on publically available code [Gladman, 2004b].

The PASIS prototype can use any of these cryptographic digests in the construction of cross checksums. However, the implementation currently uses compile time flags to select which cryptographic digest to use. Run-time specification, for example as part of R/W-PF membership, would facilitate experimentation. Different cryptographic digests are of different lengths, for example, MD5 digests are 16 B and SHA1 digests are 20 B. The functions that serialize data structures for client-server RPCs need to know the exact size of structures. To make cross checksums based on smaller digests more network-efficient, the digest is selected at compile time. The compile time default is for SHA1 and that is the hash function used by R/W-PF members that require collision-resistant hashes.

Because hashes become less secure over time, archival data stored via the R/W-PF may need to be migrated to stronger hashes over time. Although the R/W-PF does not directly support such an action, performing a read with the "old" hash and then a write with the "new" hash should be easy to implement in most storage systems. Selection of the collision resistant hash could also be part of the R/W-PF member specification. Weaker hashes could be used for short-lived data and stronger hashes for long-lived data.

In the implementation, because timestamps are used as indices at servers and witnesses in read and write requests, the cross checksum is not actually part of the timestamp. Only a collision resistant hash of the cross checksum is in the timestamp: the element $Timestamp.CrossChecksum$ in the implementation is actually $\mathbf{hash}(CrossChecksum)$. The cross checksum is sent with write requests because servers must validate the integrity of the fragment against its entry in the cross checksum. In addition, servers must validate that the hash of the cross checksum matches the digest for it in the timestamp. In the implementation, servers that respond to read requests with candidates also return the cross checksum. Servers that return only read witnesses return the timestamp without the cross checksum. A client receives sufficient cross checksums in this manner.

7.11 IDA implementation

The implementation of IDA in the prototype is based on polynomial interpolation in a Galois Field. This conceptually follows Rabin's Information Dispersal Rabin [1989] and Shamir's Secret Sharing Shamir [1979]. The implementation of polynomial interpolation is loosely based on the Crypto++ library of Wei Dei Wei Dai [2005]. The source is modified to use stripe-fragments (i.e., like RAID 4 rather than RAID 3) and more efficient implementations of Galois Fields of size 2^8 were added.

7.11.1 Example of polynomial interpolation in $\mathbb{Z}_7$

To illustrate erasure coding as polynomial interpolation, consider the prime field of integers of size seven $\mathbb{Z}_7$. To perform a 3-of-6 erasure-coding of an object comprised of three elements $\{3, 4, 0\}$ from $\mathbb{Z}_7$, these three elements are considered three points in $\mathbb{Z}_7 \times \mathbb{Z}_7$: $\{(0, 3), (1, 4), (2, 0)\}$. Three points in a space with distinct x coordinates define a polynomial of order two. More generally, m elements define a polynomial of order $m-1$. The three points in this example define the polynomial $y = x^2+3$ in $\mathbb{Z}_7$. This polynomial is used to generate the redundant $n-m$ (i.e., $6-3$) fragments: $\{(3, 5), (4, 5), (5, 0)\}$. Figure 7.1 illustrates this process graphically.

7.11.2 Lagrange polynomial interpolation

Given m points on a polynomial, Lagrange's polynomial interpolation can efficiently generate additional (redundant) points:

$$P(x) = \sum_{j=0}^{n-1} P_j(x)$$

$$P_j(x) = y_j \prod_{\substack{k=0 \\ k \neq j}}^{n-1} \frac{x - x_k}{x_j - x_k}$$

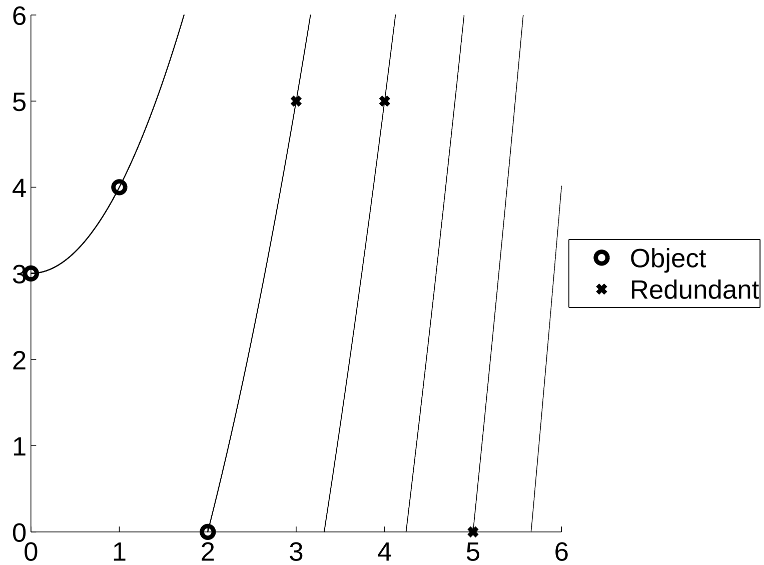


Figure 7.1. Illustration of a 3-of-6 erasure coding of the three element object $\{3, 4, 0\}$ via polynomial interpolation in $\mathbb{Z}_7$.

For the example from Section 7.11.1, this expands to,

$$\begin{aligned}
P(x) &= y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \\
&+ y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \\
&+ y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \\
P(x)_{(x_0,x_1,x_2)=(0,1,2)} &= 3\frac{(x-1)(x-2)}{(0-1)(0-2)} + 4\frac{(x-0)(x-2)}{(1-0)(1-2)} + 0\frac{(x-0)(x-1)}{(2-0)(2-1)} \\
P(x)_{(x_0,x_1,x_2)=(0,1,2)} &= \frac{3}{2}(x-1)(x-2) - 4x(x-2)
\end{aligned}$$

And so, interpolating three redundant fragments results in the following:

$$
\begin{aligned}
P(3) &= \frac{3}{2}(2)(1) - 12(1) = -9 \equiv 5 \mod 7 \\
P(4) &= \frac{3}{2}(3)(2) - 16(2) = -23 \equiv 5 \mod 7 \\
P(5) &= \frac{3}{2}(4)(3) - 20(3) = -42 \equiv 0 \mod 7
\end{aligned}
$$

To erasure code objects "in bulk", objects are broken into m stripes. Each stripe-fragment is the same size (the m^{th} stripe-fragment is padded if necessary). Each stripe-fragment is broken into elements from some field (in this example $\mathbb{Z}_7$). Each stripe-fragment is essentially an array of elements. Polynomial interpolation is performed across elements with the same offset into the different stripe-fragment "arrays". Each such interpolation is based on elements with the same ordinal positions (x intercepts) and generates redundant elements for the same set of ordinal positions (x intercepts). Constants that are a function solely of the ordinal positions of object data and redundant data are pre-computed for efficiency.

Consider the object $\{3, 1, 4, 2, 0, 5\}$. It is broken into stripe-fragments $\{3, 1\}, \{4, 2\}, \{0, 5\}$. To generate three redundant fragments, the elements $\{3, 4, 0\}$ and $\{1, 2, 5\}$ are interpolated. The pre-calculation of constants is based on these elements having the ordinals $\{0, 1, 2\}$ and the redundant fragments having the ordinals $\{3, 4, 5\}$. There are m constants for each redundant fragment. The three constants for the first redundant fragment with ordinal 3 are:

$$
\begin{aligned}
P(3) &= y_0 \frac{(3 - x_1)(3 - x_2)}{(x_0 - x_1)(x_0 - x_2)} \\
&\quad + y_1 \frac{(3 - x_0)(3 - x_2)}{(x_1 - x_0)(x_1 - x_2)} \\
&\quad + y_2 \frac{(3 - x_0)(3 - x_1)}{(x_2 - x_0)(x_2 - x_1)} \\
P(3) &= y_0 \frac{(3-1)(3-2)}{(0-1)(0-2)} + y_1 \frac{(3-0)(3-2)}{(1-0)(1-2)} + y_2 \frac{(3-0)(3-1)}{(2-0)(2-1)} \\
P(3) &= 1(y_0) - 3(y_1) + 3(y_2)
\end{aligned}
$$

$x^{\{1,\ldots,7\}}$	mod x^3+x+1	**Binary**
x^1	x	010
x^2	x^2	100
x^3	$x+1$	011
x^4	x^2+x	110
x^5	x^2+x+1	111
x^6	x^2+x+1	101
x^7	1	001

Table 7.1. Galois Field (2^3) based on generator x and irreducible polynomial x^3+x+1.

As such,

$$P(3)_{(y_0,y_1,y_2)=(3,4,0)} = 1(3) - 3(4) + 3(0) = -9 \equiv 5 \mod 7$$
$$P(3)_{(y_0,y_1,y_2)=(1,2,5)} = 1(1) - 3(2) + 3(5) = 10 \equiv 3 \mod 7$$

The pre-calculation of such constants requires $O(m^2)$ operations. Each redundant fragment generated via polynomial interpolation given such constants requires $O(m)$ operations. Lagrange's polynomial interpolation is covered in detail by Knuth [1997].

7.11.3 Example of multiplication in GF(2^3)

In this section, we review and extend an example, developed by Stinson, of the construction of and multiplication in GF(2^3) [Stinson, 1995, page 182]. A finite field is constructed from a generator. A Galois Field requires a generator and an irreducible polynomial. The irreducible polynomial for this example is x^3+x+1 and x is the generator. Additional background on finite fields and Galois Fields can be found in Menezes et al. [1996]. Table 7.1 shows the field defined by this irreducible polynomial.

In Galois Fields, addition is based on logical XOR which is computationally very efficient. There are many distinct techniques for efficiently imple-

	000	001	010	011	100	101	110	111
000	000	000	000	000	000	000	000	000
001	000	001	010	011	100	101	110	111
010	000	010	100	110	011	001	111	101
011	000	011	110	101	111	100	001	010
100	000	100	011	111	110	010	101	001
101	000	101	001	100	010	111	011	110
110	000	110	111	001	101	011	010	100
111	000	111	101	010	001	110	100	011

Table 7.2. Full lookup table for multiplication in GF(2^3).

menting multiplication in a Galois Field. The appropriate technique depends both on the size of the Galois Field and on the type of processor being used.

Lookup tables are used to implement multiplication efficiently There are two options for lookup tables: a full lookup table (e.g., $2^3 \times 2^3$), or log and anti-log lookup tables. Table 7.2 illustrates the full lookup table for this example, and is taken almost directly from Stinson [1995, page 182]. Tables 7.3 and 7.4 illustrate the log and anti-log lookup tables for this example. The log lookup table follows from Table 7.1 except for the log of zero. The log of zero is set to 15 so that the anti-log table can be constructed to account for multiplication by zero. This construction of the anti-log table avoids checking for multiplication by zero inline in the code, so that multiplication by zero does not require special case code.

7.11.4 Multiplication in GF(2^8), GF(2^{16}) and GF(2^{32})

The implementation of information dispersal can operate in GF(2^8). Either a full lookup table or a log lookup table can be used. The irreducible polynomial $x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1$ is used to generate the field. The full lookup table is 64 KiB in size. By performing the multiplication required for polynomial interpolation in a specific order only some rows of the full lookup table, each of which is 256 B in size, must be in memory. At most, one row for each of the pre-computed constants must be in memory. To

Index	**Log**
000	15
001	7
010	1
011	3
100	2
101	6
110	4
111	5

Table 7.3. Log lookup table for multiplication in GF(2^3).

Index	**Log**$^{-1}$	**Index**	**Log**$^{-1}$	**Index**	**Log**$^{-1}$	**Index**	**Log**$^{-1}$
0	-	8	010	16	000	24	000
1	-	9	100	17	000	25	000
2	100	10	011	18	000	26	000
3	011	11	110	19	000	27	000
4	110	12	111	20	000	28	000
5	111	13	101	21	000	29	000
6	101	14	001	22	000	30	000
7	001	15	000	23	000	-	-

Table 7.4. Anti-log lookup table for multiplication in GF(2^3).

erasure-code an object, at most $(n - m) \times m$ rows of the table must be in memory, since $n - m$ redundant fragments are generated, each of which requires m pre-computed constants. The memory footprint of the log and anti-log tables is smaller: 256 B for the log table and 1023 B for the anti-log table. The full lookup table requires fewer instructions than the log/anti-log tables to perform multiplication.

For small n, we have found the full lookup table to be most efficient, and for larger n the log and anti-log tables to be most efficient. We have found that unrolling a loop of the bulk polynomial interpolation four times and aligning addition (logical XOR) to four byte word boundaries helps some compilers optimize the code for the class of processors we use.

Information dispersal can also be performed in larger fields: $\mathrm{GF}(2^{16})$ and $\mathrm{GF}(2^{32})$ in the implementation. The irreducible polynomials used for these fields are $x^{16} + x^5 + x^3 + x + 1$ and $x^{32} + x^7 + x^3 + x^2 + 1$ respectively. Lookup tables in such fields are not efficient because they are too large to be accessed efficiently in current processors (just the log table is 64 KiB in size for $\mathrm{GF}(2^{16})$). Multiplication in a Galois Field greater than 2^8 is implemented via a tight loop from 16 or 32 down to 0. Only shift, logical AND, logical XOR, addition, and lookup in a small table of pre-computed values (the modulus of the irreducible polynomial, the multiplier shifted left by one, and the multiplier shifted left by one added to the modulus of the irreducible polynomial) are used in the loop.

We have found that polynomial interpolation in $\mathrm{GF}(2^8)$ is more efficient than in Galois Fields with a higher degree. However, the degree of the Galois Field places a limit on n, the number of unique erasure-coded fragments. In $\mathrm{GF}(2^8)$, only 256 unique erasure-coded fragments can be generated.

8 Evaluation

This chapter describes the results of experiments performed with the PASIS prototype. Sections 8.3 and 8.4 present the empirical results that demonstrate that the PASIS prototype is a versatile storage infrastructure. Different R/W-PF members provide substantially different performance, depending on their specified resiliency model and storage mechanisms.

The R/W-PF is optimistic and so offers best performance for concurrency- and failure-free accesses. Performance results of accesses in the face of concurrency and failures are described in this chapter. The R/W-PF is quorum-based and so, in theory, can benefit from the throughput-scalability offered by quorum constructions. Experiments were performed to investigate the throughput-scalability of quorums and witness use in the prototype. The performance of the IDA implementation used in the prototype is also described in this chapter.

8.1 Experimental setup

8.1.1 Testbed

All experiments are performed on a rack of 76 Intel Pentium 4 2.80 GHz computers, each with 1 GiB of memory, and an Intel PRO/1000 NIC. The computers are connected via an HP ProCurve Switch 4140gl that has a specified internal bandwidth of 18.3 Gbps/35.7 mpps. The computers run Linux kernel 2.6.11.5 (Debian 1:3.3.4-3).

On a computer that acts as a server, a single server process PASIS_FE_S4 is run in most experiments. On a computer that acts as a client, a single client process PASISIO_BENCHMARK is run. The client process is allowed to have multiple operations outstanding at any time. For response time experiments, we limit client processes to a single outstanding operation. For throughput experiments, we allow client processes to have multiple outstanding operations. It is necessary to allow multiple outstanding requests per client so as to saturate the network or servers in throughput experiments. The limit on the number of outstanding operations allowed is described in each experiment. We do not allow client processes "think time". However, because client computation is required to encode and decode objects, scheduling at client processes with multiple operations has the effect of acting like client "think time".

8.1.2 Repeatability

Unless otherwise noted, experiments are run for 15 seconds and measurements are taken during the middle 5 seconds. Experiments are run 5 times and the mean is reported. Some experiments do not use this procedure (e.g., the fault injection experiments in Section 8.5). The procedures for such experiments are described with the results.

To ensure that experiments can be consistently re-run, we developed a set of perl scripts that start the servers, start the clients, wait for the clients to finish, stop the servers, and collate results. The *control script* runs on a single computer, but it initiates client and server processes on many other computers. Indeed, some experiments involve up to sixty computers.

First, the control script starts all of the server processes for an experiment. After sleeping for three seconds, the control script starts client processes. The sleep period allows all servers to start before clients begin issuing requests. In most cases, the sleep step is unnecessary. However, the sleep step tolerates the occasional jitter observed when forking tens of processes that connect via SSH to server machines.

Second, the control script starts all of the client processes for an experiment. The control script waits for all client processes to return. Each client runs for the specified 15 second experiment duration. This time is broken into three phases: warmup, measurement, and cooldown. The intention of the warmup and cooldown phases is to ensure that all clients are operating during the measurement phase. The measurement phases for every client mostly overlap, but some start earlier (end later) than others based on the delay between the control script starting the first client process and the last client process. Experiments that involve only a single client (for example, response time experiments) do not require such long warmup and cooldown phases. However, to keep the experimental method consistent, such experiments are run in the same manner as experiments with multiple clients.

Finally, once all of the clients return, the control script stops all of the server processes. Client and server processes print statistics for the experiment to log files that the control script collates and synthesizes. The control script analyzes log files for error messages and collates any such messages in another file.

We ran a set of experiments designed to ensure that the duration of the experiment and of each experiment phase was sufficient to produce meaningful results. We ran experiments in which each phase of the experiment was ten times longer than the standard (i.e., 50 seconds). We also ran experiments in which individual phases were ten times longer than the standard. In all cases the results matched within one percent of the response time and throughput measures from the standard setup. This gives us confidence that the experiments are being run for long enough to collect valid data.

A client takes additional measurements during the last two seconds of the warmup phase. From these measurements, the client identifies a mean and standard deviation. It uses these to construct "bins" for data collected during the measurement phase. The "bins" provide a simple histogram of the data collected during the measurement phase. Such a histogram was found to be useful during performance tuning/debugging of the implementation (and of some experimental setups).

We use the Pentium register counter (RDTSC) to measure the duration of various events in the system. The number of processor clock cycles measured by the register counter is converted to wall clock time. The conversion factor is based on an initialization function that calls GETTIMEOFDAY. To determine the consistency (repeatability) of this conversion factor, we perform the initialization function one hundred times. To determine the maximum range of the conversion factor resulting from the initialization function, we divide the difference between the maximum observed conversion factor and the minimum observed conversion factor by the minimum observed conversion factor. This gives a range in seconds per second for the conversion factor. We did this experiment five times. The consistency of the conversion factor was found to be 0.0000321 (i.e., 32.1 μs/s). What this measurement means, is that if two experimental runs were identical in every aspect *except* for the initialization of the conversion factor, we would expect the reported measurements to be within 0.003%.

8.1.3 Methodology

Almost all experiments focus on in-memory performance. Timestamps and fragment versions are not synced to disk. Disk performance is highly dependent on workload (e.g., locality of access) and hardware (e.g., the number and type of disks at each server). The focus of this thesis is not the performance of a specific storage system implementation for a specific workload on specific hardware: the focus is versatility.

Unless otherwise noted, the working set for each experiment fits in memory. In-memory performance allows us to measure the performance of different R/W-PF members without having to account for workload effects that change with the R/W-PF member. Garbage collection (and lazy verification) is also workload dependent and so is suppressed for most experiments.

To allow experiments to run for long enough to collect statistically significant results, servers must delete versions during an experiment. Otherwise, many experiments would exhaust server memory. Servers are run with the *oracle garbage collection* option enabled: servers only retain the latest two versions of any fragment. This allows common case (concurrency- and failure- free) performance measurements to be taken.

In most experiments a fixed object of 32 KiB is specified. In some experiments a fixed fragment size is specified. In such experiments, the object size is calculated from the fragment size. Specifying a fixed fragment size makes the network traffic and server load generated by R/W-PF members with different space-efficiencies (i.e., different m) more alike.

Most experiments, perform either exclusively read operations or exclusively write operations. Note that there is a "format" phase that creates objects in both cases, so that read operations have objects to read, and write operations are performed on previously created objects. Since read operations are different than write operations, performing separate experiments allows the costs specific to each type of operation to be measured.

In theory, the performance for a workload that mixes read and write operations is simply the weighted sum of the performance of each individual type of operation. In practice, a "feature" of the Ethernet driver throttles the number of interrupts it can generate for bi-directional TCP flows[1]. Such throttling of bi-directional traffic results in lower performance for mixed workloads than for homogeneous workloads. We compiled a series of custom kernels with different constants for the Ethernet driver. Circumventing the adaptive algorithm and setting a constant interrupt limit provided the anticipated performance: the performance of a mixed workload was observed to simply be the sum of the parts. However, the custom kernels achieved poor throughput stability under load (i.e., throughput peaked and then dropped as more load was added). We decided to run experiments with the stock kernel: we appreciate that we got what we paid for, but have faith that the Ethernet driver in future Linux kernels will exhibit better behavior.

Given the nature of the R/W-PF, read operations that are concurrent to write operations may require multiple quorum RPCs to complete. Additionally, server failures may trigger timeouts and quorum probing, and malevolent components can attack performance. However, the majority of experiments are performed concurrency- and failure-free. As such, best-case performance is reported. For many workloads and deployments though,

[1]Search for the variable `itr` in the code module `e1000main.c`. The adaptive algorithm that throttles the number of interrupts the driver can generate can range from 8000 to 2000. We are indebted to James V. Hendricks and Gregg Economou for their assistance. James had previously identified this feature and Gregg compiled many kernel variants for us to explore this feature.

Policy	**Description**
Benign	Tolerates clients that crash and benign server failures (i.e., $b = 0$).
Hybrid	Tolerates clients that crash, a single malevolent server, and possibly additional faulty servers (i.e., $b = 1$).
Malevolent	Tolerates clients that crash and malevolent servers (i.e., $b = t$).
Malevolent^{+}	Tolerates malevolent clients and servers (i.e., $b = t$).

Table 8.1. Failure models for R/W-PF members evaluated. The failure models are listed in order of increasing generality.

concurrency- and failure-free access to read/write objects is expected. Specific experiments are performed to quantify the cost of access during concurrency and failures.

8.1.4 Exploration of the "versatility-space"

To focus the exploration of R/W-PF versatility, we limit the set of R/W-PF members evaluated. Unless otherwise specified, the R/W-PF member for an experiment employs the smallest sized threshold quorum construction given the resiliency model and erasure code specification. In many sets of experiments, measurements are reported for different values of t, the number of faulty servers tolerated. For a given t, we identified three axes of freedom on which to focus the evaluation of R/W-PF versatility:

- **Failure model policies.** Table 8.1 lists the different policies used to select client and server failure models for the R/W-PF members evaluated.

- **Timing model policies.** Table 8.2 lists the different policies used to select timing models for the R/W-PF members evaluated.

- **Space-efficiency policies.** Table 8.3 lists the different policies used to select m for the R/W-PF members evaluated.

Policy	Description
Synchronous	Synchronous timing model with crash benign server failures. Message delays and processing are assumed to be bound. Faulty servers are either malevolent or crash. Loosely synchronized client clocks are used to construct logical timestamps.
Synchronous&CR	Synchronous timing model with crash-recovery benign server failures. Message delays and processing are assumed to be bound. Faulty servers are either malevolent, unstable, or eventually down. Even though message delays are bound, the client may have to repeatedly probe until a quorum of servers is live. However, synchrony allows for loosely synchronized client clocks that are used to construct logical timestamps.
Asynchronous	No assumptions about timeliness are made. Logical timestamps are constructed by reading the latest time from a quorum of servers.

Table 8.2. Timing models for R/W-PF members evaluated. The timing models are listed in order of increasing generality. For the synchronous timing model, two policies are listed. The policies differ in the benign server failure model, which affects the size of quorums.

Policy	**Description**
Default	For the default space-efficiency, m is selected based on n. Given the value of n for the R/W-PF member, the largest value of m permitted is selected. This policy generates the most space-efficient R/W-PF member for the specified n.
Replication	Regardless of how large m could be, this police always selects replication for the R/W-PF member (i.e., $m = 1$). This policy uses the same value of n as for the Default policy.
Constant	An erasure code with $m = 6$ is selected for the R/W-PF member. In some cases, this necessitates that n be increased beyond what the resiliency model demands. In other cases, this results in the same value of n as for the Default policy.
Fault-based	This policies selects the space-efficiency for the R/W-PF member based on the number of faulty servers tolerated: $m = t + 1$. This policy is interesting because the number of servers n increases as t increases. As such, this policy improves the space-efficiency of the R/W-PF member as more servers are needed. In some cases, this necessitates that n be increased beyond what the resiliency model demands. In other cases, this results in the same value fo n as for the Default policy.

Table 8.3. Space-efficiencies for R/W-PF members evaluated.

These policies are selected to provide an intuitive framework for exploring the inherent trade-offs in performance and cost resulting from different resiliency model choices and storage mechanism choices. These inherent trade-offs motivate the need for a versatile storage infrastructure. The ability to explore these trade-offs in a single storage infrastructure demonstrates that the R/W-PF provides versatility.

The failure policies cover the range of failure models likely to be employed. Moreover, the failure policies are selected to highlight interesting costs. For example, comparing Hybrid and Benign shows the cost of tolerating any malevolent servers; comparing Malevolent and Hybrid shows the cost of tolerating all malevolent servers; and comparing Malevolent and Malevolent^{+} shows the cost of tolerating malevolent clients. The timing policies cover the range of interesting timing models. The Synchronous and Asynchronous policies are the two extremes. The Synchronous&CR policy represents an interesting middle ground: a Synchronous&CR member has the same bounds as an Asynchronous member but completes writes in a single round trip.

The space-efficiency policies are selected to illustrate interesting interactions between the erasure coding specification and other aspects of the R/W-PF member. For example, comparing other space-efficiency policies with Replication shows the value of erasure codes. Comparing Replication with Constant is especially effective, because both maintain a constant value for m as the number of faults tolerated increases. As more faults are tolerated, both policies yield higher blowup. However, the blowup for Constant increases at a lower rate than Replication. Replication is also interesting because data is self-verifying which simplifies ensuring its integrity against malevolent clients (i.e., the Malevolent^{+} failure model). Comparing the Default and Fault-based space-efficiency policies is interesting, because for Benign and Hybrid failure models, Default requires fewer servers than Fault-based and is less space-efficient.

The versatility-space is explored using the cross product of all of these policies. Both response time and throughput are explored. The results of the exploration, in some sense, are not "surprising" and the exploration does not identify which R/W-PF member is "right" for a given task. The exploration simply demonstrates that different resiliency model and space-efficiency policy choices result in different performance. The important results of this exploration are that different choices have different resiliency, performance, and cost, and that all experiments are performed using a single storage infrastructure, demonstrating that the storage infrastructure is versatile.

The results of the versatility-space exploration would be different on a different testbed. For example, the ratio of network bandwidth to client CPU and to server CPU could lead to different valuations of the merit of space-efficient erasure codes. This does not change the result that the R/W-PF enables the construction of a versatile storage infrastructure. The hardware used in a specific deployment determines the expected performance of specific R/W-PF members. This is no different than hardware determining the performance of any other deployed storage system.

The versatility-space is explored in the absence of any other storage system services (e.g., directory service, volume management, etc.). Other services in a complete storage system influence the overall performance and resiliency of the storage system. There are a myriad of ways to implement other services and to combine a number of such services into a complete storage system. Such system-level decisions may limit the amount of useful versatility provided by the R/W-PF or impact the performance of R/W-PF members. For example, a lock service that tolerates only benign client failures cannot usefully exploit an R/W-PF member that tolerates malevolent clients. Moreover, such a lock service requires additional messages that influence overall performance. However, other storage services do not necessarily reduce the performance or versatility of the R/W-PF. Exploring the versatility of the R/W-PF in isolation demonstrates the potential versatility that can be exploited rather than that which a specific complete storage system realizes.

8.2 Encode and decode

Encoding and decoding objects requires client computation. This section reports the measured cost of performing cryptographic primitives on clients and servers. It also reports the measured cost of erasure-coding objects with the IDA implementation.

8.2.1 Cryptographic primitives

Table 8.4 lists the measured throughput of various cryptographic primitives on the testbed computers. Experiments are run 100 times on objects that are 8 KiB in size.

Basic memory operations `memcpy` and XOR are listed to provide context for the other results. Next, the measured throughput for random number generation is listed. The throughput for encryption primitives are based on encryption measurements, not decryption measurements. Finally, the throughput of various hash/checksum mechanisms are listed.

Even though some of these cryptographic primitives are not used by R/W-PF members directly, these results provide context for the costs of erasure coding. Moreover, erasure codes such as Shamir's secret sharing [Shamir, 1979] and Krawczyk's short secret sharing [Krawczyk, 1994], that require random number generation or encryption methods.

8.2.2 Erasure coding

We performed a series of three experiments to illustrate the performance trends of the IDA implementation as n and m vary. In these experiments, we report the minimum measured value based on running each IDA encoding 250 times. Reporting the minimum measured value clearly shows the salient compute cost trends, which is the intention of this experiment. The implementation of IDA we measured is described in Section 7.11. We report results for performing IDA in two different implementations of GF(2^8). In one implementation, a full lookup table is used for multiplication, and in the other, log and anti-log lookup tables are used for multiplication.

Encoding primitive	**Response time (ms)**	**Throughput (MiB/s)**
MEMCPY	0.0017	4565.05
XOR	0.0037	2085.13
x9.17c RNG	0.9968	7.84
DES	0.1767	44.22
3DES	0.5261	14.85
AES	0.1983	39.40
CRC32	0.0207	377.88
MD5	0.0276	283.12
SHA1	0.0530	147.29
SHA256	0.0948	82.39

Table 8.4. Throughput of encoding primitives.

In the first experiment, we measured the increase in the cost of performing IDA as n increases. In this experiment, we held m constant at 5, and increased n from 6 to 30. An object that is 5 KiB in size is erasure-coded. This object size is chosen to yield five stripe-fragments 1 KiB in size each. Each increase of n by one requires an additional 1 KiB redundant fragment to be generated. Figure 8.1 presents the result. The cost of erasure coding grows linearly as n increases. IDA based on full table lookup performs better than log table lookup in this experiment.

As described in Section 7.11.2, constants for Lagrangian polynomial interpolation can be pre-computed. These constants depend on the value of m and on which erasure-coded fragments are being generated. As such, they are pre-computed before each IDA encode or decode. In this first experiment, we measured the amount of the total IDA cost that was due to pre-computing these constants. Figure 8.2 shows the results. In all cases, the cost of the pre-computation is less than 3% of the total encode cost. We draw two conclusions from these measurements. First, the impact of pre-computing constants for every IDA encode and decode is not prohibitive. Some performance improvement would accrue from caching previous constants or hard-coding constants for common R/W-PF members. Second,

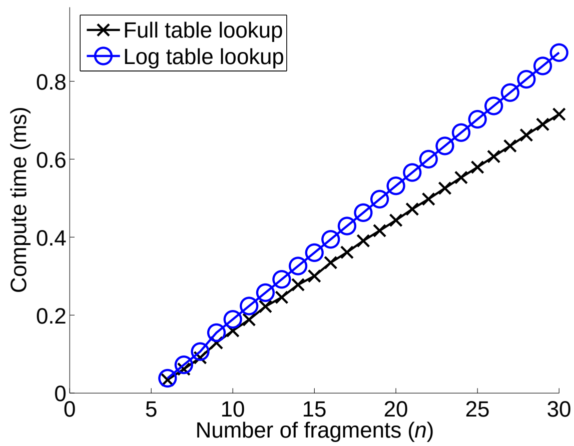


Figure 8.1. IDA encode time as fault-tolerance increases ($m = 5$).

pre-computing constants is clearly beneficial: computing the constants for each byte encoded with IDA would be expensive.

In the second experiment, we measured the increase in the cost of generating a single redundant fragment as m increases (i.e., we set $n = m + 1$ in this experiment). We increased m from 1 to 29 as the object size increased from 1 KiB to 29 KiB. As such, IDA encodes a single redundant fragment 1 KiB in size for each value of m.

Figure 8.3 shows the results. The cost of performing IDA increases linearly as m increases. This relationship was expected, since each byte of redundancy is generated as a function of m stripe-fragment bytes. There is an irregularity ("bump") in the linear relationship apparent from $m = 18$ to $m = 20$. We ran additional experiments to see if larger or smaller object sizes shifted this irregularity to lower or higher m; it did not. We suspect that this irregularity is is due to CPU cache efficacy decreasing as m increases.

In the third experiment, we measured the cost of IDA encoding as m increases for a fixed size 64 KiB object. We increased m from 1 to 29 and we set $n = m + 2$. As shown in the second experiment, as m increases, the per byte cost of erasure coding increases. However, as m increases, fragment size decreases and consequently the number of bytes of redundancy that must be generated decreases.

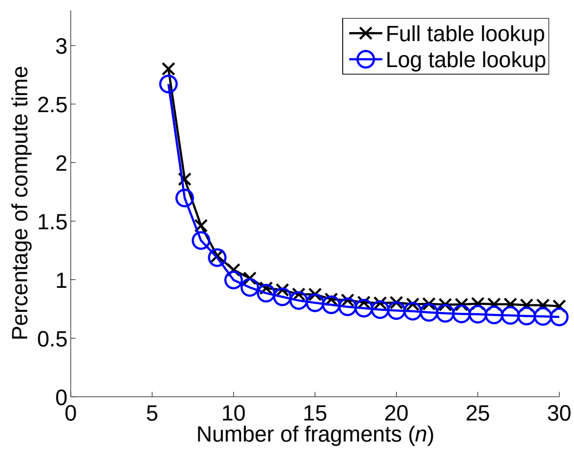


Figure 8.2. Cost of pre-computing Lagrange constanst. Percentage of total compute time for IDA encode that is due to pre-computing Lagrange constants as n increases.

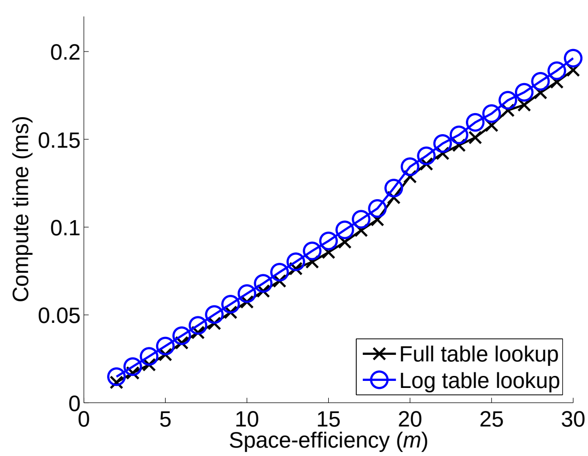


Figure 8.3. IDA encoding 1 KiB of redundancy as space-efficiency increases.

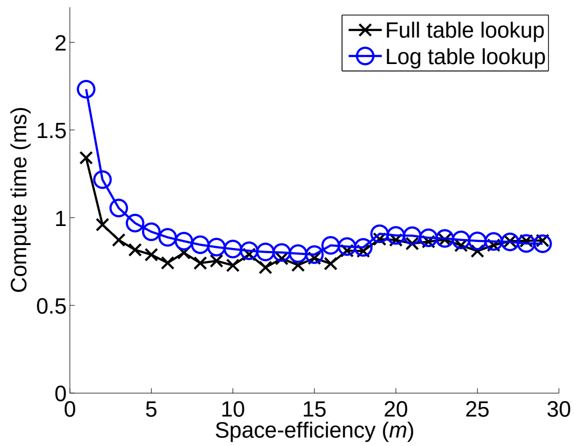


Figure 8.4. IDA encoding a 64 KiB object ($n = m + 2$).

Figure 8.4 shows the results. For $m > 5$, the cost of performing IDA remains reasonably constant. This means that the benefit due to increased space-efficiency is close to the increased cost of IDA due to increased m. For small m, the benefit of increased space-efficiency outweighs the increased cost of IDA. In practice, for $m = 1$ and $m = 2$ specialized erasure codes such as replication and RAID4 are used in the implementation. Notice that the "bump" from the second experiment is also in these results.

8.3 Resiliency and response time

We ran a series of experiments to determine how response time changes as more faults are tolerated. We ran experiments for $t = 0$ up to $t = 6$. We ran experiments for all of the possible combinations of failure model policy, timing model policy, and space-efficiency policy. We ran experiments to measure the response time of read operations and of write operations for 32 KiB objects.

We also ran a response time experiment for $t = 0$ with an object size of 4 B for the Benign/Synchronous/Replication R/W-PF member. This setup measures the cost of a single client sending a single request to a single server. We observed a mean response time of 247 μs for both read and write operations. This experiment effectively measures the "ping time" for the PASIS code base—any client-server communication takes at least this amount of time.

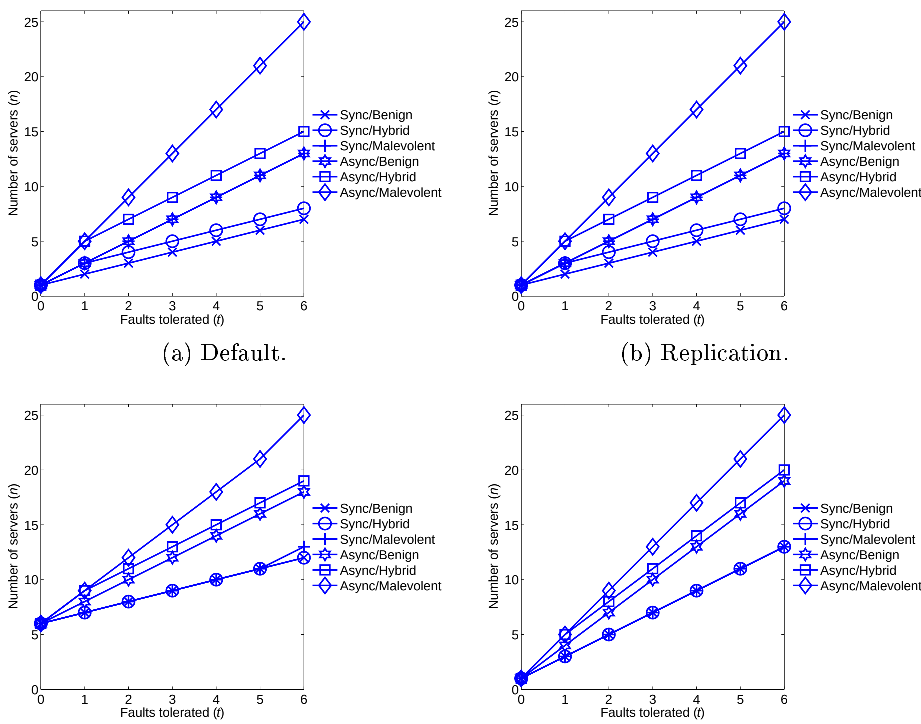
8.3.1 Fault-scalability, universe size, and blowup

The number of servers an R/W-PF member must contact, as well as its space-efficiency affect its response time. As such, we present graphs of n (the size of the universe) for the R/W-PF members evaluated in the remainder of the section. We also present graphs of $\frac{n}{m}$ (the blowup) for each member. These graphs are based on the specification of the R/W-PF members (i.e., these are not empirical measurements).

Figure 8.5 shows how the universe size increases as the number of faults tolerated increases for the various R/W-PF members. The Synchronous&CR timing model policy has the same trend as the Asynchronous timing model policy, and so is not shown. The Malevolent^{+} failure model policy has the same trend as the Malevolent failure model policy, and so is not shown. The results are separated by space-efficiency policy.

The universe size trends for the Default and Replication space-efficiency policies are identical (Figures 8.5a and 8.5b respectively). As follows from the constraints on quorum construction, the universe size required by Asynchronous R/W-PF members increases at a higher rate than for Synchronous R/W-PF members. Also following from the constraints on quorum construction, the universe size for R/W-PF members that tolerate the Hybrid failure model increases at a higher rate than for the Benign. And, the universe size for those that tolerate the Malevolent failure model increases at a higher rate then for the Hybrid. For $t = 1$, the Hybrid failure model is identical to the Malevolent failure model.

The Constant space-efficiency policy, shown in Figure 8.5c, provides space-efficiency for $m = 6$ regardless of how many faults are tolerated. As such, relative to the other space-efficiency policies, Constant requires a larger universe size for small values of t.



(a) Default. (b) Replication. (c) Constant. (d) Fault.

Figure 8.5. Space-efficiency policy: Universe size vs. faults tolerated.

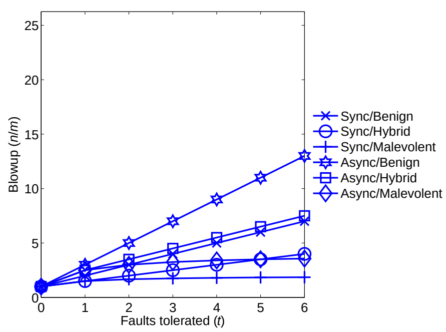
Figure 8.6 shows how the blowup changes as the number of faults tolerated increases for the various R/W-PF members. The blowup for each space-efficiency policy has a distinct trend as the number of faults tolerated increases. For the Default policy, shown in Figure 8.6a, $m = b + 1$, as such, the blowup is $\frac{n}{b+1}$. This results in Benign R/W-PF members having a higher blowup than Hybrid, which has a higher blowup than Malevolent. The blowup for the Replication policy depends exclusively on n. As such, Figure 8.6a is identical to Figure 8.5b which shows the universe size.

The Constant and Fault space-efficiency policies are more interesting. The blowup for the Constant space-efficiency policy, shown in Figure 8.6c, increases slowly relative to the Default and Replication policies. This is because $m = 6$ regardless of the value of t or n. The blowup for the Fault space-efficiency policy, shown in Figure 8.6d, approaches an asymptote as t increases. This is because the blowup for the Fault policy is $\frac{n}{t+1}$ and t is a linear function of t. The value of the asymptote depends on the timing model and failure model. Based on the quorum constructions, for all Synchronous R/W-PF members, the blowup approaches 2, regardless of the failure policy. For Asynchronous members that are Benign or Hybrid, the blowup approaches 3; whereas, for Malevolent members, the blowup approaches 4.

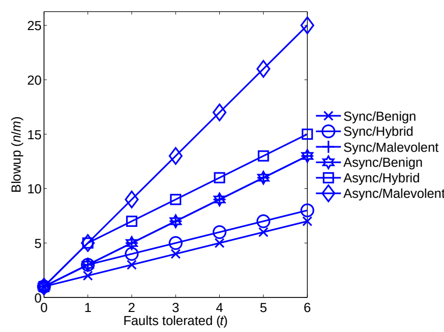
8.3.2 Response time and failure model policy

Response time measurements for the Default space-efficiency policy are shown in Figure 8.7. All combinations of timing model policy and failure model policy are shown. Response time measurements are shown for read and write operations. Notice the y-axis scale is different for read and write operations.

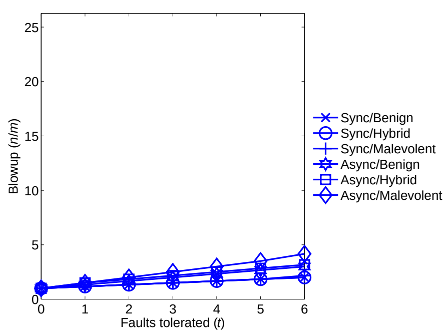
Read operation response times are shown in Figures 8.7a, 8.7c, and 8.7e. Consider the read response time at $t = 0$. In all of these experiments, this corresponds to a single server returning a 32 KiB fragment to the client. In theory, it takes 240 μs to transfer 32 KiB on a gigabit per second network link. The measurements match our expectations: 240 μs of bandwidth delay



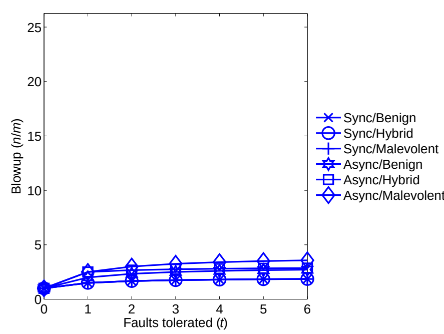
(a) Default.



(b) Replication.



(c) Constant.



(d) Fault.

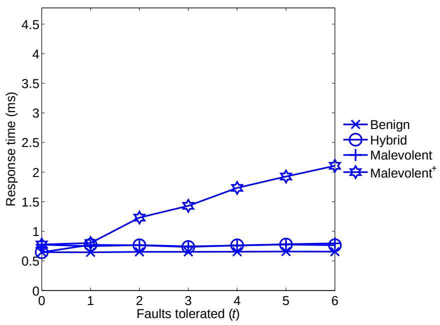
Figure 8.6. Space-efficiency policy: Blowup vs. faults tolerated.

in addition to the 247 μs latency yields the observed $\sim$500 μs response time for Benign members. Hybrid and Malevolent members incur client compute costs in addition to this.

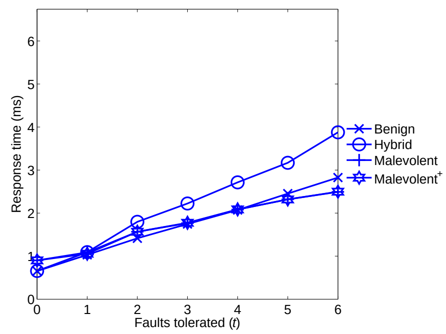
The response time of read operations for all R/W-PF members increases linearly as the number of faults tolerated increases. Indeed, for most R/W-PF members shown, the response time of read operations is relatively constant as t increases. All R/W-PF members employ read witnesses. As such, all read operations have near "perfect" space-efficiency: m fragments are read, totalling 32 KiB of data transferred over the network. Only read operations for Malevolent$^+$ R/W-PF members increase significantly in cost as t increases. This is because such members must validate the timestamp. The difference in read response time between Malevolent and Malevolent$^+$ members is client decode time.

Write operation response times are shown in Figures 8.7b, 8.7d, and 8.7f. The response time of write operations for Synchronous members, shown in Figure 8.7b, all increase at a similar rate regardless of the failure policy. Remember, for the Default space-efficiency policy, $m = b + 1$, and so the blowup of Synchronous, Malevolent and Malevolent$^+$ members increases more slowly than the Hybrid and Benign members (cf. Figure 8.6a). However, the network-efficiency gains of more space-efficiency erasure coding, comes at the cost of additional compute to perform IDA. As such, the Benign, Malevolent, and Malevolent$^+$ members have similar response times. The Hybrid member incurs additional compute cost, but is not as network-efficiency as the Malevolent members for more than one fault tolerated. As such, it exhibits the highest response time.

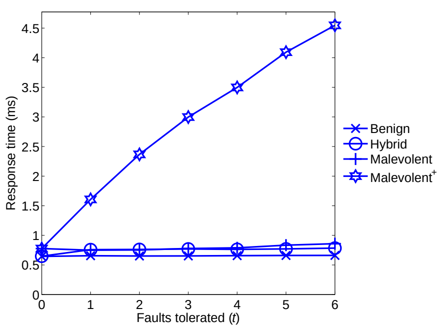
The response time of write operations for the Synchronous&CR and Asynchronous members exhibit very similar trends. The difference between Figures 8.7d, and 8.7f is the quorum RPC required by the Asynchronous members to read the logical time. Within either of these graphs, the write operation response time increases more slowly for the Benign failure policy, than the other failure policies, as the number of faults increases. The client compute required to generate cross checksums increases as the number of faults tolerated increases, as does the cost of erasure coding.



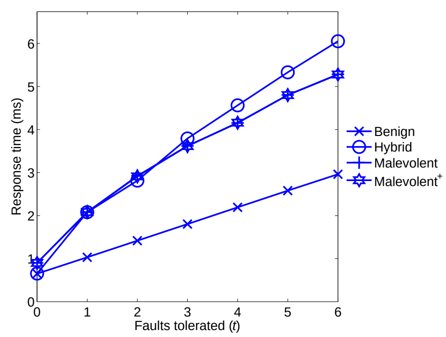
(a) Read/Synchronous.



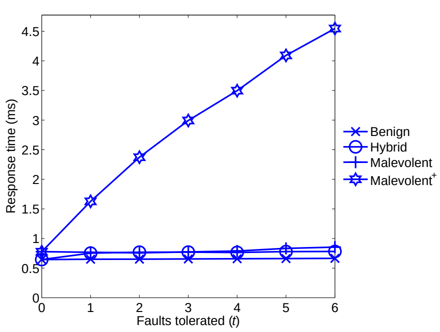
(b) Write/Synchronous.



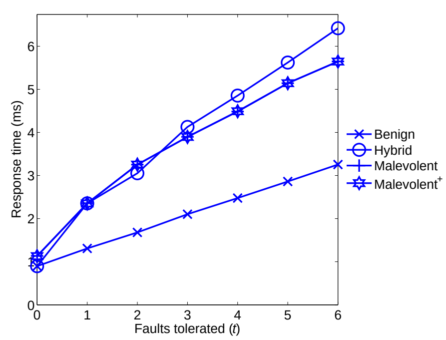
(c) Read/Synchronous&CR.



(d) Write/Synchronous&CR.

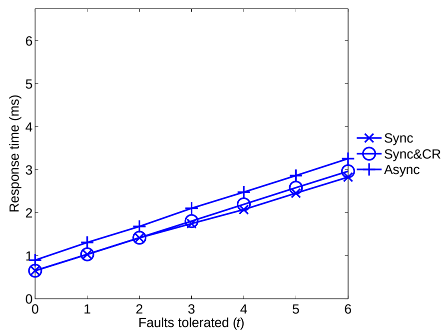


(e) Read/Asynchronous.

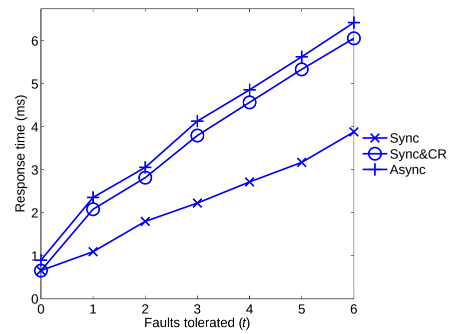


(f) Write/Asynchronous.

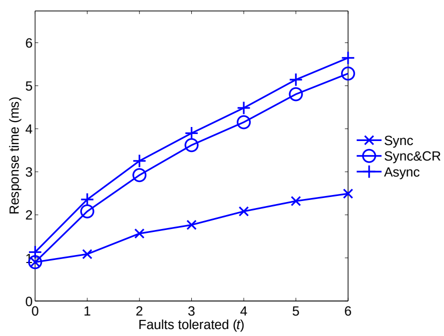
Figure 8.7. Timing model policy: Response time vs. faults tolerated.



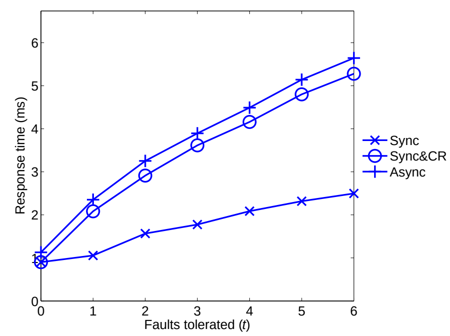
(a) Benign.



(b) Hybrid.



(c) Malevolent.



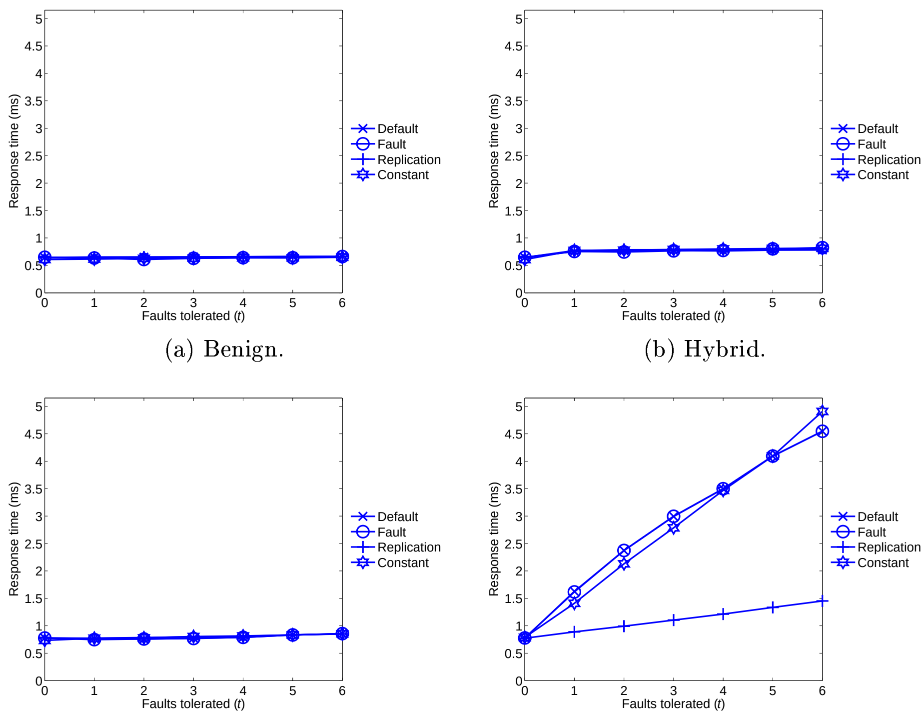
(d) Malevolent$^+$.

Figure 8.8. Failure model policy: Write response time vs. faults tolerated.

Figure 8.8 reproduces the write operation response time results from Figure 8.7. However, the results are plotted on distinct graphs for each of the failure model policies. This more clearly illustrates the costs associated with the choice of timing model. For all numbers of faults tolerated, and all failure policies, the Synchronous&CR and Asynchronous members are separated by a constant amount—approximately one quarter of a millisecond. Logical timestamp requests and responses are small, so this difference being close the minimum network latency is expected.

8.3.3 Response time and space-efficiency policy

In Section 8.3.2, only results for the Default space-efficiency policy are shown. Figure 8.9 shows the measured response times for read operations for



(a) Benign.

(b) Hybrid.

(c) Malevolent.

(d) Malevolent$^+$.

Figure 8.9. Space-efficiency policy: Read response time vs. faults tolerated.

different space-efficiency policies. As pointed out in Section 8.3.2, all R/W-PF members employ read witnesses and so are network-efficient. As such, the only R/W-PF members that experience an increase in read response time, as the number of faults tolerated increases, are Malevolent$^+$ members. The difference in read response time between Figures 8.9c and 8.9d is the compute cost of self-validating timestamps.

Notice how efficient the Replication Malevolent$^+$ member is relative to the other space-efficiency policies. As mentioned in Section 7.10, replicated objects are self-verifying, and so cross checksums are not used with replicated objects. The hash of the object is placed in the timestamp in lieu of the cross checksum. To validate such a read operation, it is necessary only to take the hash of the object. This is less compute-intensive than generating all n erasure-coded fragments and validating the timestamp.

Figure 8.10 shows the measured response times for write operations for different space-efficiency policies. As expected, write operations for Benign members, which do not compute any cross checksums, have lower response times than other members. The difference between the Malevolent$^+$ and Malevolent models, is the need to validate timestamps. This cost is incurred by read operations, and so Figures 8.10c and 8.10d are identical.

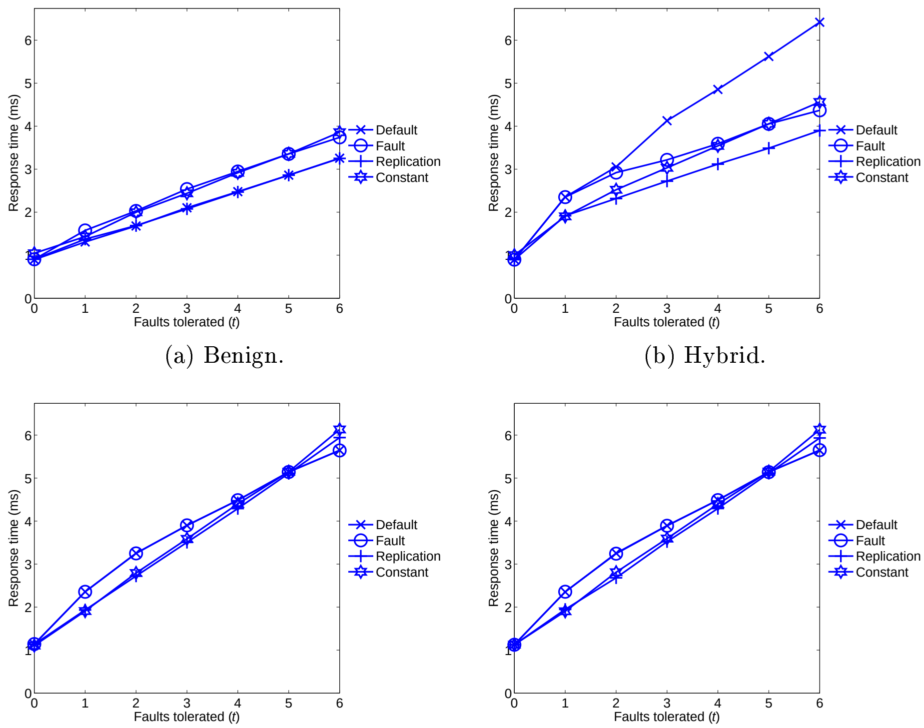
For write operations, the Hybrid failure policy has the most interesting relationship with the space-efficiency policy. The Default members have the highest write response times: they incur the cost of constructing the cross checksum to tolerate malevolent components, but do not become more space-efficient as t increases. For all values of $t > 0$, the space-efficiency is $m = 2$ for Hybrid members. The space-efficiency of Fault members increases with t and Constant members have $m = 6$. These lead to write response time increasing less slowly for such members. The increase in Replication members write response time is accounted for in network bandwidth delay rather than client compute.

8.3.4 Response time, universe size, and versatility

To draw attention to macroscopic trends, all of the unique write response time measurements are shown in Figure 8.11. Instead of plotting the write operation response time against the number of faults tolerated though, it is plotted against the size of the universe n. The Synchronous members, shown in Figure 8.11a, have lower response times and require fewer servers, then Synchronous&CR and Asynchronous members. As stated previously, regarding response time, the only substantive difference between the Synchronous&CR and Asynchronous members, is the round trip required by Asynchronous members to construct the timestamp.

8.4 Resiliency and throughput

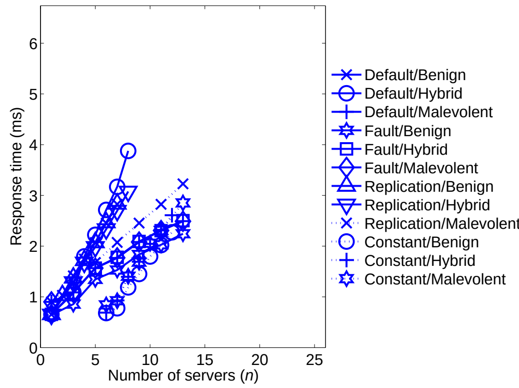
The response time measurements we present in Section 8.3 include client compute time, network delay, and server compute time. The throughput



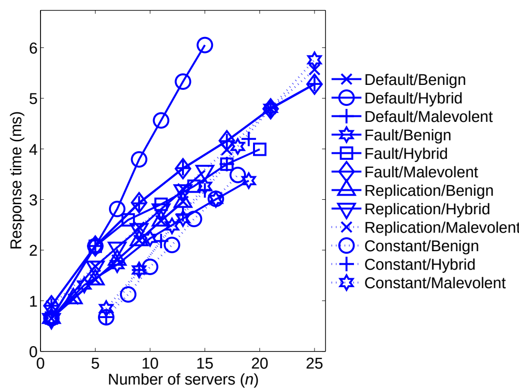
(a) Benign. (b) Hybrid.

(c) Malevolent. (d) Malevolent^{+}.

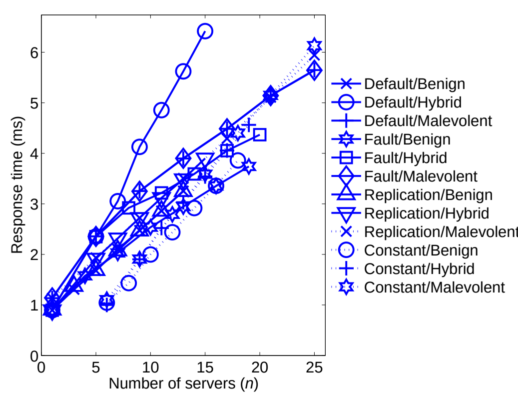
Figure 8.10. Space-efficiency policy: Write response time vs. faults tolerated.



(a) Synchronous.



(b) Syncronous with crash-recovery.



(c) Asynchronous.

Figure 8.11. Timing model policy: Write response time vs. universe size.

for a specific R/W-PF member that a set of servers provides is a function only of the latter two costs. Assuming there are sufficient client operations outstanding to keep all servers busy all of the time, then client compute does not affect throughput. We refer to this as *server-group throughput*, that is, the aggregate client perceived throughput that a set of servers can provide.

To determine the server-group throughput for different R/W-PF members, we ran a series of experiments with a server-group of size 12. We ran experiments for almost all of the possible combinations of failure model policy, timing model policy, and space-efficiency policy. The exception being that experiments were not run for Malevolent$^+$ members. The rationale for this is that the only difference between Malevolent$^+$ and Malevolent members is client compute time. We ran experiments for $t = 1$ and $t = 2$ for each member. For $t = 1$, we excluded Hybrid members, because they are identical to Malevolent members at $t = b = 1$. We ran experiments for read operations separate from write operations for each R/W-PF member.

We found throughput measurements to be sensitive to the size of requests, the number of client processes, and the number of outstanding operations per client process. In an effort to keep the profile of network traffic as similar as possible across R/W-PF members, we set the fragment size to be a constant 4 KiB. Thus, different R/W-PF members read and write objects of different sizes: objects are $m \times 4$ KiB in size.

The larger the quorum size for an R/W-PF member, the more server responses a client receives. This results in congestion on client receive links. Similarly, more client processes leads to more requests received by a server. This results in congestion on server receive links. Because different R/W-PF members have different quorum sizes in this experiment, we ran a set of experiments for each R/W-PF member over a range of number of client processes (15, 20, 25) and operations outstanding per client process (1, 2, 3). The best observed throughput is reported. These steps are taken to avoid unduly penalizing R/W-PF members for artifacts of TCP/IP.

Each of the R/W-PF members evaluated had a universe size less than or equal to 12. To take advantage of the entire server-group, each object, based on its object ID, was assigned to a universe of servers drawn from the server-group. The chained declustering layout described in Section 7.5 was used.

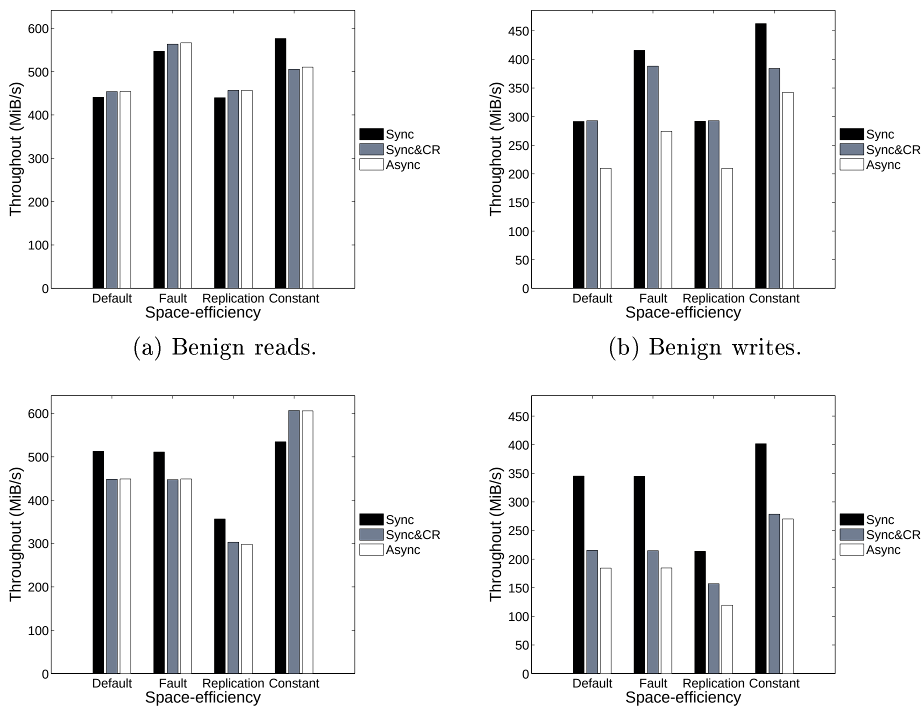
8.4.1 Throughput

The results of the throughput experiments performed with $t = 1$ are shown in Figure 8.12, and those with $t = 2$ are shown in Figure 8.13. Notice that the scale of the y-axis is different for reads and writes, as well as for $t = 1$ and $t = 2$. Observed read throughput ranges from 200 MiB/s for the Asynchronous, Replication, Malevolent, $t = 2$ member shown in Figure 8.13e, to 590 MiB/s for the Synchronous, Constant, Benign, $t = 1$ member shown in Figure 8.12a. Observed write throughput ranges from 80 MiB/s for the Asynchronous, Replication, Malevolent, $t = 2$ member shown in Figure 8.13f, to 460 MiB/s for the Synchronous, Constant, Benign, $t = 1$ member shown in Figure 8.12b.

In all cases, Synchronous members provide higher throughput than Synchronous&CR and Asynchronous members with the same failure and space-efficiency policies. The universe size for Synchronous members is smaller than for other members. Synchronous members layout more server-groups within the twelve servers and thus achieve higher throughput.

For read operations, Synchronous&CR and Asynchronous members achieve almost identical throughput for members with the same failure and space-efficiency policy. This is expected. Read operations are identical for these types of R/W-PF members. However, write operations differ. Synchronous&CR members use the local client clock to construct timestamps. This avoids a quorum RPC to determine the latest logical time. In turn, this reduces the load on the network which improves observed throughput. In all cases, Synchronous&CR members achieve better write throughput than Asynchronous members with the same failure and space-efficiency policy.

For read operations, the Fault space-efficiency policy achieves the same or better throughput than other R/W-PF members with the same failure and timing policy. Such members benefit from relatively small server-groups, while being more space-efficient than Default members.



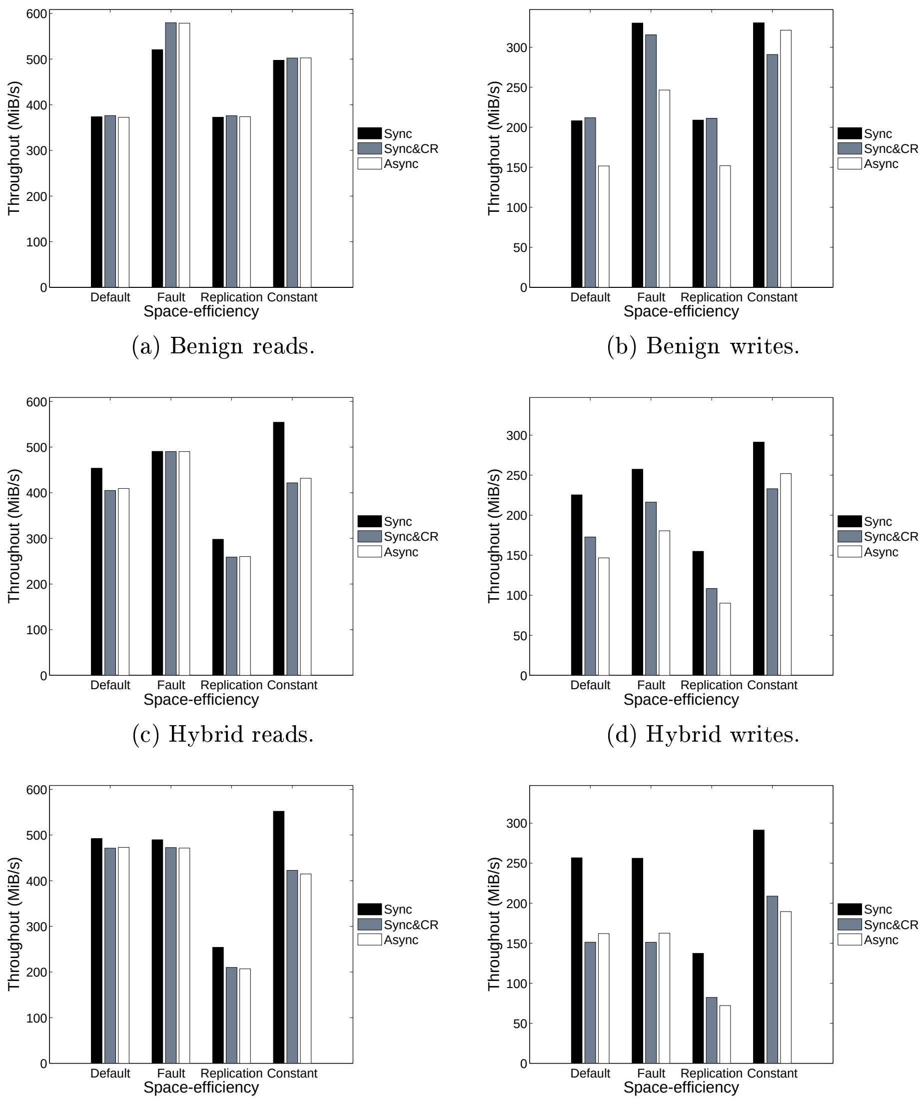
(a) Benign reads. (b) Benign writes.

(c) Hybrid and Malevolent reads. (d) Hybrid and Malevolent writes.

Figure 8.12. Twelve server case study with $t = 1$.

For write operations, the Constant space-efficiency policy achieves the same or better throughput than other R/W-PF members with the same failure and timing policy. The network-efficiency achieved by the Constant R/W-PF members is significant for write operations. Whereas, for read operations, because of read witnesses, network-efficiency is not as important for read throughput.

The experimental procedure may penalize Replication members. Because the fragment size is constant (and small) in these experiments, Replica members are essentially network-operation limited rather than bandwidth limited.



(a) Benign reads.

(b) Benign writes.

(c) Hybrid reads.

(d) Hybrid writes.

(e) Malevolent reads.

(f) Malevolent writes.

Figure 8.13. Twelve server case study with $t = 2$.

8.4.2 Usable capacity

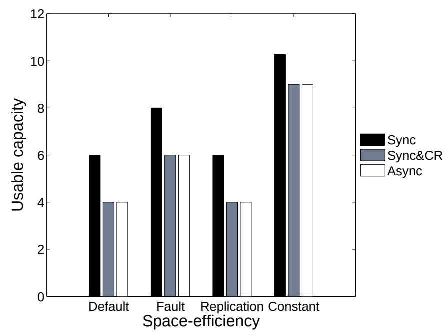
Given a server-group of size twelve, different values of m and n for R/W-PF members yield different amounts of usable capacity. Where usable capacity is the equivalent number of servers worth of raw capacity. For example, a 1 of 2 replication scheme for a server-group with twelve servers yields a usable capacity of six servers. Considering usable capacity in conjunction with throughput results allows for more concrete discussion of trade-offs in terms of cost (i.e., number of servers).

The usable capacities of the R/W-PF members with $t = 1$ are shown in Figure 8.14, and those with $t = 2$ are shown in Figure 8.15. Usable capacity is not a function of operation type and so fewer graphs are listed for usable capacity than for throughput.

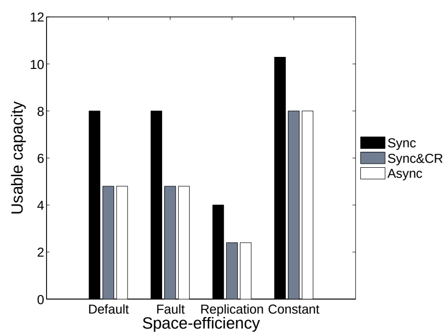
There are a few interesting trends in usable capacity. Synchronous members provide more usable capacity than Synchronous&CR or Asynchronous members given some fault and space-efficiency policy. This result is expected because such members require a smaller universe. Synchronous&CR and Asynchronous members provide the same usable capacity because m and n are identical for such members. The usable capacity for space-efficiency policies, given failure and timing policies, is most with Constant, then Fault, then Default, and then Replication. These results make sense given that usable capacity is correlated with m. The usable capacity is less for $t = 2$ than for $t = 1$ because universe size increases with t.

Consider the R/W-PF members whose throughput is discussed above. As shown in Figure 8.15c, the Asynchronous, Replication, Malevolent, $t = 2$ member provides the usable capacity of 1.33 servers. As shown in Figure 8.14a, the Synchronous, Constant, Benign, $t = 1$ member provides the usable capacity of 6 servers. This is a 4.5× difference in usable capacity.

Consider the least and most usable capacity provided by different R/W-PF members. A few members provide the usable capacity of 1.33 servers (like the Asynchronous, Replication, Malevolent, $t = 2$ member identified above). Of all of the member considered, this is the least amount of usable capacity provided. Synchronous, Constant, $t = 1$ members provide the us-



(a) Benign.



(b) Hybrid and Malevolent.

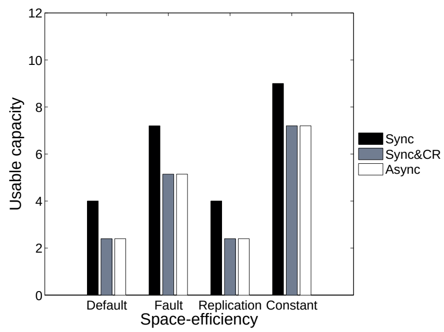
Figure 8.14. Usable capacity for twelve server case study with $t = 1$.

able capacity of 10.29 servers. This is shown in Figures 8.14a and 8.14b. Of all of the members considered, this is the most usable capacity provided.

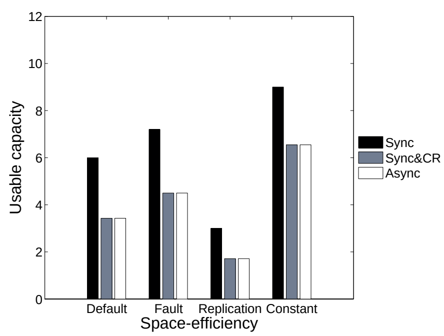
Throughput and usable capacity of the R/W-PF members evaluated varies dramatically across failure model policies, timing model policies, and space-efficiency policies.

8.5 Concurrency and failures

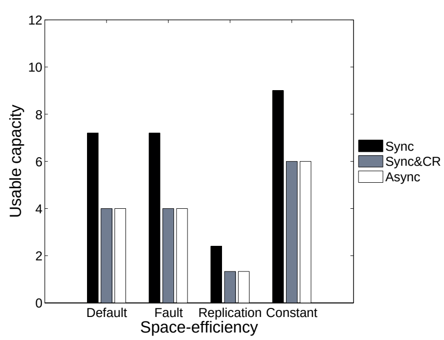
All of the results presented thus far, report measurements of best-case performance. R/W-PF members tolerate faulty servers and clients. The R/W-PF is engineered to perform efficiently in the case of concurrency- and failure-free read and write operations. The optimistic design of the R/W-PF results



(a) Benign.



(b) Hybrid.



(c) Malevolent.

Figure 8.15. Usable capacity for twelve server case study with $t = 2$.

in the performance of read operations being sensitive to concurrency and to failures.

8.5.1 Read operations

To determine the cost of concurrency and failures on read operations, we injected faults and then measured the read operation response time. Table 8.5 describes the faults injected in these experiments. We evaluate the impact these faults have on the set of Asynchronous/Synchronous, Malevolent/Malevolent$^+$ members with the Default space-efficiency. The members tolerate a single malevolent server (i.e., $t = b = 1$). The object being read is 32 KiB in size.

The results of the fault injection experiments are shown in Figure 8.16. Because only malevolent servers are tolerated by the members evaluated in Figure 8.16a, no results are shown for injections of Poisonous writes (i.e., C. Poisonous, C. Poisonous 5×, and C. & S. Malevolent).

A client performing a read operation that classifies a candidate incomplete reads back logical time (cf. C. Incomplete, C. Incomplete 5× results). This requires additional quorum RPCs to be issued. However, the prototype implementation batches additional timestamps in responses to read previous requests. The difference between one incomplete candidate and five incomplete candidates is a one additional quorum RPC, rather than the four expected, given the pseudo-code.

A client performing a read operation that classifies a candidate repairable writes the candidate back, thus making it complete (cf. C. Repairable, C. Repairable 5× results). As such, the cost of a repairable candidate is incurred only once by a client. Indeed, the read response time for repairable candidates is shown to approach that of No faults. The system is initialized with all of the objects in the working set being repairable candidates. Because the client repairs such candidates the first time it reads them, subsequent reads of that object require a single quorum RPC to complete. Server omission faults (S. Omission) require the client perform repair each time it reads the object. As such, this result indicates the cost of performing repair. Server

Fault injected	Description
No faults	Concurrency- and failure-free
C. Incomplete	Client performs a read after another client partially writes a candidate that is classified incomplete. The partial write could be due either to a client crash or read-write concurrency.
C. Incomplete 5×	Like C. Incomplete, but there are five incomplete candidates, one after the other in logical time. This could be due to a *stuttering* client, or substantial read-write concurrency.
C. Repairable	Like C. Incomplete, but the partially written candidate is classified repairable.
C. Repairable 5×	Like C. Incomplete 5×, but the partially written candidate is classified repairable.
C. Poisonous	Client performs a read after another, malevolent client completes a poisonous write.
C. Poisonous 5×	Like C. Poisonous, but the malevolent writer completes five poisonous writes, one after the other in logical time.
S. Omission	There is a single faulty server that always returns the initial value $\langle \mathbf{0}, \perp \rangle$.
S. Integrity	There is a single malevolent server that always returns a corrupt fragment.
C. & S. Malevolent	Like C. Poisonous, but there is also a single malevolent server that returns a corrupt fragment.

Table 8.5. Descriptions of faults injected.

integrity faults (S. Integrity) require the client to issue an additional read request to server outside of the preferred quorum.

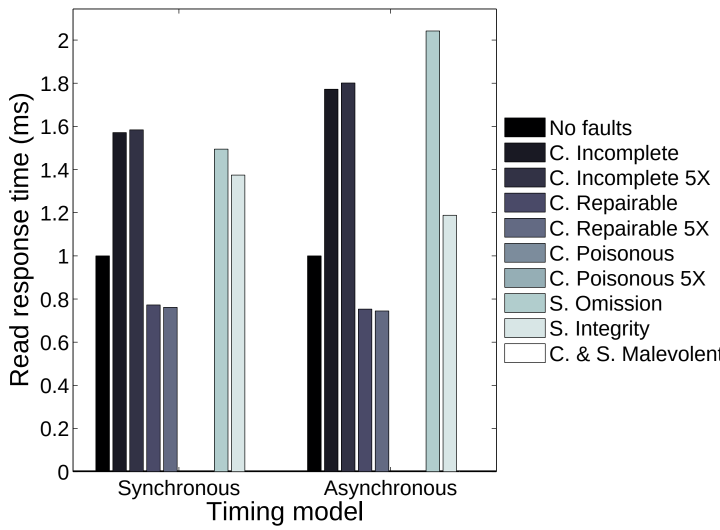
The cost of tolerating poisonous writes from malevolent clients are shown in Figure 8.16b. Poisonous writes, in some sense, are treated like incomplete candidates. However, because each poisonous write corresponds to a candidate that the client classifies complete, each poisonous candidate must be read, validated, and re-classified incomplete. This explains why the impact of the C. Poisonous 5× fault injection is so great. The client reads, decodes, and performs validation on five candidates. The reason this impact only occurs for the asynchronous timing model is that replication is used in the synchronous timing model and replicated data is self-verifying.

Lazy verification and its concomitant techniques can provide performance protection from poisonous writes [Abd-El-Malek et al., 2005b]. Specifically, servers can limit the number of unverified candidates they host in their local history on a per-object or per-client basis. Verification essentially involves the server incurring the cost of a read operation (hopefully during an otherwise idle period). Via cooperation, each server need only verify a subset of the requests they accept.

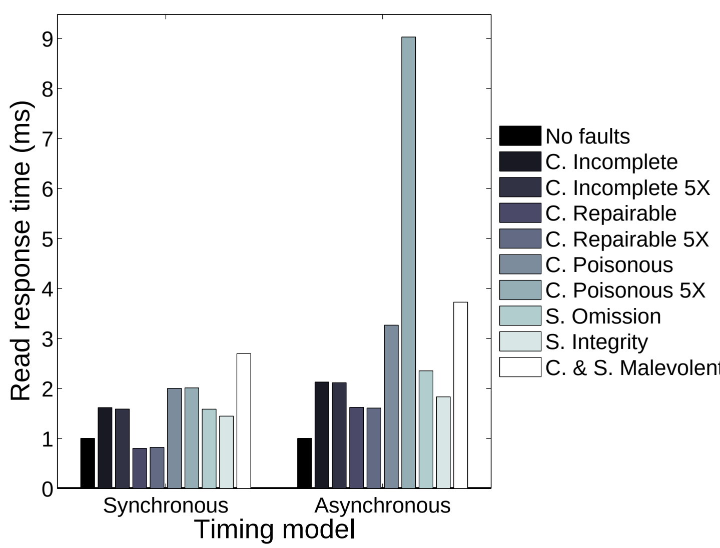
8.5.2 Write operations

The performance of a write operation is not sensitive to concurrency like a read operation. Nor is it sensitive to the same faults as a read operation. Write operations complete in one (Synchronous and Synchronous&CR members) or two (Asynchronous members) quorum RPCs. The difference being whether a local clock or a quorum RPC is used to construct the timestamp of the write operation.

Quorum RPCs must probe for a live quorum if a server in the preferred quorum crashes. Moreover, if a server crashes, the load on other servers increases. We performed an experiment to determine the performance cost of server crashes on write operations. In the experiment, the Asynchronous, Benign, Default R/W-PF member with $t = b = 5$ was evaluated. Thirty-five clients, each with ten operations outstanding, perform write operations.



(a) Malevolent failure model.



(b) Malevolent^{+} failure model.

Figure 8.16. Read response time in the face of faults.

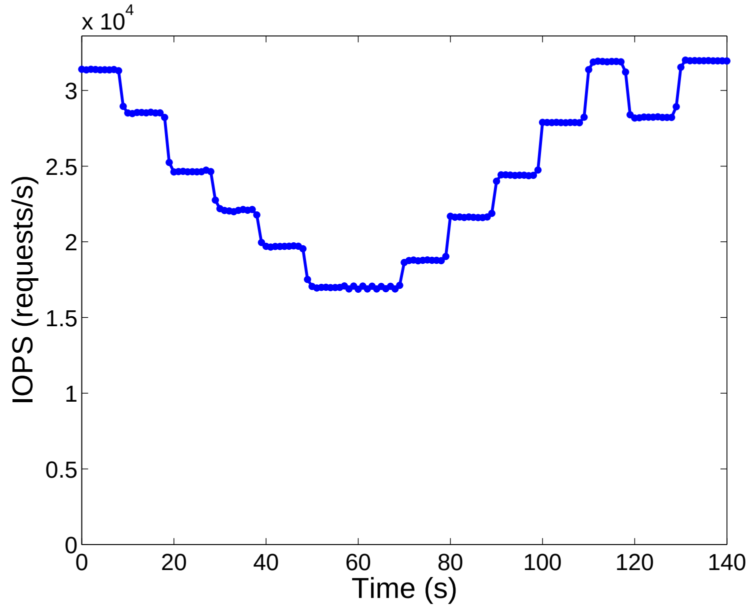


Figure 8.17. Operations per second vs. time as servers crash and recover.

The experiment ran for 150 seconds. We report the total number of write operations repeated per second, for each second of the experiment. For the first 50 seconds, we crashed a different server every 10 seconds. After 20 more seconds, we recovered a crashed server every 10 seconds. Once all the servers recovered, we crashed another server and recovered it 10 seconds later. Figure 8.17 shows the result of this experiment.

8.6 Quorums and witnesses

Quorums can provide throughput-scalability: increasing the size of the universe can increase the throughput of an R/W-PF member. We ran a set of experiments to demonstrate read throughput-scalability for some R/W-PF members. We report results for Synchronous, Benign members with the Replication and Constant space-efficiency policies. Each member tolerates a single faulty server (i.e., $t = 1$). The Δ-threshold quorum construction described in Section 5.2.1 is used in these experiments.

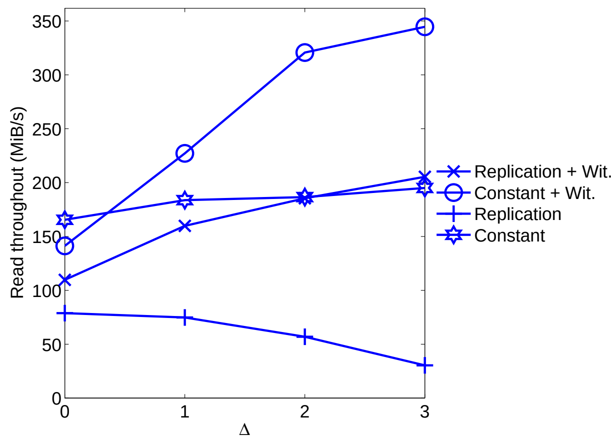


Figure 8.18. Threshold quorum throughput-scalability.

We also ran experiments with two similar members that do not employ read witnesses. These members are included in the experiment to show, empirically, the impact of not using witnesses.

Because this experiment measures throughput, we used 32 client processes, each with 5 outstanding requests in these experiments. In an effort to keep the profile of network traffic as similar as possible across R/W-PF members, we set the fragment size to be a constant 8 KiB in this experiment.

Figure 8.18 shows the results for these experiments. The read throughput of the Replication member, that uses witnesses, increases as Δ increases. This is the expected result.

Compare the Replication member with witnesses, to the same member without witnesses. Witnesses provide a compelling throughput benefit for the Replication member. However, without witnesses, no quorum throughput-scalability is observed as Δ increases. This is not the expected result. We examined counters in the server and client logs. Servers were evenly loaded and serviced an appropriate number of requests given the quorum construction. We are confident that the Δ-threshold quorum implementation is correct. From additional experiments, the results of which we do not show, we concluded that this is due to congestion of the client receive link. The number of servers that reply with entire replicas increases with Δ.

Without read witnesses the overall throughput decreases as Δ increases.

9 Conclusions

It is possible and fruitful to build a versatile storage infrastructure. The Read/Write Protocol Family (R/W-PF) enables a storage infrastructure to be built that simultaneously supports a broad range of resiliency models and storage mechanisms. The inherent data-dependent trade-offs among resiliency and performance requirements in storage systems make such versatility important. In this dissertation, the R/W-PF is described in detail. Different timing models, server failure models, and client failure models comprise the resiliency models of the R/W-PF. Quorum constructions, erasure codes, and witnesses comprise the storage mechanisms of the R/W-PF.

A prototype storage system, called PASIS, was built that employs the R/W-PF. Response time and throughput experiments with PASIS demonstrate its versatility. To give an example of the versatility provided by the PASIS storage infrastructure, in some experiments replicated objects that tolerate benign failures of clients and servers under a synchronous timing model are evaluated, whereas in other experiments erasure-coded objects that tolerate Byzantine failures of clients and servers under an asynchronous timing model are evaluated. Recent results based on the Ursa Minor storage system also support the claim that the R/W-PF enables the construction of a versatile storage infrastructure. The Ursa Minor storage system incorporates the R/W-PF. Experiments demonstrate that versatility allows an object to be stored using an R/W-PF member appropriate for its workload and reliability requirements [Ganger et al., 2005].

The quorum-based nature of the R/W-PF is the key aspect that allows a versatile storage infrastructure to be built. Quorum-based protocols localize much of the protocol logic at the client. In the case of the R/W-PF, almost all logic is located at the client. Servers have a simple, narrow interface and perform the same actions regardless of which R/W-PF member is employed.

The failure models included in the R/W-PF are all strict generalizations of one another. This facilitates using the same basic protocol steps for every protocol family member. R/W-PF members that tolerate malevolent components require more logic than those that only tolerate benign failures of components. However, that additional logic does not change the steps required in the protocol.

Members of the R/W-PF provide linearizable, wait-free read and write operations. Only read operations by correct clients and well-formed write operations are linearized. Wait-freedom hinges on unbounded storage capacity. In practice, there are bounds on storage capacity. Garbage collection of unnecessary versions is included in the prototype implementation to reclaim storage capacity. With bounded storage capacity and garbage collection, read operations are obstruction-free rather than wait-free.

9.1 Comments on continuing research

The R/W-PF demonstrates that the protocol family approach is effective for read/write objects. This dissertation does not demonstrate that the protocol family approach generalizes. However, we have some experience with another optimistic quorum-based protocol, the Query/Update protocol, that supports arbitrary deterministic functions rather than read/write operations [Abd-El-Malek et al., 2005a]. We believe that resiliency models, quorum constructions, and a variant of witnesses can be incorporated in such a protocol. As with the R/W-PF, the most complicated logic for such a query/update protocol family involves tolerating malevolent clients. Erasure coding is uniquely applicable to read/write objects. For objects that have stronger semantics than read/write objects, servers must be able to "parse" such objects to ensure the correct semantics (at least in protocol families that can tolerate malevolent clients).

Versatility offers already over-loaded system administrators more knobs to tune. This dissertation does not explain how to use versatility effectively. Tools and techniques must be developed to assist system administrators with tuning storage systems. The current research interest in autonomic storage systems complements versatility nicely. From a security perspective, the ability to select “cheap” R/W-PF members in lieu of “expensive” members that actually meet the resiliency requirements for an object may cause concern. However, given a versatile storage infrastructure, such mistakes can be corrected without purchasing a different storage system—although more servers may be required to accommodate more resilient objects.

It is desirable to be able to migrate an object from one R/W-PF member to another. This dissertation does not develop such a mechanism. A centralized migration service has been developed for the R/W-PF in the Ursa Minor system [Ganger et al., 2005]. Given the nature of the R/W-PF though, a decentralized approach to migration that shares the resiliency characteristics of the object being migrated is desirable. Verifiable secret redistribution, developed by Wong [2004], is decentralized and may provide insights into how to build a decentralized migration service. Note though that Wong considered secret shared objects exclusively. The cooperative aspects of lazy verification could also inform a decentralized migration service [Abd-El-Malek et al., 2005b].

In practice, self-validating timestamps with lazy verification are the most efficient means of tolerating Byzantine faulty clients in erasure-coded storage. However, there seems to be room for the development of a specialized cryptographic primitive that reduces the cost of validating writes of erasure-coded objects. Such a primitive could improve R/W-PF members that tolerate malevolent clients as well as the asynchronous verifiable information dispersal (AVID) technique of Cachin and Tessaro [2005a,b]. Specifically, the goal of such a primitive would be to make the AVID approach network-efficient as well as storage-efficient.

10 Glossary

Symbol	Definition
2^{set}	The power set of *set*.
$\mathcal{U}$	The entire set of servers.
U	The set of servers for a specific object.
n	The number of servers in the quorum system (i.e., $\lvert\mathcal{Q}\rvert$).
$\mathcal{T}$	The set of possible sets of faulty servers toleratd.
T	The set of faulty servers in a given system execution ($T \in \mathcal{T}$).
t	The number of faulty servers tolerated in a threshold quorum system.
$\mathcal{B}$	The set of possible sets of malevolent servers tolerated.
B	The set of malevolent servers in a given system execution ($B \in \mathcal{B}$).
b	Total number of malevolent servers tolerated in a threshold quorum system ($b \leq t$).
δ	The bound on delays in the synchronous timing model.
τ	The bound on the clock skew exhibited by loosely synchronized clocks.
m	The number of correct fragments required to decoded an m-of-n erasure-coded object.
$\mathcal{M}(Q)$	The decodable sets of a quorum (i.e., all of the subsets of Q that are of size m).
M	A decodable set.
$\mathcal{R}(Q)$	The repairable sets of a quorum.

Table 10.1. Symbols

Term	**Definition**
Accept/host	A server that *accepts* a write request *hosts* the corresponding candidate (i.e., the tuple timestamp and fragment pair).
Benign server	A server that is not malevolent. Depending on the server failure model, a benign server may be always up, eventually up, eventually down, or unstable.
Candidate	A timestamp, fragment pair.
Client	A process that reads and writes objects.
Component	A client or a server.
Faulty server	A server that is not good. Depending on the server failure model, a faulty server may be eventually down, unstable, or malevolent.
Fragment	A client encodes an object into fragments during a write and decodes fragments into an object during a read.
Good server	A server that is not faulty. Depending on the server failure model, a good server may be always up or eventually up.
IDA	Rabin's information dispersal algorithm [Rabin, 1989]. The m-of-n threshold erasure code used in PASIS the prototype implementation.
Malevolent server	A server that is not benign. The server may exhibit arbitrary and malicious failures, i.e., Byzantine failures.
Object	Generic term for a piece of data (e.g., a block or file).
RAID	Redundant array of independent (inexpensive) disks. The "traditional" set of erasure codes employed in storage systems: RAID 0 (striping), RAID 1 (replication), RAID 3/4/5 (parity), and RAID 6 (double erasure tolerant).
RPC	Client-server remote procedure call.
R/W-PF	The Read/Write Protocol Family [Wylie, 2005].
Server	A process that stores versions of fragments.
Stripe-fragment	The "first" m threshold erasure-coded fragments of an object.

Table 10.2. Terminology

Bibliography

Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. 2005a. Fault-scalable Byzantine fault-tolerant services. In *Symposium on Operating Systems Principles*. To appear. 18, 29, 86, 148

Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. 2005b. Lazy verification in fault-tolerant distributed storage systems. In *Symposium on Reliable Distributed Systems*. To appear. 14, 20, 36, 92, 142, 149

Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, 1–15. 12, 16, 19

Aguilera, M. K., Chen, W., and Toueg, S. 2000. Failure detection and consensus in the crash-recovery model. *Distributed Computing 13,* 2, 99–125. 23, 24, 26, 45

Alon, N., Kaplan, H., Krivelevich, M., Malkhi, D., and Stern, J. 2002. Scalable secure storage when half the system is faulty. *Information and Computation 174,* 2, 203–213. 35

ALON, N., KAPLAN, H., KRIVELEVICH, M., MALKHI, D., AND STERN, J. 2004. Addendum to scalable secure storage when half the system is faulty. *Information and computation*. 35

ALVISI, L., MALKHI, D., PIERCE, E., REITER, M. K., AND WRIGHT, R. N. 2000. Dynamic Byzantine quorum systems. In *Dependable Systems and Networks*. IEEE, 283–292. 30

AMIRI, K., GIBSON, G. A., AND GOLDING, R. 1999. Scalable concurrency control and recovery for shared storage arrays. Tech. Rep. CMU–CS–99–111, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA. 87

ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. 1996. Serverless network file systems. *ACM Transactions on Computer Systems 14,* 1, 41–79. 9, 16

BACKES, M. AND CACHIN, C. 2003. Reliable broadcast in a computational hybrid model with Byzantine faults, crashes, and recoveries. In *Dependable Systems and Networks*. IEEE, 37–46. 26

BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. 1991. Measurements of a distributed file system. In *ACM Symposium on Operating System Principles*. 198–212. 18

BAZZI, R. A. 1999. Non-blocking asynchronous Byzantine quorum systems. In *International Symposium on Distributed Computing*. Springer-Verlag, 109–122. 64

BAZZI, R. A. 2000. Synchronous Byzantine quorum systems. *Distributed Computing 13,* 1, 45–52. 64

BAZZI, R. A. 2001. Access cost for asynchronous Byzantine quorum systems. *Distributed Computing 14,* 1, 41–48. 64

BAZZI, R. A. AND DING, Y. 2004. Non-skipping timestamps for Byzantine data storage systems. In *DISC*. 405–419. 94

Bellare, M., Canetti, R., and Krawczyk, H. 1996. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO*. Springer-Verlag, 1–15. 86

Berlekamp, E. 1968. *Algebraic coding theory.* McGraw-Hill, New York. 31

Bhatti, N. T. and Schlichting, R. D. 1995. A system for constructing configurable high-level protocols. In *ACM SIGCOMM Conference*. ACM, 138–150. 16

Blakley, G. R. and Meadows, C. 1985. Security of ramp schemes. In *Advances in Cryptology - CRYPTO*. Springer-Verlag, 242–268. 31, 97

Blaum, M. 1987. A class of byte-correcting array codes. Tech. Rep. Research report RJ 5652 (57151), IBM. 33

Blaum, M., Brady, J., Bruck, J., and Menon, J. 1995. Evenodd – an efficient scheme for tolerating double-disk failures in raid architectures. *IEEE Transactions on Computers 44,* 2, 192–202. 33

Blundo, C., Cresti, A., De Santis, A., and Vaccaro, U. 1996. Fully dynamic secret sharing schemes. *Theoretical Computer Science 165,* 2, 407–440. 34

Boichat, R. and Guerraoui, R. 2000. Reliable broadcast in the crash-recovery model. In *Symposium on Reliable Distributed Systems*. IEEE, 32–41. 26

Bracha, G. and Toueg, S. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM 32,* 4, 824–840. 18

Byers, J. W., Luby, M., Mitzenmacher, M., and Rege, A. 1998. A digital fountain approach to reliable distribution of bulk data. *Computer Communication Review 28,* 4, 1–19. 33

CACHIN, C. AND PORITZ, J. A. 2002. Secure intrusion-tolerant replication on the Internet. In *International Conference on Dependable Systems and Networks*. IEEE, 167–176. 18

CACHIN, C. AND TESSARO, S. 2005a. Asynchronous verifiable infromation dispersal. In *Symposium on Reliable Distributed Systems*. IEEE. 20, 35, 149

CACHIN, C. AND TESSARO, S. 2005b. Brief announcement: Optimal resilience for erasure-coded Byzantine distributed storage. In *International Symposium on Distributed Computing*. Springer. 20, 35, 149

CACHIN, C. AND TESSARO, S. 2005c. Optimal resilience for erasure-coded Byzantine distributed storage. Tech. rep., IBM Research. 94

CASTRO, M. AND LISKOV, B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20,* 4, 398–461. 18

CHARRON-BOST, B., TOUEG, S., AND BASU, A. 2000. Revisiting safety and liveness in the context of failures. In *International Conference on Concurrency Theory*. 552–565. 64

CHEN, W.-K., HILTUNEN, M. A., AND SCHLICHTING, R. D. 2001. Constructing adaptive software in distributed systems. In *International Conference on Distributed Computing Systems*. IEEE, 635–643. 16

CHOR, B., GOLDWASSER, S., MICALI, S., AND AWERBUCH, B. 1985. Verifiable Secret Sharing in the Presence of Faults. In *IEEE Symposium on Foundations of Computer Science*. IEEE, 335–344. 35

CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. 2004. Row-diagonal parity for double disk failure correction. In *Conference on File and Storage Technologies*. USENIX Association, 1–14. 11, 33

Cristian, F., Aghili, H., Strong, R., and Dolev, D. 1995. Atomic broadcast: from simple message diffusion to Byzantine agreement. *Information and Computation 118,* 1, 158–179. 15

Cristian, F. and Fetzer, C. 1999. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems 10,* 6, 642–657. 15, 22

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. 2001. Wide-area cooperative storage with CFS. In *ACM Symposium on Operating System Principles*. 202–215. 18

Douceur, J. R. and Wattenhofer, R. P. 2001. Optimizing file availability in a secure serverless distributed file system. In *Symposium on Reliable Distributed Systems*. IEEE. 90

Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *Journal of the Association for Computing Machinery 35,* 2, 288–323. 22, 84

EMC Corp. August 2003. EMC Centera: content addressed storage system. EMC Corp. web site. http://www.emc.com/products/systems/-centera.jsp?openfolder=platform. 9, 16

EqualLogic Inc. 2003. PeerStorage Overview. http://www.equallogic.com/pages/products_technology.htm. 9, 16

Feldman, P. 1987. A practical scheme for non-interactive verifiable secret sharing. In *IEEE Symposium on Foundations of Computer Science*. IEEE, 427–437. 35

Fischer, M. J., Lynch, N. A., and Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32,* 2, 374–382. 22

Frølund, S., Merchant, A., Saito, Y., Spence, S., and Veitch, A. 2003. FAB: enterprise storage systems on a shoestring. In *Hot Topics in Operating Systems*. USENIX Association, 133–138. 9, 11

Frølund, S., Merchant, A., Saito, Y., Spence, S., and Veitch, A. 2004. A decentralized algorithm for erasure-coded virtual disks. In *International Conference on Dependable Systems and Networks*. IEEE, 125–134. 61

Gallager, R. G. 1963. *Low density parity-check codes*. MIT Press. 33

Ganger, G. R., Abd-El-Malek, M., Cranor, C., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., and Wylie, J. J. 2005. Ursa Minor: versatile cluster-based storage. Tech. Rep. CMU-PDL-05-104, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA. 15, 147, 149

Ganger, G. R., Strunk, J. D., and Klosterman, A. J. 2003. Self-* Storage: brick-based storage with automated administration. Tech. Rep. CMU–CS–03–178, Carnegie Mellon University. 9

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In *ACM Symposium on Operating System Principles*. ACM, 29–43. 9, 16

Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. 1998. A cost-effective, high-bandwidth storage architecture. In *Architectural Support for Programming Languages and Operating Systems*. 92–103. 16

Gifford, D. K. 1979. Weighted voting for replicated data. In *ACM Symposium on Operating System Principles*. ACM, 150–162. 28

Gladman, B. 2004a. *AES implementation*. http://fp.gladman.plus.com/cryptography_technology/rijndael. 97

Gladman, B. 2004b. *SHA implementation*. http://fp.gladman.plus.com/cryptography_technology/sha/. 98

Golding, R., Bosch, P., Staelin, C., Sullivan, T., and Wilkes, J. 1995. Idleness is not sloth. In *Winter USENIX Technical Conference*. USENIX Association, 201–212. 92

Gong, L. 1989. Securely replicating authentication services. In *International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 85–91. 20, 34, 56

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. 2003. A protocol family for versatile survivable storage infrastructures. Tech. Rep. CMU-PDL-03-103, CMU. 15

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. 2004a. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks*. 14, 20, 36, 37, 57

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. 2004b. The safety and liveness properties of a protocol family for versatile survivable storage infrastructures. Tech. Rep. CMU–PDL–03–105, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA. 15, 69

Gray, C. G. and Cheriton, D. R. 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating System Principles*. 202–210. 17

Gray, J. N. 1978. *Notes on data base operating systems*. Vol. 60. Springer-Verlag, Berlin, 393–481. 17

Hadzilacos, V. 1984. Issues of fault tolerance in concurrent computations. Ph.D. thesis, Harvard University. 25

Hadzilacos, V. and Toueg, S. 1994. A modular approach to the specification and implementation of fault-tolerant broadcasts. Tech. Rep. TR94-1425, Department of Computer Science, Cornell University. 15

Hafner, J. L., Deenadhayalan, V., Kanungo, T., and Rao, K. 2004. Performance metrics for erasure codes in storage systems. Tech. Rep. RJ–10321, IBM. 33

Hassin, Y. and Peleg, D. 2001. Average probe complexity in quorum systems. In *ACM Symposium on Principles of Distributed Computing*. ACM, 180–189. 29

Herlihy, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages 13,* 1, 124–149. 71, 77

Herlihy, M., Luchangco, V., and Moir, M. 2003. Obstruction-free synchronization: double-ended queues as an example. In *International Conference on Distributed Computing Systems*. IEEE, 522–529. 78

Herlihy, M. P. and Tygar, J. D. 1987. How to make replicated data secure. In *Advances in Cryptology - CRYPTO*. Springer-Verlag, 379–391. 19

Herlihy, M. P. and Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3, 463–492. 10, 71, 76

Hiltunen, M. A., Immanuel, V., and Schlichting, R. D. 1999. Supporting customized failure models for distributed software. *Distributed Systems Engineering 6,* 3, 103–111. 15

Hiltunen, M. A. and Schlichting, R. D. 1996. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering 11,* 5, 275–285. 16

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS) 6,* 1, 51–81. 18

Hurfin, M., Mostefaoui, A., and Raynal, M. 1998. Consensus in asynchronous systems where processes can crash and recover. In *Symposium on Reliable Distributed Systems*. IEEE, 280–286. 26

IBM Almaden Research Center. August, 2003. Collective Intelligent Bricks. IBM Almaden Research Center web site. http://www.almaden.ibm.com/StorageSystems/autonomic_storage/-CIB/index.shtml. 9

Jayanti, P., Chandra, T. D., and Toueg, S. 1998. Fault-tolerant wait-free shared objects. *Journal of the ACM 45,* 3, 451–500. 71, 77

Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G., and Kemme, B. 2003. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS) 28,* 3, 257–294. 29

Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. 2001. The SecureRing group communication system. *ACM Transactions on Information and Systems Security 1,* 4, 371–406. 18

Kleiman, S. October 2002. Personal communication. Network Appliance, Inc. 11

Knuth, D. E. 1997. *Seminumerical algorithms*. Vol. 2. Addison-Wesley. 102

Krawczyk, H. 1993. Distributed fingerprints and secure information dispersal. In *ACM Symposium on Principles of Distributed Computing*. 207–218. 35

Krawczyk, H. 1994. Secret sharing made short. In *Advances in Cryptology - CRYPTO*. Springer-Verlag, 136–146. 31, 97, 117

Krohn, M. N., Freedman, M. J., and Mazieres, D. 2004. On-the-fly verification of rateless erasure codes for efficient content distribution. In *IEEE Symposium on Security and Privacy*. 35

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaten, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. In *Architectural Support for Programming Languages and Operating Systems*. 190–201. 19

Kung, H. T. and Robinson, J. T. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6,* 2, 213–226. 17

Lamport, L. 1985. On interprocess communication. Tech. Rep. 8, Digital Equipment Corporation, Systems Research Center, Palo Alto, Ca. 17

Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4,* 3, 382–401. 11, 23

Lamport, L. L. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks 2*, 95–114. 17, 79

Lampson, B. W., Paul, M., and Siegert, H. J. 1981. *Distributed systems — architecture and implementation: an advanced course*. Vol. 105. Springer-Verlag, New York. 26

Lee, E. K. and Thekkath, C. A. 1996. Petal: distributed virtual disks. In *Architectural Support for Programming Languages and Operating Systems*. 84–92. 9, 16, 17, 19, 89

Liskov, B. 1991. Practical uses of synchronized clocks in distributed systems. In *ACM Symposium on Principles of Distributed Computing*. ACM, 1–9. 23, 95

Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. 1991. Replication in the Harp file system. In *ACM Symposium on Operating System Principles*. 226–238. 17

Luby, M. 2002. LT Codes. In *IEEE Symposium on Foundations of Computer Science*. IEEE, 271–280. 33

Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., and Spielman., D. A. 2001. Efficient Erasure Correcting Codes. *IEEE Transactions on Information Theory 47,* 2, 569–584. 33

Lustre May 2004. Lustre. http://www.lustre.org/. 16

Malkhi, D. and Reiter, M. 1998. Byzantine quorum systems. *Distributed Computing 11,* 4, 203–213. 18, 19, 24, 28, 60, 89

Malkhi, D., Reiter, M., and Wool, A. 2000. The load and availability of Byzantine quorum systems. *SIAM Journal of Computing 29,* 6, 1889–1906. 28, 29, 84

Malkhi, D. and Reiter, M. K. 2000. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering 12,* 2, 187–202. 20, 89

Malkhi, D., Reiter, M. K., Tulone, D., and Ziskind, E. 2001. Persistent objects in the Fleet system. In *DARPA Information Survivability Conference and Exposition.* IEEE, 126–136. 18

Malkhi, D., Reiter, M. K., and Wright, R. 1997. Probabilistic quorum systems. In *ACM Symposium on Principles of Distributed Computing.* ACM, 267–273. 30

Martin, J.-P. and Alvisi, L. 2004. A framework for dynamic Byzantine storage. In *International Conference on Dependable Systems and Networks.* IEEE, 325–334. 30

Martin, J.-P. and Alvisi, L. 2005. Fast Byzantine consensus. In *International Conference on Dependable Systems and Networks.* IEEE, 402–411. 18

Martin, J.-P., Alvisi, L., and Dahlin, M. 2002. Minimal Byzantine storage. In *International Symposium on Distributed Computing.* 19, 20, 35

Mazieres, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. 1999. Separating key management from file system security. In *ACM Symposium on Operating System Principles*. ACM, 124–139. 35

Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. 1996. *Handbook of Applied Cryptography*. CRC Press. 102

Meyer, F. J. and Pradhan, D. K. 1991. Consensus with dual failure modes. *IEEE Transactions on Parallel and Distributed Systems 2,* 2, 214–222. 24

Mills, D. L. 1992. *Network time protocol (version 3)*. IETF. 95

Mills, D. L. 1995. Improved algorithms for synchronizing computer network clocks. *IEEE-ACM Transactions on Networking 3,* 3, 245–254. 95

Mishra, S., Fetzer, C., and Cristian, F. 2002. The Timewheel Group Communication System. *IEEE Transactions on Computers 51,* 8, 883–899. 15

Mullender, S. J. 1985. A distributed file service based on optimistic concurrency control. In *ACM Symposium on Operating System Principles*. 51–62. 18

Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. 2002. Ivy: a read/write peer-to-peer file system. In *Symposium on Operating Systems Design and Implementation*. USENIX Association. 19

Naor, M. and Wool, A. 1998. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing 27,* 2, 423–447. 28, 29, 84

NIST. 1995. *FIPS 180-1: Secure Hash Standard*. National Institute of Standards and Technology. http://www.itl.nist.gov/fipspubs/fip180-1.htm. 97

Noble, B. D. and Satyanarayanan, M. 1994. An empirical study of a highly available file system. Tech. Rep. CMU–CS–94–120, Carnegie Mellon University. 18

Panasas, Inc. July 2005. Panasas ActiveScale Storage Cluster. http://www.panasas.com/products_overview.html. 16

Pâris, J.-F. 1986. Voting with witnesses: a consistency scheme for replicated files. In *International Conference on Distributed Computing Systems*. IEEE Computer Society, 606–612. 36

Pâris, J.-F. 1989. Voting with bystanders. In *International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 394–401. 37

Patterson, D. A., Gibson, G., and Katz, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data*. 109–116. 9, 31

Pedersen, T. P. 1991. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology - CRYPTO*. Springer-Verlag, 129–140. 35

Peleg, D. and Wool, A. 1996. How to be an efficient snoop, or the probe complexity of quorum systems. In *ACM Symposium on Principles of Distributed Computing*. ACM, 290–299. 29

Perry, K. J. and Toueg, S. 1986. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering SE–12,* 3, 477–482. 23, 25

Pierce, E. T. 2001. Self-adjusting quorum systems for Byzantine fault tolerance. Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin. 19

Plank, J. S. and Thomason, M. G. 2004. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *International Conference on Dependable Systems and Networks*. 115–124. 33

Rabin, M. O. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36,* 2, 335–348. 31, 99, 152

Reed, D. P. 1983. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems 1,* 1, 3–23. 18

Reiter, M. K. 1995. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), 99–110. 18, 20

Renesse, R. V., Birman, K., Hayden, M., Vaysburd, A., and Karr, D. 1998. Building adaptive systems using ensemble. *Software—Practice & Experience 28,* 9, 963–979. 16

Rivest, R. L. 1992. *The MD5 message-digest algorithm.* Network Working Group, IETF. 97, 98

Rosenblum, M. and Ousterhout, J. K. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10,* 1, 26–52. 82

Rowstron, A. and Druschel, P. 2001. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *ACM Symposium on Operating System Principles.* ACM, 188–201. 18

Saito, Y., Frolund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. In *Architectural Support for Programming Languages and Operating Systems.* ACM, 48–58. 16, 17

Schlichting, R. D. and Schneider, F. B. 1983. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems 1,* 3, 222–238. 23

Schneider, F. B. 1984. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems 2,* 2, 145–154. 24, 65

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22,* 4, 299–319. 17, 79

SHAMIR, A. 1979. How to share a secret. *Communications of the ACM 22,* 11, 612–613. 31, 97, 99, 117

SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. 2003. Metadata efficiency in versioning file systems. In *Conference on File and Storage Technologies*. USENIX Association, 43–58. 82

STEINER, J. G., SCHILLER, J. I., AND NEUMAN, C. 1988. Kerberos: an authentication service for open network systems. In *Winter USENIX Technical Conference*. 191–202. 86

STINSON, D. R. 1995. *Cryptography: Theory and practice*. CRC Press. 102, 103

STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A. N., AND GANGER, G. R. 2000. Self-securing storage: protecting data in compromised systems. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, 165–180. 82

THAMBIDURAI, P. AND PARK, Y. 1988. Interactive consistency with multiple failure modes. In *Symposium on Reliable Distributed Systems*. IEEE, 93–100. 24

THEKKATH, C. A., MANN, T., AND LEE, E. K. 1997. Frangipani: a scalable distributed file system. In *ACM Symposium on Operating System Principles*. ACM, 224–237. 19, 89

THOMAS, R. H. 1979. A majority consensus approach to concurrency control. *ACM Transactions on Database Systems 4*, 180–209. 28

TOMPA, M. AND WOLL, H. 1988. How to share a secret with cheaters. *Journal of Cryptography 1,* 2, 133–138. 34

van Renesse, R. and Schneider, F. B. 2004. Chain replication for supporting high throughput and availability. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, 91–104. 90

van Renesse, R. and Tanenbaum, A. S. 1988. Voting with ghosts. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Washington, DC, 456–462. 37

Wang, X., Feng, D., Lai, X., and Yu, H. 2004. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Tech. Rep. 2004/199, Cryptology ePrint Archive. August. http://eprint.iacr.org/. 97

Wei Dai. 2005. *Crypto++ reference manual*. http://cryptopp.sourceforge.net/docs/ref/. 97, 99

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. 1996. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems 14,* 1, 108–136. 17

Wong, T. M.-T. 2004. Decentralized recovery for survivable storage systems. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. 149

Wool, A. 1998. Quorum systems in replicated databases: science or fiction. *Bull. IEEE Technical Committee on Data Engineering 21,* 4, 3–11. 29, 84

Wylie, J. J. 2005. A read/write protocol family for versatile storage infrastructures. Tech. Rep. CMU–PDL–05–108, Carnegie Mellon University. 152

Wylie, J. J., Bakkaloglu, M., Pandurangan, V., Bigrigg, M. W., Oguz, S., Tew, K., Williams, C., Ganger, G. R., and Khosla, P. K. 2001. Selecting the right data distribution scheme for a survivable storage system. Tech. Rep. CMU–CS–01–120, Carnegie Mellon University. 14

Wylie, J. J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Killiccote, H., and Khosla, P. K. 2000. Survivable information storage systems. *IEEE Computer 33,* 8, 61–68. 14, 81

Wylie, J. J., Goodson, G. R., Ganger, G. R., and Reiter, M. K. 2004. A protocol family approach to survivable storage infrastructures. In *FuDiCo II: S.O.S. (Survivability: Obstacles and Solutions), 2nd Bertinoro Workshop on Future Directions in Distributed Computing.* 15

Yu, H. 2004. Signed quorum systems. In *ACM Symposium on Principles of Distributed Computing.* ACM, 246–255. 30

Zhang, Z., Lin, S., Lian, Q., and Jin, C. 2004. RepStore: a self-managing and self-tuning storage backend with smart bricks. In *International Conference on Autonomic Computing.* IEEE, 122–129. 17