Secure Store: A Secure Distributed Storage Service

A Thesis
Presented to
The Academic Faculty

by

Subramanian Lakshmanan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
August 2004

Secure Store: A Secure Distributed Storage Service

Approved by:

Dr. Mustaque Ahamad, Advisor

Dr. H. Venkateswaran

Dr. Douglas M. Blough

Dr. Wenke Lee

Dr. UmaKishore Ramachandran

Date Approved: 12 August 2004

ACKNOWLEDGEMENTS

This thesis would not have been possible without the encouragement and support of a number of people. I would like to dedicate this thesis to all of them and express my gratitude.

First and foremost, I would like to thank my advisor Dr. Mustaque Ahamad and Co-advisor Dr. H. Venkateswaran. Prof. Mustaque gave me ample freedom in pursuing the topics of my interest while always guiding me with his expert comments and invaluable insights that influenced my research to a great extent. He has also been very accommodative of my working habits and I have personally learnt a lot from him at work. Prof. Venkat taught me that theory can find a place in practice and enhanced my interests in mathematical aspects of computing. I have also been fortunate to enjoy his friendship and encouragement that extends far beyond work.

I would like to thank the members of my thesis committee for their immense support and guidance. Dr. Douglas Blough has always been receptive to new ideas with his critical thoughts right from my early days at Georgia Tech. Dr. Wenke Lee has been a guiding expert in all questions related to security and helped me appreciate the practical aspects of security in the real world. Dr. Umakishore Ramachandran's comments have imparted a system's perspective both in this dissertation and in my general approach to research.

All my teachers have played a very significant role in shaping my academic and personal life. I would like to thank Mrs. Sripadma Narasimhan, Mrs. Ananthalakshmi Gopalakrishnan, Mrs. Rajalskhmi, Mrs. Janaki Subbiah, Mrs. Roseline Prabhakar, Mr. Sangapillai and a number of others at my primary, secondary and higher secondary school. I'm greatly indebted to my tutors at Anugraham classes. Without their dedicated and sincere help, I would not have developed an ardent interest in science and engineering, nor would I have studied in I.I.T. that opened the doors to innumerable oppurtunities.

I would like to thank all my friends, in particular, Ajay Gummalla and Sashidhar Merugu

who have been of immense help right from the day I landed in Atlanta. They will always be the first ones I will look up to for any help and guidance.

Finally, my deepest thanks to my parents, sister, two brothers, and their families for being so encouraging and supportive of me in all my endeavors. My nephews and niece: Raja, Akila, Lechu, Vignesh and Aadhithya have made me realize how true the following couplet by the celebrated tamil poet Thiruvalluvar is : "kuzhal inithu yaazh inithu enbar thammakkall mazhalai chol kelathavar" (an attempted translation: "flute and veena would sound sweet to people who have not heard the sweet words of their children").

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

As computers become pervasive in environments that include the home and community, new applications are emerging that will create and manipulate sensitive and private information. These applications span systems ranging from personal to mobile and hand held devices. They would benefit from a data storage service that protects the integrity and confidentiality of the stored data and is highly available. Such a data repository would have to meet the needs of a variety of applications that handle data with varying security and performance requirements.

Providing simultaneously both high levels of security and high levels of performance may not be possible when many nodes in the system are under attack. The agility approach to building secure distributed services advocates the principle that the overhead of providing strong security guarantees should be incurred only by those applications that require such high levels of security and only at times when it is necessary to defend against high threat levels. A storage service that is designed for a variety of applications must follow the principles of agility, offering applications a range of options to choose from for their security and performance requirements.

This research presents *secure store*, a secure and highly available distributed store to meet the performance and security needs of a variety of applications. *Secure store* is designed to guarantee integrity, confidentiality and availability of stored data even in the face of limited number of servers being compromised. *Secure store* is designed based on the principles of agility. *Secure store* integrates two well known techniques, namely replication and secret-sharing and exploits the natural tradeoffs that exist between security and performance to offer applications a range of options to choose from to suit their needs.

This thesis makes several contributions, including (1) illustration of the the principles of agility in building a secure distributed store, (2) a novel gossip-style secure dissemination

protocol whose performance is comparable to the best-possible benign-case protocol in the absence of any malicious activity, (3) demonstration of the usefulness and the performance benefits of using weaker consistency models for data access in our target environment, (4) a file system implementation demonstrating the feasibility and the practicality of our approach, and (5) a novel key allocation scheme and a technique called collective endorsement that can be used in other secure distributed applications. The research demonstrates the feasibility of the agile security approach for storing critical and sensitive data.

CHAPTER 1

INTRODUCTION AND MOTIVATION

Recent developments in the computing field are marked by two clear trends: (1) the emergence of pervasive computing technologies, and (2) an increasing number of security breaches in networked systems that are used by a large number of users for a variety of activities. As pervasive computing related technologies become cheaper and more feasible, ubiquitous applications and supporting infrastructure are being deployed on a wider scale. Particularly, a variety of computing devices, ranging from sensors to hand-helds and servers are increasingly being deployed in environments that include the home and community.

As computational devices enable new applications that touch the personal lives of users, these devices will create and manipulate sensitive and private information about the users that must be protected against leakage and tampering, both accidental and targeted. Clearly, the pervasive computing infrastructure and the data created and handled by it needs to be secured if it is to be deployed and used on a wider scale.

The resource constraints in pervasive computing environments may not allow the devices to store data locally. Such devices would also be an obvious target for an adversary and could be compromised with relative ease. Hence, an essential part of the supporting infrastructure would be a reliable data storage service. It is very important to secure the data storage service against unauthorized access and make it highly available for the applications that rely on the service.

An increasing number of security breaches are being reported in systems ranging from personal desktops to critical enterprise systems. Worms and viruses frequently exploit vulnerabilities in operating systems and application software. Very often it is a known and published vulnerability that is exploited, taking advantage of the fact that many users do not apply latest security patches in a timely manner. In spite of the sophisticated software engineering practices and more than thirty years of research in computer security, building

a robust and totally secure system has been an elusive art.

While developments in the field of cryptography have made it possible to communicate securely over untrusted communication channels, current techniques to ensure that only authorized clients access critical information and services are not adequate. Hence it is necessary to develop new techniques to secure the pervasive computing infrastructure and the data it handles.

1.1 Motivation

We motivate our research by considering an example application environment. **Aware Home** [4], that has been built at Georgia Tech, is a home of the future that assists elderly residents in their day-to-day activities so they can live in their homes longer. It is an information rich environment where a number of devices like sensors and cameras capture information about the residents and their activities. Such information could be used in a number of ways, including learning their living patterns to assist them better, archiving medical readings for later use by a doctor or identifying and connecting an ailing resident with an external medical facility in case of an emergency.

Clearly, information created and manipulated by the devices in **Aware Home** is sensitive and private to the residents. Such sensitive information is stored and handled by applications that span devices ranging from personal to mobile and handheld computers. Clearly, the information that is used to make decisions at times of emergency must be highly available. Also, because of the privacy concerns and sensitive nature of the information, access to it must be secured. In particular, integrity and confidentiality of the stored data should not be compromised.

Several characteristics of computers that execute such applications make them unsuitable for storing sensitive information. First, the devices may be resource poor and may not be able to store long-term data. Second, they can be easily stolen or compromised and hence cannot be trusted with long term storage of data that has confidentiality and integrity requirements. Third, when data size becomes large, storage management is expensive and prone to errors. As ubiquitous applications proliferate, the need for services that can store

data securely will arise in environments that span the home and community where fewer assumptions can be made about proper system management.

1.2 Research Challenges

Designing a secure data repository service for applications like the one described above poses several interesting challenges. First, the storage service has to be distributed and replicated across multiple nodes to allow high availability, fast ubiquitous access and to avoid single point of vulnerability. When the service is distributed among multiple servers, some of the servers can be compromised by an adversary. Hence, the service must be designed to meet the security and performance requirements of clients even in the face of a limited number of compromised servers (e.g, Byzantine fault tolerance).

Access to stored data must be secured. In particular, unauthorized access to data must not be allowed. Traditionally, security of stored data has been addressed by encrypting the data before storing it at untrusted servers.The decryption key is made available only to the owner of the data and other authorized users. However, keys have finite lifetime and, re-keying stored data and key management becomes expensive particularly when data is replicated at a number of servers, some of which can be compromised. Hence, other techniques have to be used to address confidentiality requirements of long-lived sensitive data. When data is replicated and shared among multiple clients, data consistency is an issue to be addressed. Finally, providing simultaneously both high levels of security and high levels of performance may not be possible. Hence a storage service that is designed for a variety of applications must offer applications a range of options to choose from to meet their security and performance requirements.

This research illustrates the principles of agility in building secure store systems. Systems typically cater to the needs of a variety of applications that differ in their requirements in security, performance and others. It is a well understood fact that there is always an inherent tradeoff between security and performance and that security comes at a price. A high level of security may be needed only at times when the system is under attack and the threat level is high. Besides, a common solution that guarantees a high level of security

would unnecessarily penalize applications that do not need such strong guarantees. Agile systems address this problem by providing service at varying security levels and offering explicitly to the clients the tradeoff between security and performance. The agility approach advocates the principle that the overhead of providing strong security guarantees should be incurred only by those applications that require such high levels of security and only at times when a high threat level is observed.

This research presents *secure store*, a secure and highly available distributed store to meet the performance and security needs of a variety of applications. *Secure store* is built on the principles of agility. The design of *secure store* integrates two well known techniques, namely replication and secret-sharing to achieve this goal. *Secure store* provides desirable levels of security guarantees and performance by exploiting the natural tradeoffs possible between the two conflicting goals. The store offers integrity, confidentiality and availability of data in the face of limited number of compromises. The maximum number of compromised servers that is to be tolerated is a chosen parameter that determines the levels of security guarantees offered by the system. Several issues concerned with the storage service like data dissemination in the presence of malicious servers and guaranteeing consistency of data access are also addressed. A prototype of the system has been built and empirically evaluated to demonstrate the practical feasibility of the store.

In the remainder of this chapter, we discuss principles of agility, describe the agile store project and define our research goals in the context of agile store. We motivate our research by exploring some of the challenges in addressing our goals. The chapter is concluded with a summary of the contributions of this dissertation and an outline of the following chapters.

1.3 The Agility Approach and The Agile Store Project

The concept of agile and adaptive systems have been explored by others in the past in systems that do not address security as a major concern [56, 63]. We characterize here agility in systems that address security as a central concern, in particular, those that tolerate malicious node compromises.

1.3.1 Characterizing Agile Systems

Agile systems strive to meet the requirements of applications that vary widely in their security and performance needs by providing service at varying levels of security and offering the clients an explicit tradeoff between security and performance. The desirable levels of security and performance are chosen by clients as parameters while accessing the service. For example, in the case of a secure storage service that tolerates a limited number of server compromises, the number of compromises to be tolerated is a parameter that the client chooses for storing a data item at a particular level.

In addition to allowing the clients to choose the parameters appropriately, agile systems would also adapt to prevailing conditions continuously, striving to offer the best service possible under current conditions. An agile storage service, apart from storing data at different security levels, would also allow the relevant parameters to be changed dynamically even after the data is stored. This would be useful when, for example, a high amount malicious activity is detected in the system and clients want to upgrade the security level of the stored data items.

To provide clients with feedback about the malicious activity and the threat level, typically agile systems would also have components to detect intrusions at servers and clients and watch for faulty behavior.

1.3.2 The Agile Store Project

The Agile Store project at Georgia Tech is aimed at exploring a flexible, agile and practical architecture for a distributed store. The research project is aimed at designing, implementing and evaluating an integrated and agile architecture that will allow us to characterize the tradeoffs that are inherent in the operation of a secure storage service. A primary requirement is that the overheads incurred by protocols that are developed to overcome compromises must depend on the level of attacks or malicious activity. Such agility for the protocols is driven by novel fault diagnosis and intrusion detection techniques that are being explored in the context of a distributed storage service. The project seeks to develop an architecture that provides the highest level of performance when no or few attacks are

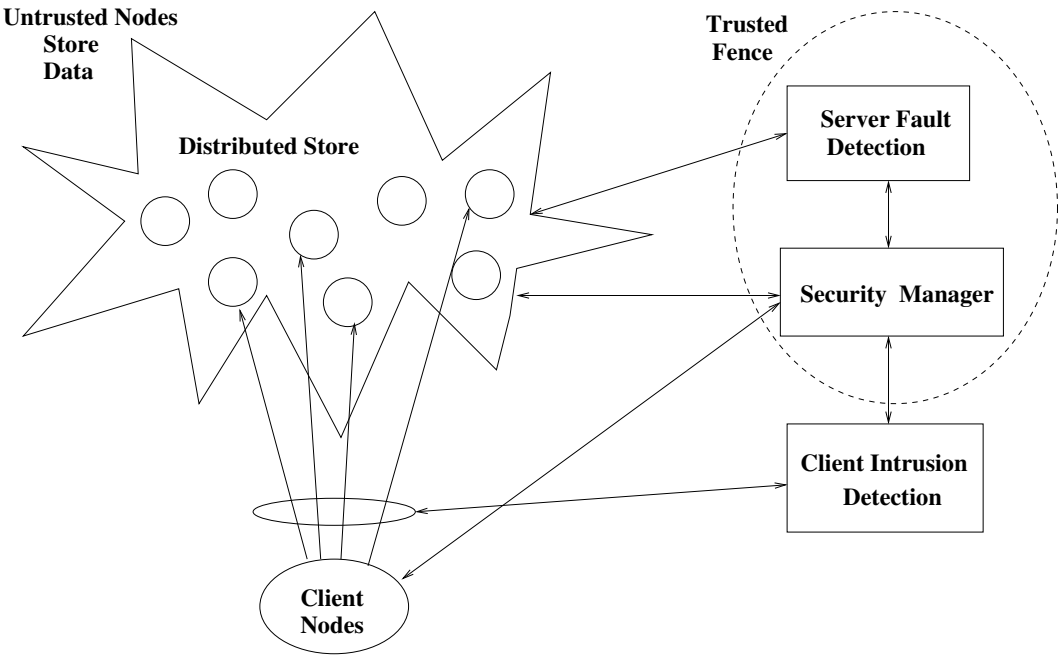


Figure 1: Agile Store Architecture

detected or suspected. As the number of compromised nodes increases or malicious attacks from clients are detected, agile store protocols would continue to maintain the security levels for sensitive data with potentially degraded levels of performance. This dissertation is part of the agile store project that addresses a subclass of problems in its context.

Figure 1 presents an overview of the architecture of *agile store*. A set of servers designated as data servers offer the data storage service. Clients access the service using a specific set of data access protocols of their choice, based on the requirements of the application. A fault diagnosis service monitors the server responses to determine if it has been compromised. An intrusion detection service monitors the activities of clients to identify misbehaving clients. Both intrusion detection and fault diagnosis services receive data from data servers but run on trusted nodes.

A security manager takes input from the intrusion detection and fault detection components and based on this input can do one or more of the following:

- Give feedback to the clients about the current threat level. Clients would in turn, based on this feedback, can upgrade or downgrade the security level of their data

items stored in the store or choose parameters appropriately for new data items.

- Choose the parameters for the clients appropriately. Upon noticing increased threat level, the manager can instruct the data servers to upgrade the security level and vice versa.

- Upon detecting intrusion at the client node, the system can remove a particular client from access control lists and block the client's access to data items until measures are taken to correct the client.

Central to agile store is a data storage architecture that supports protocols and mechanisms to exploit the security-performance tradeoff. This research focuses on a data storage architecture and the associated set of protocols to store data at a distributed set of servers. *Secure store* is that part of agile store that addresses the data storage and retrieval for a wide class of applications where very strong data consistency is not required.

1.4 Dissertation Overview

Our research focuses on the design, implementation and evaluation of *Secure store*, a data storage service that is part of agile store that addresses the needs of a wide class of applications. In this section, we provide an overview of the goals of our research and the contributions made in this dissertation.

1.4.1 Goals and Problem Definition

Our primary goal is to build a Byzantine fault tolerant data repository service that meets the security and performance requirements of a variety of applications. Our research will illustrate the principles of agility by identifying tradeoffs between security guarantees and other metrics that characterize the service, and devising methods to explicitly offer clients the benefits of such tradeoffs.

We address the problem of building a secure storage service that is distributed among a number of servers some of which can be compromised. The store should guarantee integrity and confidentiality of stored data and should be highly available. The store should meet

the security and performance requirements of a variety of applications, even when some of the servers that implement the service are compromised by an adversary. Hence, no single server should be trusted. We assume that at any time a bounded number of servers can be compromised by an adversary and can act maliciously. In general, the higher the number of compromised servers to be tolerated, the lower is the system performance. Hence, the number of compromised servers that should be tolerated is left as a parameter for the clients to choose according to their security and performance requirements. The storage service should adhere to the principles of agility, overheads being incurred for security only at times of high levels of threat or malicious activity and only those applications that require such high levels of security guarantees should experience the overheads.

1.4.2 Target Applications

Secure store addresses the needs of one class of applications: applications that handle personal and confidential documents. Data sharing in these applications is limited. Strong consistency (e.g, atomicity) for stored data is not a requirement since these applications handle mostly personal data with limited sharing. However, there is a strong need for long-term confidentiality and integrity of stored data. In particular, clients may not access the data for an extended period of time and would expect the store to maintain integrity and confidentiality of data while maintaining high availability at any time.

Examples of such applications include desktop applications that handle personal documents like tax and financial documents, corporate secrets or sensor recorded data in environments like the aware home that capture information about the residents. Most data that require data confidentiality are those that are either strictly personal or shared only to a limited extent.

1.4.3 Dissertation Research Tasks

Towards meeting our goal of building an agile secure storage service, the following have been accomplished.

- The basic data storage architecture has been designed. This includes a novel data distribution scheme to store data over a set of servers, and the associated set of protocols to retrieve and access the data

- The secure data dissemination problem in the context of secure store has been addressed. Secure store achieves replication by a background dissemination protocol that has been designed to be secure and fast.

- The problem of providing various levels of data consistency in the presence of Byzantine faults has been addressed. This ensures that data accessed is not outdated and meets the requirements of the applications.

- A prototype file system has been implemented based on the secure store architecture, as a proof of concept. The implementation has been empirically evaluated to understand the costs and tradeoffs involved in such a design and to demonstrate the agility approach.

1.4.4 Thesis Contributions

We highlight here the main contributions of this dissertation that illustrate the principles of agility.

Security-Performance Tradeoff

It is a well understood and accepted fact that there is an inherent tradeoff between security and performance. Any system that strives to offer a secure service pays a performance penalty compared to its insecure counterpart. This effect is more pronounced in distributed systems that tolerate malicious compromises. Performance penalty directly depends on the number of compromises to be tolerated. The data storage scheme we have developed is a combination of two well known techniques, replication and secret-sharing. Our data storage scheme, along with the data access and associated protocols, offers clients an explicit tradeoff between data confidentiality on one hand and availability and access costs on the other. To quantify the tradeoffs, we have developed probabilistic definitions for the security properties to reflect the intuition that the more likely an adversary can steal or modify

data, the less secure the data is. We have developed an architecture where the same set of servers can store various data items at varying security levels, thus giving *secure store* a feature of agility, which is, letting clients choose their security and performance levels.

Adapting to Malicious Activity

Data stored in *secure store* is replicated through a background dissemination process to increase data availability and improve spatial data locality. One component of data dissemination is secure dissemination of verification strings associated with each data item. One primary requirement for the dissemination component of *secure store* is to avoid public key operations for performance considerations. The existing protocols suffered from high latency for secure dissemination, even when there were no malicious servers. This thesis presents a dissemination protocol where the dissemination latency is comparable to the corresponding protocol meant for benign-case, when there is no server nodes are compromised. The price paid for security, the additional delay in dissemination, is directly proportional to the amount of malicious activity in the system. Thus, a client or an administrator can set a very high safety parameter, a high value for the number of malicious servers to be tolerated, and still not incur any overhead as long as there is no malicious activity. This is the first protocol to our knowledge where security overheads depend on the actual degree of malicious activity and illustrates an interesting aspect of agility: continuously adapting to prevailing levels of malicious activity.

We have developed a novel key allocation scheme and a technique called collective endorsement for the purpose of secure dissemination. The key allocation scheme and the collective endorsement approach themselves are contributions of this thesis, that can be used in other applications in secure distributed systems.

Consistency-Performance Tradeoff

Our target applications are those where data are either not shared or shared to a limited extent. Ordering requirements on data accesses by one or more clients defines data consistency. In traditional benign distributed stores and distributed shared memories, researchers have methodically weakened consistency guarantees to weaker, yet useful levels and devised simpler consistency protocols that offer better performance. This tradeoff between data

consistency and performance exists in systems that tolerate malicious faults too. For example, guaranteeing single copy semantics requires a three round protocol between all servers, with $O(n^2)$ communication every round, while weaker consistency levels, like causal semantics, result in more scalable protocols. Since most data that require high confidentiality are also those that are not shared extensively, weaker levels of consistency like causality would be sufficient for our target applications. We have developed secure protocols that guarantee such weaker levels of consistency guarantee in the presence of a limited number of malicious servers. Our protocols offer a significant performance benefit when compared to those that guarantee stronger levels of consistency.

1.5 Thesis Organization

The remainder of this thesis is organized as follows.

In chapter 2, a detailed literature survey of work related to our research is presented. This chapter gives a brief introduction to secret-sharing algorithms which are used extensively in our data storage scheme. This chapter also discusses some of the shortcomings of traditional encryption-based and secret-sharing based security for distributed storage. We also discuss some of the ongoing projects in secure distributed storage.

In chapter 3, we discuss the store's architecture, giving an overview of the various components that make up the store. We describe in detail the data storage scheme and the associated read/write protocol we developed, and do a preliminary analysis showing the tradeoffs our architecture can offer between security and performance.

In chapter 4, we discuss an important component of the secure store architecture, secure data dissemination. The novel key allocation scheme and the collective endorsement technique are presented and the dissemination protocol itself is developed using the key allocation and the collective endorsement technique.

In chapter 5, we present techniques to guarantee weaker forms of consistency in *secure store*. First, protocols are developed to guarantee weaker consistency in purely replicated systems and subsequently *secure store* data access protocols described in chapter 3 are modified to include consistency guarantees.

In chapter 6, we describe the file system that has been implemented based on *secure store* architecture. We present experimental results demonstrating the practicality of the ideas presented in this thesis and illustrate the agility properties of *secure store*.

Finally, in chapter 7, we summarize our research and conclude with suggestions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This thesis builds upon a large collection of existing work. In this chapter, we review the literature that is related to our research. We introduce in this chapter secret-sharing algorithms that we use in our data access protocol that is described in the subsequent chapter. We point out the shortcomings of traditional encryption-based and secret-sharing based security in distributed storage and present a case for a hybrid scheme that we develop in the next chapter. We also discuss work related to consistency in secure distributed storage and secure dissemination. In addition, we discuss some of the ongoing and recent projects related to ours.

2.1 Threat Models

Security threats to computers have existed even when most of the computers were standalone systems and were not extensively networked. System designers addressed security issues in multi-user environments. These were essentially access control mechanisms such as memory protection supported by the hardware and the operating system along with safety measures like fault isolation and safe software engineering practices. Most of the attacks were due to attackers having physical access to the machine and hence physical security was adequate in early days. However, computer viruses could infect a machine by means of storage media like floppy disks. Some researchers have also analyzed and categorized operating system vulnerabilities that have been exploited in the past [33].

With an increase in network connectivity, open communication over untrusted channels had to be secured. An untrusted communication channel was modeled as one under the control of an adversary who was allowed to eavesdrop and tamper the messages passing through the channel. Cryptographic primitives were developed to solve the problem of source and message authentication and secrecy. Public key infrastructures were deployed

that enabled secure communication with trusted services over the internet.

With virtually all computers networked and reachable from any where in the world, a vulnerability in any computer provides a malicious entity a chance to launch an attack. This has been witnessed with increasingly sophisticated attacks caused by viruses, worms and others that exploit vulnerabilities in systems. Intrusion detection techniques were developed to identify malicious patterns in both network traffic and system usage. While these techniques are becoming increasingly effective in detecting attacks, it is almost impossible in practice to prevent compromises altogether. Hence, it is essential to model compromised servers and develop distributed services and protocols that can withstand compromises or failures.

Experiences with unexpected server behaviors due to causes like hardware failures lead the fault-tolerance community to define various fault models and develop fault tolerant systems. These fault models characterized essentially benign and accidental faults. Lamport defined a fault model called Byzantine fault [32] which encompasses arbitrary behavior of the faulty node, including malicious execution. He defined the Byzantine General's problem, the problem of achieving consensus in the presence of malicious players. Systems that tolerate a limited number of Byzantine faults have been designed. All compromised nodes are viewed to be under the control of a single adversary, allowing the compromised nodes to collude with each other for a common goal. Protocols were developed to achieve Byzantine agreement in synchronous distributed systems which typically involve three rounds of $O(n^2)$ communication among all the servers. It was proved that a system of n servers can tolerate a maximum of $\lceil (n-1)/3$ malicious servers at any time.

2.2 Replication Techniques

Replication and redundancy are key techniques to achieve fault-tolerance and high availability in distributed services. Replication techniques were successfully applied to tolerate Byzantine faults in a distributed environment. Schneider presented a generalized state machine replication approach to fault tolerance [54] using the Byzantine agreement protocol as a building block. In this approach, each request is directed to a designated leader who

forwards the requests in the order received to all other servers in the system. Fault tolerance is achieved by the Byzantine agreement protocol that allows all non-faulty servers to arrive at a consensus on both the order of the requests and the value of each request. Malicious and faulty behaviors of leaders can be identified by other servers and the leader can be changed over time.

Recently, Liskov and Castro gave a practical implementation of the state machine approach for a file system that tolerated Byzantine faults [35]. They eliminated public key operations by using Message Authentication Codes and showed that the overhead of using their file system was only 3% over NFS in the normal case. The file system was implemented in a local area network.

Data replication leads to data consistency issues and various replication techniques have been developed to tolerate malicious faults that vary in their consistency models. We will first review the consistency models and then revisit replication techniques.

2.2.1 Consistency Models

Data consistency captures the ordering guarantee a storage service makes for the read and write operations submitted by one or more clients. Consistency models are particularly useful when data is distributed at multiple sites and is shared among clients. Various consistency models have been defined and used in the past.

The most intuitive and easy to understand model is the single copy semantics or atomicity where reads and writes are ordered consistent with the order in which they appear in physical time [31]. This would be the case when the data item is stored at a single location and operations are performed one at a time. Typically guaranteeing this strong level of consistency is costly in a distributed store. This consistency model can be systematically weakened to allow efficient implementations yet meet the needs of many applications.

Weaker consistency models have been proposed for systems like distributed file systems, distributed shared memory and multiprocessor systems [31, 30, 20]. Safe semantics [31] guarantees that there is a total ordering among write operations and that a read operation that is not concurrent with any write operation will return the latest the value. When

the read is not concurrent with a write operation there is no guarantee on the value returned. Regular semantics is somewhat stronger than safe semantics and requires that a read operation return either the latest value or a value written by one of the concurrent writes [31].

Even weaker levels of consistency can be defined that are useful to many applications. For example, the causal consistency model [6] permits more efficient implementations and can meet the needs of many applications. In this model, while the actual ordering of concurrent reads and writes can be arbitrary, it is guaranteed that causally related operations are ordered the same way at all nodes. Causality is the happens-before relation that is defined in [29].

Although weaker consistency models may be appropriate for some applications, no single model may meet the consistency needs of all applications. Thus, several consistency levels may have to be provided in the same system [56, 63]. The Bayou system best exemplified this approach.

2.2.2 Replication Techniques and Consistency

The state machine approach offers single copy semantics, with all service requests strictly ordered like in the case of a single centralized server. The state machine approach does not scale well with large number of servers for the reason that each request requires two or more rounds of communication involving all the servers.

Quorum systems are popular for managing replicated data. In a quorum scheme, operations are done with sets of servers (quorums) that sufficiently overlap with each other to tolerate some number of malicious failures. Quorum systems can offer consistency levels like safe or regular semantics. Quorum approach eliminates multiple rounds of pair-wise communication and greatly reduces the load on the servers. Thus, quorum systems are much more scalable than state-machine based systems. Byzantine quorum systems have been developed to tolerate malicious servers and clients, retaining the scalability and better performance of regular quorum systems.

The Phalanx [39] and Fleet [41] systems were built using a quorum approach to tolerate

Byzantine faults. Alvisi et. al. presented a scheme to dynamically change the threshold value (the number of failures to be tolerated) based on the estimated number of faults perceived [7] and thus avoiding the use of large quorums when the actual number of compromised nodes is small.

Although access cost are reduced to some extent by weakening the consistency guarantees, it can still be quite high for the class of applications that this thesis targets. Most applications that require long-term confidentiality for stored data also exhibit limited data sharing characteristics. Thus, weaker levels of consistency like causality would be a better choice if, in particular, using such a consistency model would result in significant performance benefits. The protocols we develop in the next chapter will be enhanced later to guarantee such weaker consistency levels.

2.3 Protecting Data Confidentiality

Replication-based approaches do not offer data confidentiality unless encryption schemes are used. When data is replicated among several servers, data has to be encrypted before storing it to ensure data confidentiality. A client, which does the encryption, would hide the key from the servers, so that a compromised server cannot disclose any information stored at it. When the data so encrypted is shared among a dynamic set of clients, key management becomes an important issue. Whenever a client leaves the set of authorized clients, data has to be re-encrypted using a new key, and the new key has to be distributed to the remaining clients. Furthermore, keys have finite lifetime and data has to be periodically re-encrypted using a new key. All replicated copies have to be renewed. If one of the servers is compromised, such a server could retain a copy of the data encrypted with the old key, and content of long-lived data could be leaked over time. Thus, even if data is encrypted with a key that only authorized clients know, storing all the information in encrypted form entirely at any server site would lead to a possibility of information leakage. Thus clearly, encryption mechanisms alone does not suffice to guarantee long-term confidentiality of data stored at untrusted servers.

2.3.1 Secret-sharing Algorithms

Alternatively, secret sharing schemes that do not use encryption keys were developed to offer data confidentiality even when some number of nodes are compromised. A (b, k) secret sharing scheme [55, 9] transforms a data item into k pieces (called data shares or fragments) such that b or fewer shares do not give any information about the data content and any $b + 1$ shares can be used to reconstruct the original data value. This scheme can be used to guarantee both data confidentiality and data availability when the number of compromised nodes is not more than b. Secret sharing schemes offer confidentiality through access control at the servers as opposed to encryption schemes that are based on problems that are hard to compute.

Shamir and Blakley gave simple secret sharing schemes based on polynomial interpolation and intersection of hyper planes, respectively [55, 9]. In Shamir's scheme, to tolerate b malicious parties, a random polynomial in a finite field of degree b is generated whose y-intercept is the secret to be shared. The values this polynomial takes at various points constitutes the secret shares. With $b+1$ or more shares, the polynomial can be interpolated and the secret recovered. However, with b or less shares, no information can be obtained about the secret. Blakley's scheme achieves similar properties with shares being the hyper planes in a n-dimensional space that intersect at a random point whose first co-ordinate is the contained secret.

A number of other schemes have also been developed [50, 53, 26] building on the basic secret-sharing idea that trade security for reduced space overhead. Tompa et. al. [57], Feldman [15], Pederson [46] and Krawczyk [25] have also considered malicious corruption of shares either by servers or by a client.

More recently Herzberg et. al. developed a proactive secret sharing scheme where servers could proactively recover and renew their shares in a distributed manner to protect information against an adversary who can dynamically compromise nodes [22]. An adversary in possession of renewed shares cannot infer any information about the secret even if he possesses some of the old shares, as long as he does not have more than b among a single set of shares. This scheme is particularly useful for defending against long-terms adversaries.

As an adversary compromises more and more nodes over a long period of times, servers keep ahead of the adversary by renewing the shares periodically. Such a scheme would protect the confidentiality of the secret as long as adversary cannot compromise more than b servers between two consecutive share renewals.

Herlihy and Tygar developed a scheme where data is encrypted and the key is secret-shared [21]. Krawczyk presented a computationally secure secret sharing scheme combining secret sharing with encryption and Rabin's information dispersal [26]. Naor and Wool presented a scheme in which access control servers are different from storage servers [44]. However, they considered the case of benign server faults and malicious clients.

Secret sharing schemes, at the expense of higher communication and computational cost, eliminate the problem of key management. They also provide a means for long-term confidentiality. While a pure secret sharing scheme offers better confidentiality, such a scheme would result in high access cost. For example, consider a system of n servers. Let us assume that not more than b servers are compromised. Consider transforming a data item into n shares using a (b, n) secret sharing scheme . Writing such a data item would involve contacting all n servers. Reading would involve contacting a minimum of $b + 1$ servers and up to $2b + 1$ servers when b servers are compromised. When n is very large, write cost could be significantly high even when the number of compromised servers is small.

The cost of write operations could be reduced by allowing a client to write only $2b + 1$ data shares and then generating rest of the shares at other servers from the already written ones [22]. Such a generation of each data share involves one or more rounds of $O(b^2)$ messages exchanged between $2b + 1$ or more servers. Thus, generating new shares at other servers is a costly and time-consuming process.

Yet another approach is one that combines data replication with key secret sharing. In this scheme, a data item is encrypted and the encrypted version is replicated across servers. The key that is to be shared among clients is fragmented using a secret sharing scheme and distributed across servers. Thus, any client that wants to access a data item would first obtain the key by contacting $b + 1$ servers and then access an encrypted copy of the data item. This is the approach taken by Herlihy et. al. [21]. An extension to this scheme is

the scheme proposed by Krawczyk [26] which secret-shares encrypted data using Rabin's information dispersal algorithm [50]. We consider this scheme as one of many secret sharing schemes that our proposed system could use as a basic building block.

Clearly there is a need for a scheme that offers the security guarantees of the secret sharing schemes while also retaining the desirable features of the replication approach, such as better performance and higher availability. The scheme we have developed, described in the next chapter, combines these two approaches in a flexible fashion and offers the benefits of both these schemes.

2.4 Secure Dissemination

Secure store, like many other replicated systems, utilizes lazy replication in the background via a secure gossip-style dissemination protocol to improve performance of read and write operations. Using the dissemination protocol, data written to a small set of servers are propagated to other servers in the background. We briefly review here work related to secure dissemination.

Dissemination protocols have been extensively studied in benign environments in the past [13, 49, 18, 51, 45]. Most of these protocols are gossip style protocols. Gossip style protocols disseminate data by requiring each server to exchange information with a randomly chosen partner in rounds of gossips. Such protocols result in robust and scalable dissemination.

Malkhi, Mansour and Reiter [36] were the first to consider dissemination in malicious environments without using public key signatures. In [40], Malkhi et. al. presented a class of protocols that took advantage of a well defined logical structure of the system. In all these earlier protocols, a server accepts an update only if $b + 1$ other servers inform the server that they have accepted the update. These protocols are conservative in nature, where a participating server cannot help in dissemination until it accepts the update.

Malkhi and others [37], and Minsky and Schneider [43] independently suggested a class of gossip protocols where a server accepts an update if it receives the same update via $b + 1$ non-intersecting paths. Diffusion times for these protocols were $O(log\ n + b)$ and

$O(log\ n) + b$ respectively. Message size and buffer requirements for Minksy's protocol were particularly low. Both these protocols and the earlier ones [36, 40] do not rely on any cryptographic primitive and are information theoretically secure as opposed to the protocol we present in chapter 4. We have developed a gossip style protocol for dissemination in malicious environments which takes $O(log\ n) + f$ rounds to disseminate an update, where f is the actual number of malicious servers in the system.

Our protocol uses a novel key allocation scheme and a technique called collective endorsement that replaces public key signatures with a list of message authentication codes. Schemes allocating sets of symmetric keys to participating nodes have been used in multicast key management applications [60, 16]. Naor and others also suggested using a list of MACs to replace public key signatures in [10]. They used a key allocation technique which gave a probabilistic guarantee against forgery by a limited coalition of malicious servers. Liskov and Castro [34] eliminated public keys in Byzantine fault tolerance by replacing signatures with a vector of MACs. An exclusive symmetric key was shared between every pair of servers.

2.5 Other Projects

A number of other systems have been designed for large-scale Byzantine fault-tolerant storage [3, 2, 23, 58, 64]. Notable among these are Oceanstore [3] and Farsite [2] projects. Oceanstore is a global scale persistent store scaling to thousands of machines or even more. Unlike *secure store*, they use redundancy coding techniques for fault tolerance and rely on encryption for secrecy. Replication is achieved by on-demand caching in contrast to proactive replication in *secure store*. Farsite is a peer-to-peer system where a set of nodes can decide to co-operate with each other to offer a virtual file system. Similar to *secure store*, a set of nodes together maintain a metadata service for the file system. Replication is done in a much more controlled manner, always keeping a check on the number of copies of each object.

E-vault uses Rabin's Information Dispersal Algorithm to store data fault tolerantly by

web-based clients through a web interface. SITAR provides a generic architecture to compose secure services using available COTS software [58]. CoCa is a certification service that uses quorum techniques for service availability and secret sharing and threshold computation techniques for maintaining secrecy [64].

The PASIS project [62] at CMU addresses a number of problems related to this dissertation. PASIS considers various secret sharing and other schemes to store data securely in a data repository. However, PASIS does not consider integration of replication and secret sharing. Fray et. al. proposed an approach similar to ours, fragmentation-scattering [17], where fragments of cipher text are replicated. However replication was achieved by clients broadcasting the fragments. They did not consider fragment dissemination or periodic renewal of fragments.

While our research builds on work done by others in the past, there are a number of novel contributions it makes to the area of building secure storage services. Our research is the first attempt at building an agile secure system. We have designed a new data storage scheme that combines replication with secret-sharing and offers a continuous tradeoff between security and performance. The dissemination protocol we have developed can disseminate information faster than other protocols. More importantly, this protocol illustrates the principle of agility which states that the price paid for security should be in proportion to the amount of actual malicious activity and that the performance should be comparable to benign-case system in the common case when there is no or little malicious activity. We are the first to address weaker consistency in secure distributed storage. We have developed techniques to access data with a weaker consistency consistency guarantee that can result in substantial performance benefits.

Our research efforts described in this thesis are complementary to other techniques developed towards building secure and agile systems like intrusion detection [5, 47], fault diagnosis [24], reconfigurable quorum systems [24, 7], and flexible and efficient access control and authentication infrastructure. [52, 48].

CHAPTER 3

SECURE STORE ARCHITECTURE AND PROTOCOLS

3.1 Overview

This thesis addresses the problem of building a secure storage service that is distributed among a number of servers some of which can be compromised. The store guarantees integrity and confidentiality of stored data and is designed to be highly available, even when some of the servers are compromised by malicious entities. The storage service adheres to the principles of agility offering clients explicitly a tradeoff between security and performance. Thus, with appropriate feedback mechanisms, overheads for better security will be incurred only at times of high levels of threat when malicious activity actually takes place and only by those applications that require high levels of security guarantees.

In this section, we introduce the system model for *secure store* and list the assumptions we make. We present the basic architecture of *secure store*, discussing briefly the various components that go into building the store. We describe in detail the data storage scheme and the associated data access protocols we have developed. We also do a preliminary analysis of the data storage scheme using a probabilistic model for server compromises and show the tradeoffs our architecture can offer between security and performance.

3.2 System Model and Assumptions

Secure store is implemented by a set of n servers designated as data servers, some of which can be compromised by an adversary and exhibit malicious behavior. Clients make read and write requests with subsets of servers by sending a request message to each of the servers and possibly getting replies for read requests. For the purpose of discussion in this chapter, we assume nodes and the communication network to be synchronous and the network to be reliable. In practice, in our implementation, we use acknowledgement messages and timeout mechanisms to detect server failures, network outages and message losses.

Each request is authenticated and authorized individually by every server. Hence, we assume the presence of appropriate public key infrastructure. Each client and server node has a private key for which the corresponding public key is well known. Besides these keys, clients and servers also negotiate symmetric keys periodically to exchange messages. Thus, all communication channels are made secure against eavesdropping, modification and replay attacks and our protocol messages are exchanged over secure channels. Client requests are authorized by servers using unforgeable capabilities issued by a separate authorization service.

3.2.1 Threat Model

We assume a single powerful adversary who can compromise any server and the compromised server would be under the complete control of the adversary. However, we limit the rate at which the adversary can compromise servers. We assume a Byzantine fault model for compromised servers, that is, a compromised server can behave arbitrarily. Any compromised server can disclose data stored at the server, corrupt the data and possibly collude with other compromised servers.

We assume that during any continuous time interval of length T_v, a server can be compromised with a probability p. We refer to the constant T_v as the vulnerability window. We assume that the probability of compromise of a server is independent of other servers being compromised. Thus, we do not consider the case of related compromises. Both system architecture and protocols are designed to tolerate a maximum of certain number of Byzantine faults. This number, denoted by b, is referred to as the threshold value of the system in this thesis. This is a parameter chosen for each data item by the client at the time a data item is created. For a given n and an assumed p, the expected number of failures in a time interval of length T_v is np. However, threshold value b could be set to a lower or a higher value to tolerate a different number of failures depending on whether better performance or better security is desired. For most part of this chapter, we assume that a threshold value b has been chosen and that in any time interval of length T_v, not more than b servers are compromised. In our theoretical analysis in a subsequent section,

we will show how the choice of b affects security and performance for a given probability of compromise.

In this thesis, we assume that clients are not malicious. In practice, clients may become compromised and exhibit malicious behavior. They can try to disrupt the service by performing incomplete or inconsistent writes and leaving the system in an incorrect state. Malicious clients may also try to mount denial of service attacks, exhausting server resources.

First of all, even if the store we build defends against inconsistent and incomplete writes by malicious clients, malicious clients may write garbage to the store at the application level that would look like valid writes to the underlying system. Besides, defending against malicious clients is very difficult to achieve even when significant costs are incurred by our protocols. The agile store project and others are developing intrusion detection techniques that would identify malicious attempts by clients. We advocate restricting damage done by malicious clients by detecting malicious activity at client nodes and denying access to malicious clients by removing them from ACLs. For the rest of the thesis, we will assume that clients are not malicious.

3.3 Overview of Architecture

Figure 2 shows the architectural framework of *secure store*. In the following, we describe briefly various components of the framework.

- *Public Key Infrastructure* : We assume the existence of a public key infrastructure to allow nodes to authenticate each other and set up secure channels. Each server and client has a private key for which the public key is well known. This provides an authentication framework which is used to allow clients and servers to authenticate each other. This infrastructure is used to set up a secure channel (negotiate symmetric keys for communication) between any two nodes that need to communicate. Communication channels are assumed to be secure against eavesdropping, modification and replay attacks.

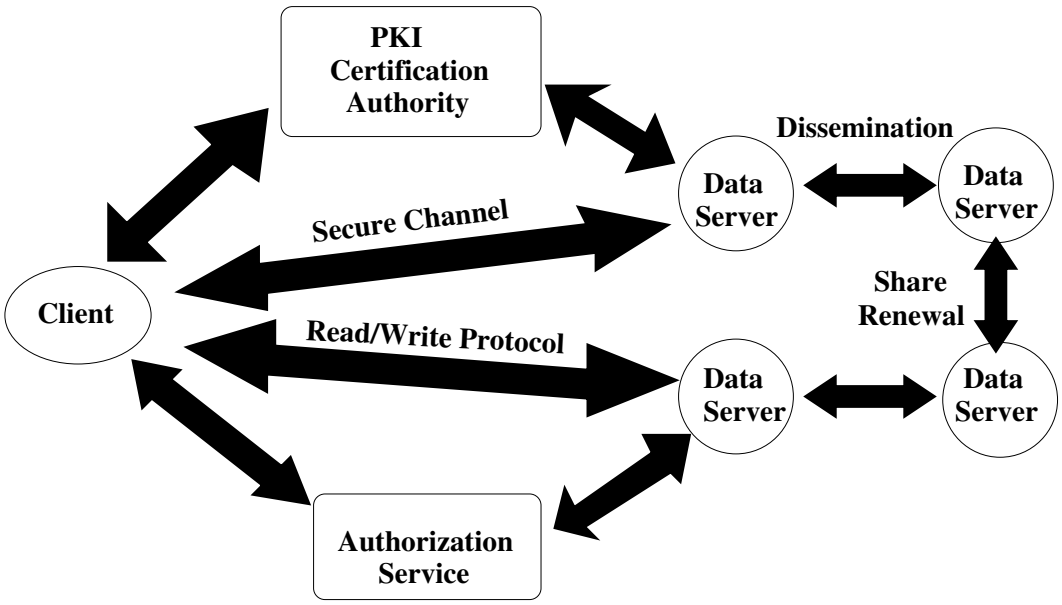


Figure 2: Secure Store: Architectural Framework

- *Authorization Service* : We also assume the existence of an authorization service which maintains ACLs for each data item stored in the store. The authorization service is a Byzantine fault tolerant state machine implemented by a small set of servers. Authorized clients make changes to ACLs by interacting with the authorization service. Authorization service helps data servers enforce access control for client requests by issuing unforgeable capabilities to authorized clients. Data servers require valid capabilities before granting access to a client.

- *Data Servers* : Secure store is implemented by a group of data servers that are distributed over a wide area network. Some of these servers can be compromised by an adversary. Clients make read and write requests to a subset of servers. Servers are primarily involved in storing data and servicing client requests to store and retrieve data. Data is stored at the servers in the form of secret shares computed by the clients using a secret sharing scheme. Servers also store metadata associated with each data item. This metadata includes, among other information, *uid* of the data item, parameters that were chosen when the data was written and information that helps clients make data access decisions according to their consistency requirements. Each of the servers

authenticate and authorize each client request independent of other servers. Servers make authorization decisions based on unforgeable capabilities that the authorization service issues to clients. Servers participate in a background dissemination process in which they exchange new updates. Some of the servers are dedicated to renewing data shares periodically. The share renewal servers are located close to each other (e.g, in the same LAN) with a broadcast facility.

- *Clients* : Clients make read and write requests to a subset of the servers. The choice of the subset of servers depends on the proximity of the servers to the client site or could be completely random, restricted only by the logical topology of the servers. A client authenticates itself to a server and sets up a secure channel before data shares or other information is transferred in either direction. A client also presents a capability that is issued by the authorization service along with a request. The only long-term keys a client site needs to store are (1) its private key that it uses to authenticate itself to others, and (2) the public key of the root certification authority of the assumed PKI. Other keys that are stored at a client site are short-lived symmetric keys that are negotiated with servers and are not stored beyond a session. Secure store offers only a storage abstraction to the clients. It's up to the clients to use this storage service in a manner suitable for its applications, e.g, a file system implementation that uses the secure store. Hence, secure store does not maintain a name-space hierarchy or other file system related information. Such abstractions could be offered by separate services. A file system that uses secure store has been implemented and is described in chapter 6. In our implementation, name-space hierarchy is maintained by a separate service called metadata service.

- *Data Access Protocols* : These are a set of read/write protocols that are necessary to provide clients access to data. The data storage scheme and the read/write protocols will be discussed in detail in the subsequent sections. Upon a write, a client determines the parameters to be used to store the data item, uses a secret sharing algorithm to generate shares of the data item according to the chosen parameters, generates a

unique identifier for the the data item and sends one share each to the chosen subset of servers along with the *uid*. Upon a read, a client collects shares pertaining to a single write from a subset of servers and assembles them back to the original data item. Protocols are designed with the assumption that clients do not act maliciously.

- *Data Dissemination* : Data shares that are stored at one set of servers are replicated at other servers in the system via a secure dissemination protocol. Dissemination makes data available for access at other servers and improves system performance. Servers exchange updates seen by each other by participating in a gossip style dissemination protocol that runs in the background asynchronously. Each server periodically chooses another server at random and pulls updates seen by the other server. The protocol is designed to tolerate a limited number of faulty servers that can act maliciously. The number of malicious servers tolerated is same as the threshold value b chosen for storing a data item.

- *Periodic Share Renewal* : A set of servers dedicated for share renewal renew shares of data items that require long-term confidentiality periodically. This preserves the confidentiality of data items against a dynamic adversary who compromises a large number of servers over a prolonged period of time. In [22], Krawzcyk presents a share renewal protocol for Feldman's secret sharing scheme [15]. Secure store uses Feldman's scheme for secret-sharing and the protocol proposed in [22] for share renewal for long-lived sensitive data. For other data items that do not require share renewal, some of the cheaper secret-sharing schemes that are surveyed in [62] could be used.

3.4 Data Storage Scheme

3.4.1 Background and Approach

Replication has been the key to fault-tolerance and high availability in distributed services. Replicated storage has been studied well in the literature. Traditionally, security in such systems has been addressed by requiring the clients to encrypt the data before storing it at the servers. Decryption key is made available only to authorized clients. However, such a scheme has certain drawbacks as was discussed in section 2.3.

We take the approach of secret-sharing the information among a set of servers. A (b, k) secret sharing scheme [55, 9] transforms a data item into k pieces (called data shares or fragments) such that any b shares do not give any information about the data content and any $b+1$ shares can be used to reconstruct the original data value. This scheme can be used to guarantee integrity, confidentiality and availability of data when number of compromised servers is not more than b. Secret sharing schemes offer confidentiality through access control at the servers as opposed to encryption schemes that are based on problems that are hard to compute. Some secret sharing schemes also allow periodic renewal of shares by the servers without client participation [22]. If the adversary is limited to compromising no more than b nodes in any time-interval of certain length, say T_v units, by doing share renewal at a faster rate (more than once every T_v units), no information would be leaked to an adversary ever. Thus, secret sharing schemes are capable of guaranteeing lifetime secrecy of data content.

While it offers better confidentiality, a pure secret sharing scheme would result in high access cost and poor performance. We integrate the two approaches, replication and secret sharing, to meet both performance and security requirements of applications. Figure 3 gives an overview of the secure store's approach to data storage. The set of servers is arranged logically in a two-dimensional matrix. A data item to be stored is secret-shared and stored as shares at servers along a row. The data shares are replicated along the column asynchronously in background using a dissemination process. Periodically, shares are renewed by a dedicated set of servers. No data share is stored at any server for a long period of time. After a period of time, a share stored at a server is either replaced by a renewed share or erased. The degree of replication (number of rows in the matrix) and the degree of secret-sharing are chosen parameters that determine the performance and security levels of the system. When the number of columns is one, the system corresponds to a purely replicated one. As the degree of secret-sharing is increased, system becomes more secure against server compromises at the cost of low availability and high access costs. When the number of rows is set to one, the system degenerates to a purely secret-shared one.

3.4.2 Data Storage Architecture Overview

We will first describe a system which supports only one set of parameters (including the threshold value b) for all stored data items. We will later extend the architecture to support storage of data items with different parameters.

The set of n data servers is arranged logically in a two-dimensional matrix as shown in figure 3. The server in i^{th} row and j^{th} column is denoted by S_{ij}. Number of columns is denoted by c and number of rows r with $rc = n$. We assume that c is at least $b+1$. A data item is transformed into c shares using a (b, c) secret sharing scheme and is stored as shares at the servers. For a particular data item, servers along a column store copies of the same data share and each column stores a different share. Both security and performance levels change with values chosen for b and c, as we will show in our analysis.

The operation of our secure store is characterized by the following three sets of protocols:

1. Read and write protocols that are used by clients to access the data.
2. A dissemination protocol which is used by the servers to propagate new data shares among themselves.
3. A share renewal protocol that is run periodically to generate new data shares for long-term confidentiality.

Data consistency becomes an issue when data is replicated. We postpone the discussion of data consistency until chapter 5. Our store is capable of offering two kinds of weak consistency guarantees, namely Monotonic Read Consistency and Causal Consistency. For the sake of clarity, the read and write protocols we present in this section do not address consistency requirements.

In this thesis, we use a specific verifiable secret sharing scheme due to Feldman [15] for our protocols. The use of this scheme, as discussed in [22], results in (1) easy verification of a share during dissemination, (2) renewal of shares in a purely distributed fashion, and (3) regeneration of lost shares. We could use any secret sharing scheme that has these

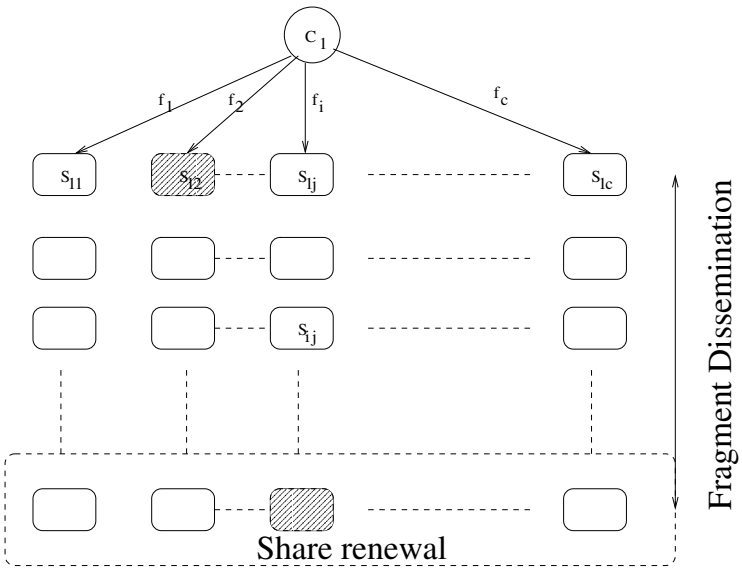


Figure 3: Secure Store: Data Storage Architecture

properties. In the following subsections, we discuss the protocols for reading and writing data, dissemination and share renewal.

3.4.3 Data Access Protocols

We do not consider the case of clients being malicious. While malicious clients cannot do any harm to data items for which they do not have access, they can, however, exhaust a server's storage or write garbage to the data items for which they have write access. We rely on detecting malicious clients and using authorization and access control mechanisms to stop malicious clients from doing harm.

Figure 4 shows the write and read protocols when all clients are assumed to be non-malicious. In a write operation, for a chosen fault tolerance threshold b and the number of columns c, a client transforms a data item into c shares using a (b, c) secret sharing scheme. One-way functions $h(x)$ are computed for these shares and concatenated to form a verification string. We discuss issues related to the choice of an one-way function in the following subsections. Verification string is required to let a server know if a share it received in dissemination has been corrupted. Verification string also helps a reading client choose the right set of $b+1$ shares to reconstruct the data. For write, the shares are sent to servers

Write(x_j, v) by client C_i:

1. Let timestamp ts = current clock value concatenated with uid(client).
2. Fragment value v into c shares v_1,v_2,..v_c using a (b, c) secret sharing scheme.
3. Compute one-way function of each of the shares, $h(v_i)$.
4. Form the verification string VS and compute signature
   $VS = h(v_1)|h(v_2)...|h(v_c)$. (concatenation).
   $sig = \{uid(x_j), ts, v\}_{K_{c_i}^{-1}}$, where $\{\}_{K_{c_i}^{-1}}$ denotes signature using private key of the client.
5. Choose a row k.
   for ($m = 1$ to c){
   send {"write",uid(x_j),ts,v_m,VS,sig} to server S_{km}.
   }
6. Repeat 5 for a different row until $c - \lfloor b/l \rfloor \geq b + 1$ where l is the number of rows contacted.

Read(x_j) by client C_i:

1. Choose a row k.
   for $m = 1$ to $2b + 1$ {
   send {"read",$uid(x_j)$} to S_{km}.
   }
2. Receive a list of timestamps from each server with the corresponding data shares and verification strings.
3. A timestamp is said to be "good" if it appears in at least $b + 1$ replies (lists) and the corresponding verification strings are the same. Let t_r be the highest timestamp among such good timestamps.
4. If there is no good timestamp, repeat from 1 for a different k.
5. Pick shares corresponding to t_r. Pick $b + 1$ shares among these that are successfully verified by the verification string. Reconstruct the data value.
6. Check if signature is valid. If valid, return the reconstructed data value. If signature is not valid, repeat from 1 for a different k.

Figure 4: Write and Read Protocols

along a row, each receiving a different share along with uid of the data item, timestamp, the verification string and the signature of the whole write. Since some servers that are contacted can be compromised, the write is repeated with different rows until the number l

of rows contacted is such that $c - \lfloor b/l \rfloor$ is at least $b+1$. Since maximum number of columns in which all l servers contacted are compromised is $\lfloor b/l \rfloor$, writing to l rows ensures that in each of $b+1$ or more columns, at least one non-malicious server has received the write message.

While reading, a client sends a read request for the object to servers along a randomly chosen row. It collects $b+1$ or more shares corresponding to the same timestamp and reconstructs the data value. Finally it verifies the signature before accepting the value. If the read is not successful, the client contacts additional servers. Variations of this protocol are possible by varying the number and choice of servers a client contacts initially and by varying the way additional servers are contacted.

3.4.4 Dissemination

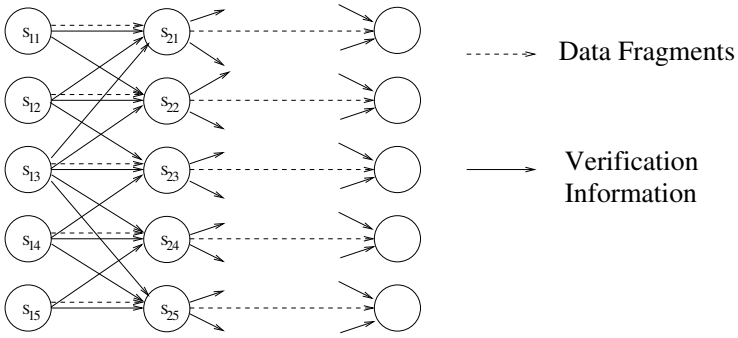


Figure 5: Secure Dissemination

Our write protocol may write each share to only one server in a column. To provide better performance and availability, shares written at one set of servers should be disseminated to other servers so that the data is available for access at other servers. Hence, in the secure store, shares are disseminated along columns. Data dissemination for non-malicious environments was studied in [13]. Presence of malicious servers requires a more careful treatment.

Dissemination of verification string:

1. During a write operation, client writes VS to all servers it contacts(at least $2b + 1$).
2. A server accepts a VS as valid for a write only if it hears from a client directly or when $b + 1$ other servers have accepted the same VS as valid.
3. A server disseminates a VS to all other servers, both within and across columns.

Dissemination of shares:

1. A server disseminates the shares it receives to other servers in the same column. Shares do not have to be verified before being disseminated to other servers.
2. A server accepts a share as valid if the share is successfully verified by a valid VS.
3. Shares can be served to clients even before verification.

Detecting corruption and generating correct shares:

1. A server detects corruption if it receives two or more different shares for the same write.
2. A server can probabilistically suspect corruption of the shares it holds.
3. Upon detection or suspicion, a server uses a valid VS to check the validity of the shares. If the server does not have a valid VS, it pulls VS from other servers.
4. If a server finds a share to be corrupted, and does not find the correct share with any other server in its column, it initiates a share recovery protocol with servers from $b + 1$ or more other columns.

Figure 6: Dissemination Protocol

Byzantine nodes can modify the data being disseminated and thus can compromise the availability and integrity of the data. One approach for secure dissemination is to attach an unforgeable signature of the writing client to the data being disseminated. We disseminate data shares rather than the data item itself. Since shares could change over time due to periodic share renewal (discussed in section 3.4.5), the client has to recompute the signature of the shares. However, it is a desirable feature to do share renewal without requiring client participation, as we will see in section 3.4.5.

Secure dissemination schemes without public key signatures have been addressed by others [36, 37, 43]. These schemes require data to be written to at least $b+1$ non-faulty nodes initially before being disseminated. We cannot readily use these schemes to disseminate data shares for the reason that each share is written to only one server initially.

We use verification strings to secure the dissemination of data shares. Usual signature verification is replaced by a set of one-way function verifications. Thus, a server verifies a data share f it receives in dissemination by checking it against the one-way function of the share $h(f)$. The verification string, which is a concatenated list of one-way functions of the shares of a data item, should be fully reliable since a corrupted verification string can verify a manipulated share to be correct. Hence, verification string itself is written by the client along with shares and disseminated to other servers securely using any of the secure dissemination schemes [36, 37, 43, 28]. In the next chapter, we present a dissemination protocol we have developed for this purpose that has some desirable features. We require that the verification string be disseminated across columns, among all servers. A server accepts a verification string as valid only if $b+1$ or more servers are known to have accepted the same verification string for a given timestamp. Once a server accepts a list of one-way functions as valid for a timestamp, it accepts the corresponding share by verifying that the one-way function applied on that particular share matches the corresponding part of the verification string. This way, a compromised server cannot modify and disseminate a corrupted share without going undetected, even when colluding with other malicious servers.

Hence, our dissemination protocol works in two parts: (1) dissemination of verification string, (2) dissemination of shares. Figure 6 describes the dissemination protocol. We have presented here a simple protocol for dissemination of verification strings. In the next chapter, we will replace this simple protocol with a protocol we have developed using a novel technique called collective endorsement.

In addition to dissemination of shares and verification string, we add a component to detect corrupted shares and regenerate correct shares. Servers also probabilistically suspect corruption of shares and verify the correctness of shares by pulling verification strings from other servers. Share recovery involves getting secondary shares from $b+1$ or more other

servers, each from a different column, to recover the corrupted or missing share. This scheme is described in [22]. Once the right share is constructed, this share is disseminated along the column. Share recovery is costly and we expect that, if there are only few malicious servers in the system, share recovery would be done only infrequently.

During share renewal, the shares of a date item change, but the new verification string can still be computed securely and reliably, even in the presence of active attackers during the phase of share renewal [22]. Thus, even after share renewal, we can disseminate the new shares securely as we did before share renewal.

3.4.5 Share Renewal Protocol

We assume that an adversary cannot compromise more than b nodes in any time frame of length T_v . However, this does not prevent an adversary from compromising $b+1$ or more nodes over a longer period of time and obtain $b+1$ or more shares to learn the content of a data item. Hence, the shares need to be periodically renewed, in a distributed manner, so that an adversary who obtains b shares before share renewal cannot use them in any manner in future to gain any information, even if he finds additional data shares. The share renewal of a data item is done without the participation of the client that wrote it. This enables share renewal even when the client is offline for an extended time period.

Share-renewal is very expensive and the frequency with which shares of a data item are renewed can be tuned depending on the sensitivity of the stored data item. Shares of data items that lose value over time are renewed less frequently as they age. Also, shares of data items that are frequently over-written by clients do not need renewal. Share renewal is done only for those data items that need long term secrecy.

In this thesis, we use the share renewal protocol proposed by Herzberg et. al. for Feldman's secret sharing scheme as discussed in [22]. For a given data item, servers belonging to one row initiate a share renewal protocol. At the end of share renewal, the shares are renewed while the data content and other meta-data are retained. The verification string is also updated securely for the new shares. The protocol guarantees that if the number of non-faulty servers is a majority, at the end of the protocol, all non-faulty servers

hold valid shares and a copy of the same valid verification string. From then on, the new shares are disseminated as before. No data share is stored at any non-malicious server beyond T_v seconds after its renewal. A share is erased either when a new share arrives or when the share expires. This is critical to guaranteeing confidentiality since share renewal schemes rely on erasing the old shares. Herzberg et. al. [22] assume a secure broadcast channel for share renewal protocol. In our system, we could dedicate a set of servers for this purpose, on a single shared wire, doing share renewal for different data items.

The one-way function used in Feldman's scheme is g^x where g is a primitive element in the field[1] from which values for x are chosen. While this works for our solution, this is costly in terms of storage space required (large verification string) and in terms of computation.

Data items that are over-written frequently by clients and those that do not have strict long-term confidentiality requirements do not need share renewal. For such data items, eliminating share renewal eliminates the need to use an expensive one-way function like g^x for verification string. In such cases, a simple digest function like MD5 could be used for $h(x)$. Data items that need stronger secrecy guarantee would use the more expensive one-way function g^x.

To save on storage space, we use $h(g^x)$ where h is a cryptographic digest like MD5 at the expense of incurring an additional round during share-renewal. Although verification using such expensive one-way functions is computationally intensive, this cost would not be incurred in the dissemination protocol in the common case when most of the servers are not malicious. Only when servers detect corruption of shares, or suspect corruption probabilistically, the verification cost would be incurred. However, reading and writing for such cases would be computationally expensive. So would be share-renewal in the absence of writes. This is a tradeoff clients are offered to decide if a data item should be stored at such a high security level. In place of the function g^x for $h(x)$, we could use any one-way function that satisfies certain properties as required by the share-renewal protocol.

[1] For the purpose of this paper, values for data items and data shares are assumed to be chosen from the finite field Z_p, for an appropriate prime p.

3.4.6 Incomplete Writes

Although we assume clients are not malicious, write operations may not be completed because of benign client failures and problems with network connectivity. Such incomplete writes, if left undetected, would overwrite the already stored value for a data item. Hence it is essential to deal with the case of incomplete writes. Making the protocols immune to incomplete writes would require servers achieving a consensus on the completion of write before overwriting a data value with a new one. This is very costly in the presence of malicious servers. We advocate repairing the damage after its done rather than preventing it. Data items can be stored in versions (creating backups) so that old values are not over-written before the the new write is verified to be complete. Incomplete writes can be detected with a time lapse in the background by requiring servers to exchange information about writes seen, as part of the dissemination protocol. This lazy consensus approach is a practical alternative to in-time consensus algorithms. Incomplete writes thus detected can be garbage collected and the storage space can be reclaimed. We do not discuss this issue further in this thesis and leave it as a topic for future work.

3.4.7 Supporting Multiple Security Levels

So far we have described *secure store* as a system that supports only one set of parameters (b and c apart from others like consistency level) for stored all data items. However, this architecture can be extended to support data items stored using different sets of parameters to achieve the desired levels of security and performance.

Each server is indexed between 0 and n. The row number and column number for each server differs from data item to data item. The row and column numbers depend on the server's unique index and the value chosen for c, the number of columns for a data item. The row number is integer quotient of the index when divided by c and the column number is the remainder of index modulo c. Thus, each node in the system (client or server) can calculate the logical topology of the servers for each data item and choose the servers appropriately to send read/write requests or for dissemination purposes.

Correctness and security of the *secure store* protocols depend on the correct values for b

and c. For example, in the dissemination protocol, a server accepts a verification string as valid if it infers that $b+1$ other servers have accepted the verification string. Share renewal protocol requires all non-malicious servers involved in the protocol to agree on the correct values for b and c. Thus, the values of b and c used by a server for a data item should be correct and reliable.

Since the clients are trusted, servers that receive a write request from a client trust the parameter values they receive directly from the clients. A server that receives these parameter values through dissemination for the first time trusts the values it receives from its dissemination peer. Although the dissemination peer may be malicious and might corrupt the values for the parameters, the justification for trusting such a value is that if the dissemination peer is malicious, whatever damage that could occur by trusting the parameter values from a malicious server could as well be done by the malicious server itself. Thus, trusting the parameters received through dissemination does not add new vulnerabilities. For example, if a wrong value for c leads to disseminating a data share to the wrong server, the malicious server itself could send the data share to the intended server.

Finally, a reading client has to infer correct values for the parameters while reading a data item. If the stored data item is accessed only by one client, the reading client knows about the correct values for the parameters. On the other hand, if the data item is shared between multiple clients, there should be a service that maintains the name space hierarchy and other metadata for the stored data items. This service can also securely maintain the values for the parameters. A reading client would read the parameter values from this service before reading the data item from the store. For example, in the file system we have implemented, described in chapter 6, a replicated state machine called the metadata service maintains securely the name space hierarchy and the associated metadata, including the storage parameters for each data item.

3.5 Analysis

In this section, we do an analysis of the data storage scheme of secure store based on a probabilistic model and show how the choice of a threshold value and other parameters

affect the security and performance of the system. During any continuous time interval of length T_v units, we assume that any server can be compromised with a probability p. Thus, expected number of compromised servers during a time interval of length T_v would be np. However, a lower or higher value can be chosen for b to tolerate certain number of failures depending on whether better performance or stronger security is desired. We assume that the probability of compromising one node is independent of the other. Thus, in the analysis, we do not consider the case of related or similar attacks on nodes operating on same OS or run time code. For the purpose of analysis we also assume that the system is in a steady state, void of concurrent reads and writes. Thus, we assume reads and writes do not fail because of consistency requirements. The analysis we provide here is a simplified one and its goal is to provide us with insights into how some parameters affect security and performance levels offered by the system. In particular, it gives us a direction as to what threshold value and the degree of replication should be chosen, given the desired levels of various security and performance metrics.

We consider the following security metrics.

- **Availability** : Availability is defined as the probability that a legitimate client can read a data item that has been written successfully.
- **Confidentiality** : Confidentiality is defined as complement of the probability that an adversary can read a data item that has been written successfully.
- **Integrity** : Integrity is defined as complement of the probability that a reading client could be returned corrupted data content without corruption being detected.

We assume that the servers are organized in c columns and r rows with $rc = n$, where n is the total number of servers. Furthermore, we assume that a (b, c) scheme is used for secret sharing. With these assumptions, the security metrics can be evaluated as follows:

Availability(α):

$\alpha(b, c, n)$ = probability of finding at least $b + 1$ non faulty servers, one each from a different column.

$$\alpha = \Sigma_{i=b+1}^{c} \binom{c}{i} (1-p^r)^i * (p^r)^{(c-i)}$$

Confidentiality(κ):

$\kappa(b, c, n)$ = 1 - probability of finding at least $b+1$ malicious servers, one each from a different column.

$$\kappa = 1 \ - \ \Sigma_{i=b+1}^{c} \binom{c}{i} (1-q^r)^i * (q^r)^{(c-i)}, \ q = 1-p$$

For our system, integrity is same as confidentiality since any compromised node can both disclose data shares and corrupt them. If use of signature is considered, integrity becomes the probability that a signature can be forged. In rest of the section, we discuss only confidentiality and availability as the primary security metrics.

In addition to these security metrics, for our system, we also define the following performance metrics.

- **Read cost** : Read cost is defined as the expected number of servers a client needs to contact to read a data item successfully. A data item is read successfully upon collecting $b+1$ distinct shares for the data item.

- **Write cost** : Write cost is defined as the number of servers a client needs to contact to write a data item at a confidence level h. By confidence level, we mean the probability that a write has been successfully completed, that is, at least one non-faulty server from each of $b+1$ or more columns has registered the write.

Both read and write costs are defined in terms of number of servers contacted. We expect the communication cost to be the dominant factor compared to the computation cost and hence discuss only communication costs in this section. This may not be true when we consider large data items. Secret sharing and data reconstruction are costly for such items. Furthermore, computation cost also increases with threshold value. Number of messages sent/received during a read or write operation is twice the number of servers contacted to complete the operation.

Read cost is calculated based on the following protocol: A client contacts $2b+1$ servers, each from a different column. If it has successfully collected $b+1$ shares, read returns. Otherwise, client contacts additional servers as necessary. If p_{rs} is the probability that a client would find $b+1$ non-faulty servers among the $2b+1$ it contacts, then expected number of servers a client needs to contact is approximately $(2b+1)/p_{rs}$.

When a client does a write, it writes to some number of rows so that the probability of finding at least one non-faulty server from each of $b+1$ or more columns is greater than or equal to h where h is the confidence level. If $p_{ws}(k)$ is the probability of a write being successful when a client writes to k rows, then write cost is $\mathbf{wc} * c$ where $\mathbf{wc}$ is such that $p_{ws}(\mathbf{wc}-1) < h$ and $p_{ws}(\mathbf{wc}) \geq h$.

Given the probability of a node being compromised and the total number of servers, there are two parameters that determine the values of the security and performance metrics. These are the threshold level b and the degree of replication which is the number of rows or alternatively the number of columns c.

We calculated the four metrics by varying these two parameters for a system of 45 servers with a probability of compromise $p = 0.15$.

Plots 7(a), 7(c) and 7(e) in figure 7 show availability and confidentiality. Both availability and confidentiality are plotted in logarithmic scale. For a value x plotted in the graph, the corresponding probability (availability or confidentiality) is $1-10^{-x}$. Plots 7(b), 7(d) and 7(f) show read and write costs. For plots 7(a) and 7(b), number of columns was varied for fixed values of threshold. For graphs 7(c) and 7(d), threshold value was varied for fixed values for number of columns. For graphs 7(e) and 7(f), threshold value was varied and number of columns was set to $2b+1$ accordingly.

We observe the following dependencies :

- For a given threshold level, increasing the number of columns (and hence decreasing the number of rows) increases availability and decreases confidentiality. Write cost increases linearly with number of columns but read cost remains almost a constant. In this case, the tradeoff is only between availability and confidentiality. Confidentiality decreases with higher write cost.

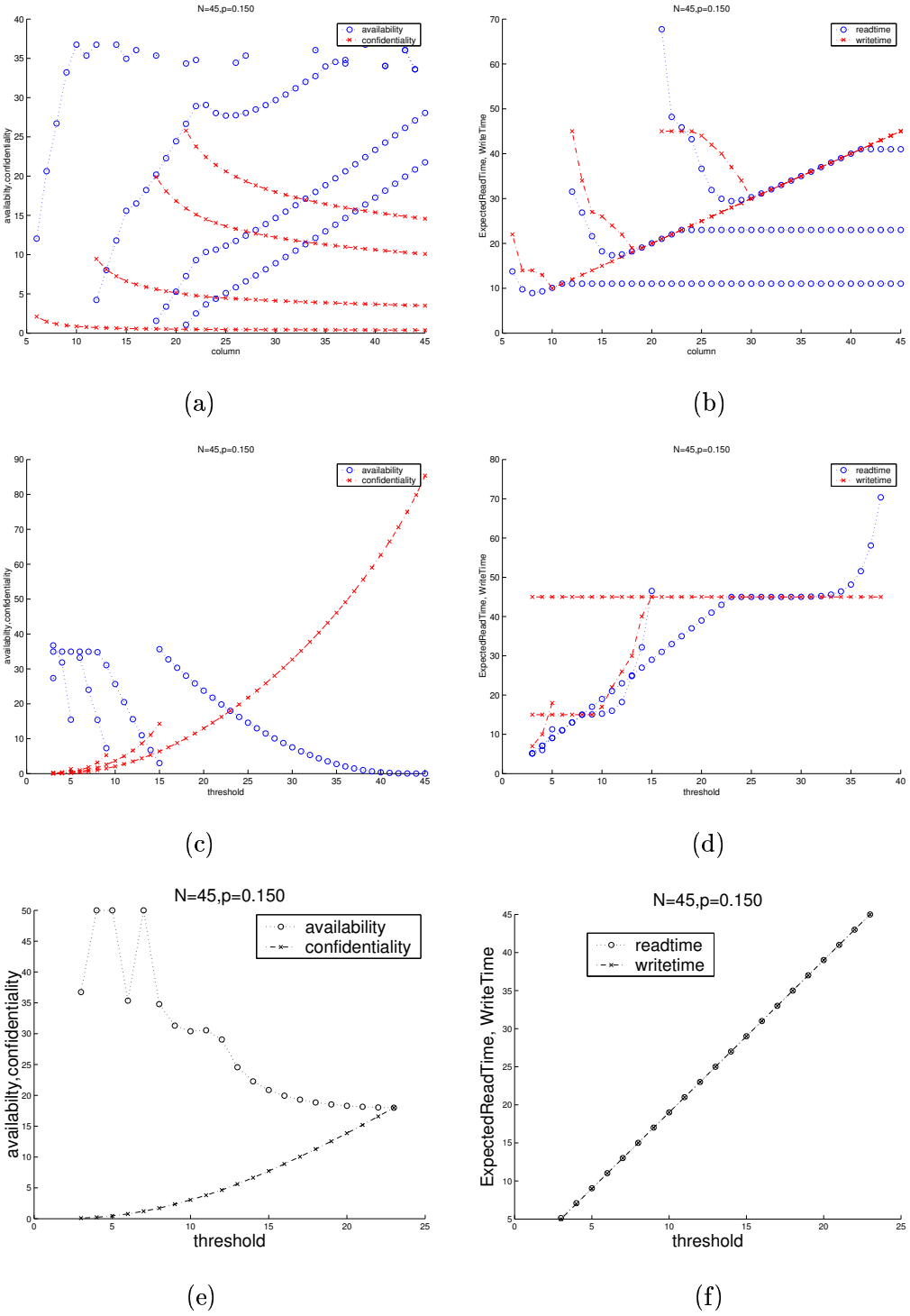


Figure 7: (a) α,κ as functions of number of columns for various threshold values b = 1,11,17,20, (b) access costs as functions of c for various threshold values b = 1,11,20, (c) α,κ as functions of threshold value for various c values, c = 3,5,9,15,45, (d) access costs as functions of threshold value for various c values c = 5,15,45, (e) α,κ as functions of threshold value for $c = 2b+1$, (f) access costs as functions of threshold value for $c = 2b+1$. A probability value of $1-10^{-x}$ for α or κ is plotted as x.

- For a given number of columns (and rows), increasing the threshold value decreases availability and increases confidentiality. Read cost increases linearly with threshold. Write cost remains almost a constant for low threshold levels. As threshold value approaches the number of columns, write cost starts increasing.

- By choosing c value optimally for every b value, increasing b increases access costs and confidentiality and decreases availability.

We can see from the plots in figure 7 that a hybrid scheme provides more flexible design options for the secure store. Given p, the likelihood of a node being compromised, the choice of the degree of replication and secret sharing depends on the security and performance metrics that need to be optimized. Clearly, when access cost or availability is of paramount importance, pure replication ($r = n, c = 1, b = 0$) is the best option. On the other hand, when confidentiality is the critical metric, pure secret sharing ($r = 1, c = n, b = \lfloor (c-1)/2 \rfloor$) is the best option. There is a wide range of confidentiality, availability and access costs where the desired levels are achieved when the server nodes are arranged in a certain number of rows and columns. Thus, both replication and secret sharing are essential when certain bounds are placed on the security and performance metrics. For example, with a confidentiality requirement of $\kappa \geq 1 - 10^{-3.5}$, when access cost should not exceed 22 servers, the optimal choice would be $b = 10, c = 21$.

For a given assumption for p, the security level (confidentiality and integrity) can be increased by sacrificing availability and low access costs. On the other hand, decreasing the threshold level results in improved performance and better availability but exposes the stored data to a higher risk of being compromised. Thus, our analysis demonstrates that our hybrid scheme offers greater flexibility in meeting performance and security goals of a secure store.

Apart from the flexible tradeoff our system offers between security and performance, one additional benefit in our system is tolerance to related attacks. When nodes vulnerable to related attacks are placed in the same column, stealing information is not any easier for an adversary.

3.6 Summary

In this chapter, we outlined the architecture of *secure store* briefly discussing various components of the architecture. We described in detail the data storage scheme along with the associated read/write protocol. We also gave an outline of a simple dissemination protocol. We did a preliminary analysis of the data storage scheme assuming a probabilistic model for server compromises and showed how *secure store* offers a tradeoff between security and performance. We will revisit this tradeoff when we describe our file system implementation in chapter 6, and validate our analytical results with empirical data. In the next chapter, we consider the dissemination component of *secure store* in detail.

CHAPTER 4

THE DISSEMINATION PROBLEM

4.1 Overview

In chapter 3, we introduced the *secure store* architecture and described the dissemination component. Dissemination in *secure store* happens in two parts: (1) dissemination of verification string, and (2) dissemination of shares. While the data shares can be disseminated using any benign-case protocol, the verification string has to be disseminated securely and reliably tolerating malicious servers. We presented in figure 6 a simple protocol for dissemination of verification strings. In this chapter, we consider the problem of secure dissemination of verification strings in detail.

The secure dissemination protocol we develop in this chapter will be generic enough to be applicable in many other contexts. For example, the same protocol could be used to communicate a message or some information introduced at or known to a few nodes to other nodes in a system. We will call the information that is being disseminated an update. An update may be a message that is sent by an authorized person, to be communicated to all the servers in the system, possibly during an emergency situation. An update could also be a new value of a data item that is replicated at the servers for high availability. Servers communicate with each other in rounds of gossip to disseminate updates introduced at a subset of servers. Non-faulty servers should accept only those updates that are introduced by authorized clients and must reject others, in particular, the spurious ones generated by malicious servers. This property is guaranteed as long as the number of compromised servers does not exceed a threshold b.

We first introduce the key allocation scheme used for the dissemination protocol and describe a novel technique called collective endorsement. We use this technique to develop a secure gossip-style dissemination protocol. We evaluate the performance of the protocol through simulations and experiments and compare it with other known secure dissemination

protocols.

4.2 Background

Using public key signatures to protect the disseminated data against data corruption and source spoofing reduces the dissemination problem to one in a benign setting where servers fail by crashing. However, public key signatures are computationally expensive [34, 10], particularly when large volumes of data are to be disseminated. Hence, eliminating usage of digital signatures for dissemination is desirable in a setting where the client set is large or the frequency at which updates are introduced is high.

Gossip protocols for malicious environments without using public key signatures for dissemination have been explored in the past [36, 40, 43, 37]. These protocols do not take advantage of a possibly smaller number of actual faults when compared to the assumed threshold b. The best known class of protocols [43, 37] disseminate an update in $O(log\ n)+b$ rounds, where b is the assumed maximum number of servers that can be compromised, irrespective of the actual number of malicious servers in the system. We present a gossip style protocol for dissemination in malicious environments which takes $O(log\ n)+f$ rounds to disseminate an update, where f is the actual number of malicious servers in the system. In the absence of any malicious activity, our protocol takes only twice as long as the best possible gossip style protocol for benign settings. The buffer requirements and message sizes are higher in our protocol than other protocols and thus it trades off memory and bandwidth resources to improve latency. Since memory and communication resources available to server nodes continue to increase, our approach is viable and will be able to offer lower latency.

Our protocol uses a novel key allocation scheme that allocates a set of symmetric keys to each participating server. A client introduces an update at a randomly chosen set of servers, called the initial quorum. A server accepts an update when an authorized client introduces the update at the server. Each server endorses an accepted update by computing message authentication codes (MACs) for the update using the keys allocated to the server. We call such a list of MACs an endorsement. Endorsements are disseminated to other servers. When a server verifies that $b+1$ other servers have endorsed an update, the server accepts

the update. Our key allocation scheme reduces the total number of keys in the system and hence message lengths and buffer requirements, when compared to a naive node-to-node key sharing scheme.

The key allocation scheme and the collective endorsement technique presented here are quite general and can be used in other applications in Byzantine environments where a set of nodes has to vouch for the correctness of some information. We showed in [28], as an example application, how to use endorsements to render authorization tokens in a distributed system unforgeable. Authorization tokens are used for distributed authorization [61] and in a malicious environment, a token is valid only if at least $b+1$ servers endorse the token.

4.3 Key Allocation Scheme

In this section we present a key allocation scheme that will be used for endorsements in later sections. We do not address the key distribution problem in this thesis. Each server in the system gets a set of symmetric keys from a universal set. These keys will be used by each server to either endorse some information or verify endorsements. An endorsement for some information is a set of MACs computed using that information and a subset of the universal set of keys. If more than one server participate in computing the MACs, we call the endorsement a collective endorsement. We assume the usual cryptographic properties of MACs.

Let the set of servers be indexed with two indices taken from $0\ to\ p-1$, for a prime p greater than both $\sqrt{n}$ where n is the total number of servers in the system and b where b is the maximum number of servers that can be malicious at any time. A server is denoted by $S_{\alpha,\beta}$ with $0\ \leq \alpha, \beta \leq\ p-1$. The universal set of keys consists of p^2+p symmetric keys as follows:

$$U\ =\ \{k_{i,j} | i\ =\ 0\ to\ p-1,\ j\ =\ 0\ to\ p-1\} \cup \{k'_i | i\ =\ 0\ to\ p-1\}.$$

A server $S_{\alpha,\beta}$ is allocated the following set of keys:

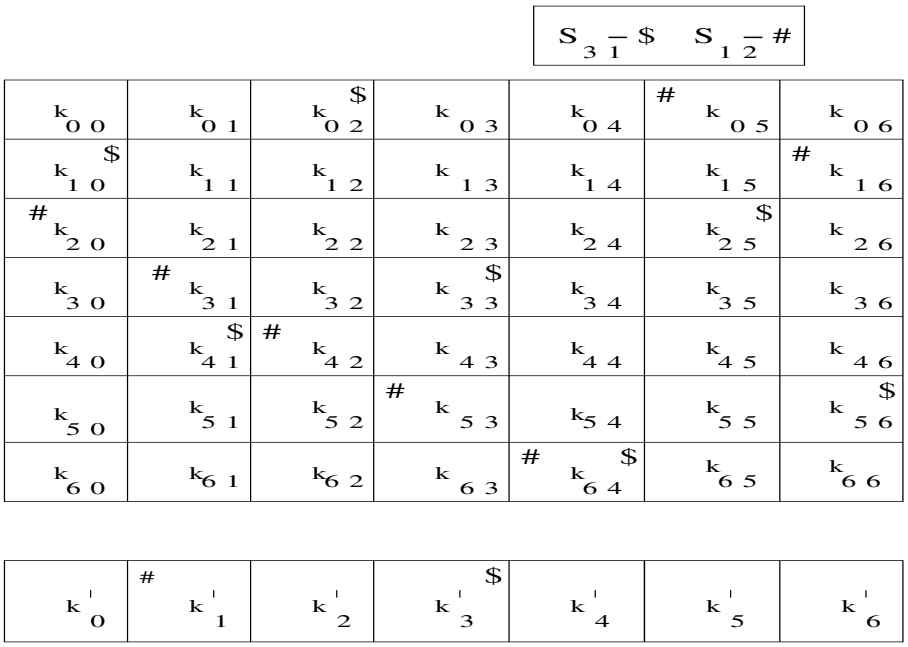


Figure 8: Key allocation for 2 servers : $S_{3,1}$ and $S_{1,2}$ for $p = 7$. Keys allocated to $S_{3,1}$ are marked by \$ and those allocated to $S_{1,2}$ are marked by #.

$$\{k_{i,j} | i = \alpha j + \beta \ mod \ p, j = 0 \ to \ p-1\} \cup \{k'_\alpha\}.$$

That is, server $S_{\alpha,\beta}$ is allocated p keys from the first set of keys along the straight line $i = \alpha j + \beta \ mod \ p$ and the key k'_α from the second set. Figure 8 shows key allocation for two servers, for $p = 7$. We now state two simple properties of this key allocation scheme.

Property 1: Any two servers share exactly one key.

If two servers have the same first index α, they are allocated the same key k'_α from the second set. But they do not share a key in the first set. If the first indices of two servers are different, they share exactly one key in the first set[1] and are allocated different keys from the second set. The following property which is used to validate an endorsement follows directly from property 1.

[1] In a field mod p, if $\alpha_1 \neq \alpha_2$, two straight lines $i = \alpha_1 j + \beta_1$ and $i = \alpha_2 j + \beta_2$ intersect at only one point, where j at the intersection point is given by $(\beta_2 - \beta_1)(\alpha_1 - \alpha_2)^{-1}$.

Property 2: If a server verifies m distinct MACs in an endorsement using the corresponding keys successfully, at least m servers should have participated in computing the MACs unless the verifying server itself generated those MACs.

The following acceptance condition for an endorsement directly follows from property 2.

Acceptance Condition: A server accepts an endorsement as valid if and only if the server verifies at least $b+1$ MACs in the endorsement, none of which was generated by the server itself.

Since only a maximum of b servers can be faulty in the system, any endorsement computed by $b+1$ servers is accepted as valid. We do not address the problem of key distribution since it is not the focus of this paper. Schemes that have been explored by others in other fields like multicast and sensor networks can be used in our system [60, 12]. In the next section, we use this key allocation scheme in our dissemination protocol.

4.4 Problem Statement and Assumptions

We consider the problem of update dissemination in a distributed system where the maximum number of malicious servers at any given time is not more than a threshold b. A client introduces an update at at least $2b+1$ servers. Once introduced, the update is disseminated to other servers in the system securely. An update is said to be valid only if an authorized client introduces it. In particular, spurious updates introduced by malicious servers should not be accepted by non-faulty servers. Hence, a non-faulty server accepts an update as valid if the update is either introduced by an authorized client or accepted by at least $b+1$ other servers. We assume that every server can communicate with every other server securely. In particular, our protocol uses a pull strategy and communication channels are assumed to be secure against impersonation and replay attacks. We assume a synchronous system since our protocol works in rounds of gossip.

Let the set of servers be $S_{i,j},\ i = 0\ to\ p-1,\ j = 0\ to\ p-1$ for a prime $p > 2b+1$ [2].

[2]Number of servers can be less than p^2 but each server receives two indices i, j between 0 and $p-1$, chosen randomly and without repetition.We only require that each server share at least $2b+1$ keys with other servers.

Each server is allocated $p + 1$ keys as described in the previous section.

4.5 Gossip Protocol

Our dissemination algorithm works as described in figure 9. Initially a client introduces an update at q randomly chosen servers, for an q greater than $2b + 1$. Choice of q will be discussed in later sections. We call this set of q servers the initial quorum. Each of the q servers independently authenticates and authorizes the client request before accepting the update. Updates are timestamped to prevent replays. Each of the non-faulty servers that accepts the update directly from a client generates MACs for the update using all the $p+1$ keys it has and stores the MACs in its buffer. The update itself is disseminated to other servers using a protocol meant for benign environments [13, 49]. Dissemination of MACs is done in rounds of gossip. Each server keeps a counter of successfully verified MACs for each update. Every round, each server selects a partner randomly and requests MACs. Upon a request, a server sends all MACs in its buffer, generated by it or received from other servers, to the requesting server. The requesting server verifies the correctness of all the MACs for which it has the key. It discards the invalid ones and updates the counter for the newly verified MACs. If it has successfully verified at least $b + 1$ MACs, the server accepts the update as valid.

Once accepted, the server generates the rest of the MACs for the update using the keys it has. The server stores all the verified or generated MACs and other received MACs (for which the server does not have the key to verify) in a buffer to disseminate to other servers in future rounds. If a received MAC for a key that the server does not have is different from the already stored one (received at an earlier round), it replaces the stored MAC with the just received one (see 4.5.2). All MACs are sent and stored accompanied by identifiers of the keys used to generate them. Figure 10 shows the progress of a typical run of the protocol in a system of 840 servers with a b of 10, for an update introduced at 12 non-malicious servers. The plot shows the number of servers that have accepted the update at the end of each round.

The correctness of the gossip protocol can be demonstrated by showing that the following

At server $S_{\alpha,\beta}$:

1. If an update is introduced by an authorized client, accept the update and generate MACs using the allocated keys. Store the generated MACs in buffer to be disseminated to other servers.

2. Each round :
   2.1. Choose a random partner.
   2.2. Ask for updates and collect MACs.
   2.3. For each received MAC
      If the key to verify is present,
         Verify the MAC using the allocated key, store in buffer if successfully verified, reject otherwise.
      Else,
         Accept the incoming MAC and store in buffer, possibly replacing an already stored MAC for the same key and update with the received MAC(see 4.5.2).

3. If some server asks for updates, forward all the stored MACs.

4. Accept an update as valid if the update is verified by $b+1$ MACs using distinct keys. If accepted, generate the rest of the MACs for the update and store in buffer.

Figure 9: Gossip Protocol

properties hold.

Safety: No spurious update introduced by a group of malicious servers will be accepted by any non-faulty server as long as not more than b servers are malicious.

Since no non-faulty server will generate any MAC for an update before accepting it, not more than b servers will generate MACs for a spurious update. This implies, from property 2 of the key allocation scheme, that no server will be able to verify $b+1$ MACs for a spurious update. Thus, no non-faulty server will accept a spurious update.

Liveness: If an update is introduced by a client at a sufficient number of servers, the update will eventually be accepted by all servers.

A server has to verify at least $b+1$ MACs to accept an update. For a sufficiently

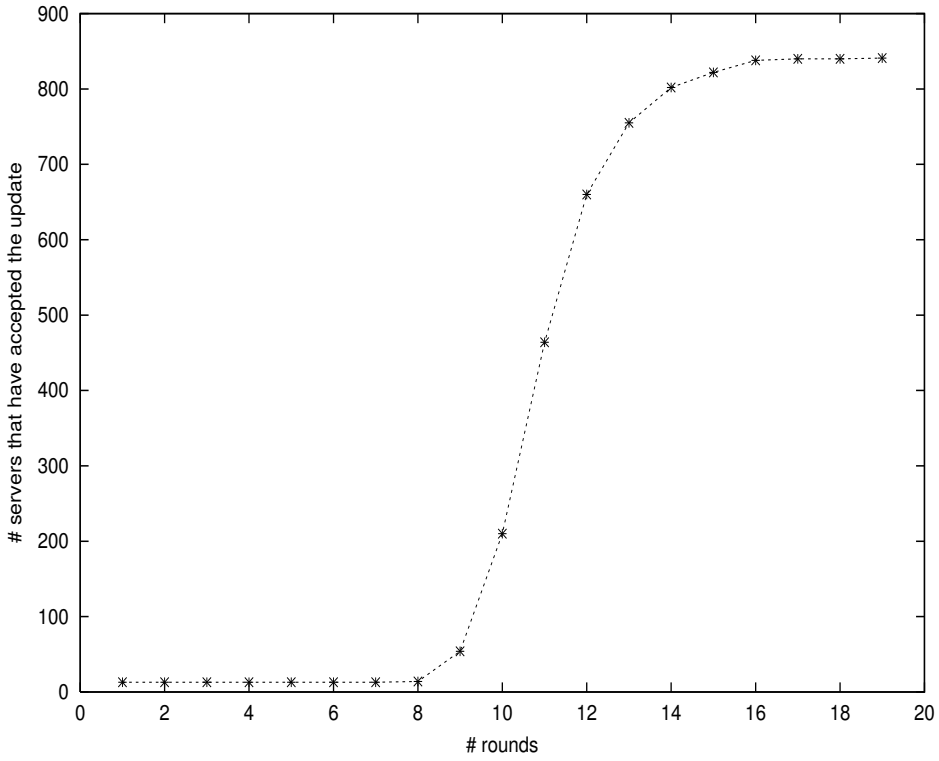


Figure 10: Number of servers that have accepted the update as a function of the round number in a typical run for n=840,b=10 for an update injected at 12 non-malicious servers.

large initial quorum, there will be a number of servers that would accept the update as valid from MACs generated by the servers in the initial quorum. The size of the initial quorum is $2b + 1 + k$ for some small value of k (see section 4.5.1). All those servers whose key allocation lines intersect those of the servers in the initial quorum at at least $2b + 1$ distinct points will accept the update. Once a fraction of servers accept the update, these servers generate MACs in turn, which are disseminated to other servers. This would lead to remaining servers accepting the update. Malicious servers cannot stop valid MACs from reaching non-faulty servers. They can only delay the flow of MACs reaching non-faulty servers. The pull strategy we use further limits the power of malicious servers to stop the flow of valid MACs. Every generated MAC will eventually reach every server. Hence all non-faulty servers would accept the update eventually.

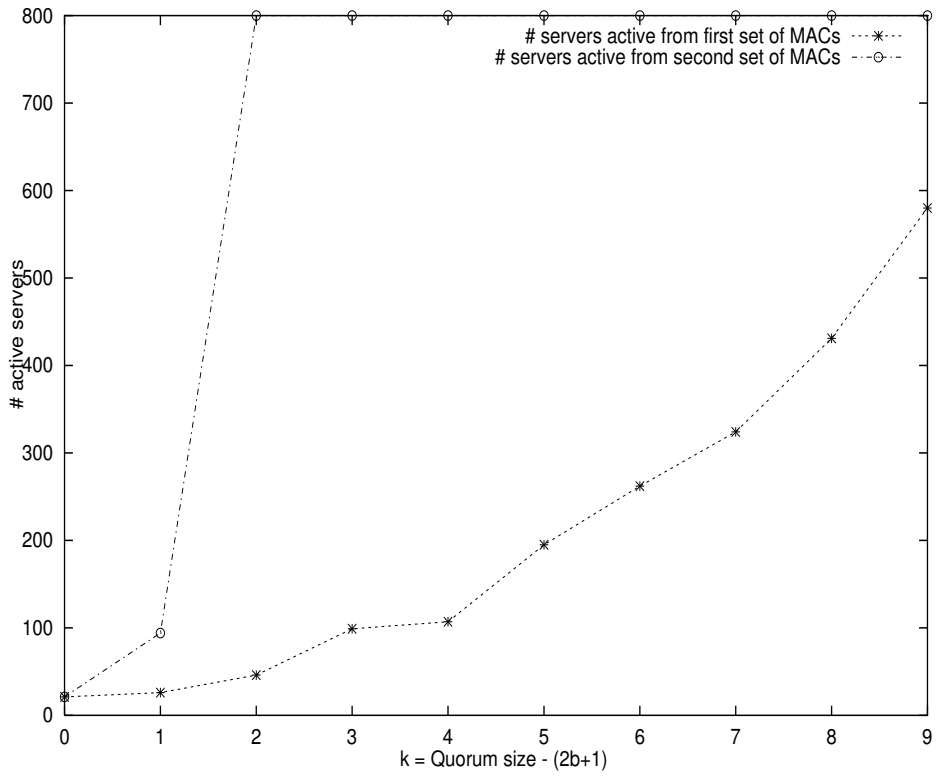


Figure 11: Number of servers that accept the update from first and second set of MACs for different sizes of initial quorum, k - difference between quorum size and optimal quorum size, 2b+1, for $n = 800$ servers and $b = 10$.

4.5.1 Choice of Initial Quorum

When introducing an update, a client chooses an initial quorum of servers to introduce the update into the system. The size of this quorum is determined by the requirement that a sufficient number of servers outside the initial quorum should each share at least $2b+1$ keys with the initial quorum. After MACs generated by the initial quorum are disseminated to all the servers in the first phase, a fraction of servers accept the update and generate more MACs. Other servers will accept the update after receiving the MACs generated in the second phase. In appendix A.1, we show that if $p \geq q \geq 4b+3$, all servers will accept the update in two phases for any choice of initial quorum of size q. This is only a theoretical upper bound and in practice we have found that we require a much smaller initial quorum. If the servers in the initial quorum have keys allocated along parallel lines from the first set, then the size of the initial quorum can be $2b+1$. If the initial quorum is chosen randomly,

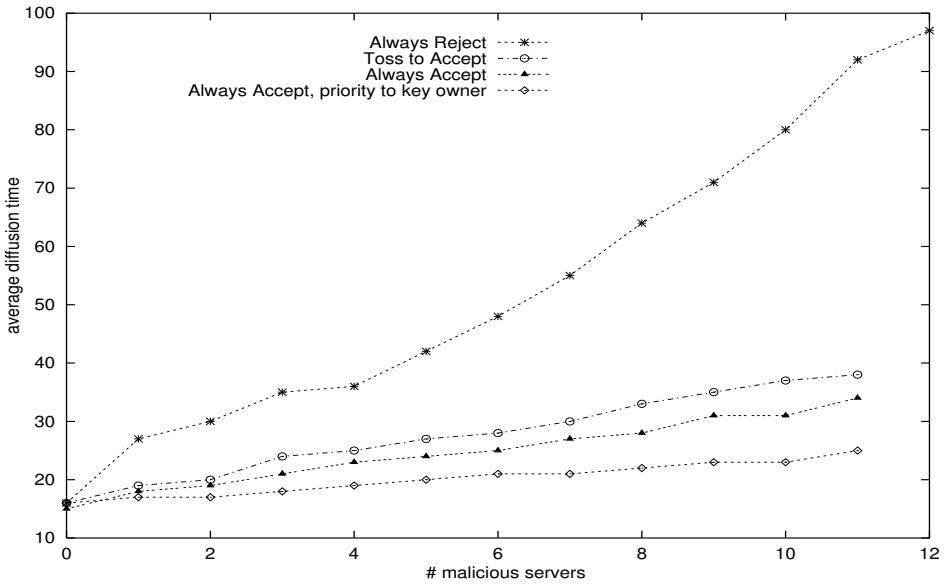


Figure 12: Average diffusion time against actual number of faults for $b = 11$ and $n = 1000$ servers, for various policies on resolving conflicts between MACs.

our simulations show that the size of the initial quorum is $2b + 1 + k$ for a small k. Figure 11 shows the average number of servers that directly accept the update in the first phase from the MACs generated by the initial quorum of servers as a function of k for randomly chosen initial quorum and the total number of servers that will accept the update at the end of second phase. As can be seen from the plots, for a system of about thousand servers with a maximum of ten being malicious, a small k equal to two or three serves our purpose.

4.5.2 Dealing with Conflicting MACs

In the gossip protocol shown in figure 9, a server that receives a MAC for an update for a key that it does not have cannot verify the correctness of the MAC. However, the server stores the MAC to be forwarded to other servers. A malicious server may generate invalid MACs for a valid update to fill other servers' buffers and hence delay the acceptance of an update. A receiving server, when it receives a MAC that is different from the already stored MAC for the same update and the same key, has to decide which MAC to keep. There are three different possible strategies for handling this situation. One strategy is for the receiving server to always reject the incoming MAC. That is, the first received MAC stays in the server buffer and all others are rejected. Second strategy is to accept the incoming

MAC with a certain probability. Third strategy is to always accept the incoming MAC, replacing any previously accepted MAC. Figure 12 compares the performance of each of these strategies. Our simulations show that the third strategy is the most effective while the first is the least effective. This is because the always-accept strategy gives all generated MACs a chance to reach every server quickly.

The protocol could be further optimized by requiring the receiving server to give preference to MACs received from servers that have the keys for those MACs over those from servers that do not have the keys for the MACs. The last plot in figure 12 shows diffusion times for this policy. To implement this policy, each server has to know what keys are allocated to other servers.

4.5.3 Key Consensus

We do not consider the problem of how each server receives the keys allocated to it in this thesis. The problem of key distribution is orthogonal to the problem we address in this chapter and any secure key distribution scheme developed for other applications [60, 12] could be used. Alternatively, if a threshold state machine service is available for other purposes, as in the case of our file system described in chapter 6, such a service could distribute the keys to the participating servers.

In this section we discuss an issue that may be of concern in key distribution, namely consensus on shared keys. Each key in our key allocation scheme is shared by p servers. Some of these servers may be malicious. Hence, some servers that share a key may not have identical copies of the key unless a Byzantine fault tolerant consensus protocol is used for key distribution. In the file system we build, our metadata service is a single central threshold service that can be used to distribute the keys to the data servers. Even if this service is not available, we point out that a strict consensus on all keys is not necessary. Any distribution algorithm that distributes the keys correctly when no participating server is malicious would work for our purpose. As long as each server shares $2b+1$ keys with other servers, there will be at least $b+1$ good keys that will be useful in the dissemination process. Hence, as long as keys that are not allocated to any malicious server are correctly shared,

Metric	Tree Random (see [40]	Short-Path (see [37])	Youngest-Path Path Verification (see [43])	Collective Endorsement
Diff. Time	$\Omega(b.log(n/b))$	$O(log\ n + b)$	$O(log\ n) + b + c$	$O(log\ n) + f$
Mesg. Size	$O(1)$	$\psi(n, b)$	$30(b+1).\ O(log\ n)$	$d.\ O(p^2)$
Storage	$O(b)$	$\psi(n, b)$	$30(b+1).\ O(log\ n)$	$d.\ O(p^2)$
Comp. Time	$O(log\ b)$	$\Omega((\frac{\psi(n,b)}{log(n/b)})^{b+1})$	$O(b^{b+1} + b.log\ n)$	$O(p/(log\ n))$

Table 1: Performance comparison of different gossip protocols. n - number of servers, d - size of a MAC, c - a small constant, $\psi(\mathbf{n}, \mathbf{b}) = ((\mathbf{n/b + 2}))^{\mathbf{O(log(b+2+log\ n))}}$. All measures are per host per round except diffusion time which is measured in number of rounds.

our dissemination algorithm works correctly. As an example, a simple key distribution scheme could be used where, for each key a designated key leader distributes keys to other servers. All our simulations and experiments were run by making invalid all keys that are allocated to at least one malicious server.

4.6 Performance

In this section, we analyze the performance of our protocol and compare it with known protocols for dissemination in Byzantine environments. We consider four performance metrics : (1) diffusion time, (2) average message length per host per round, (3) average buffer size required per host per round and (4) average computation time at a server every round. The other performance metric, host load which is defined as the average number of messages received per round is one, which is same for all the protocols discussed here.

We ran simulations and experiments with an implementation of our protocol to validate the claims we make here about the performance of our protocol. We implemented both our protocol and a version of path verification protocol suggested by Minsky and Schneider [43] for a cluster of thirty machines running Linux on 300MHz Pentium processors. For both implementations, round length was set to fifteen seconds. A typical experiment involved starting a randomly chosen set of servers in malicious mode and the rest in normal mode and injecting updates at a randomly chosen set of $b + 2$ non-malicious servers at a chosen frequency. Most effective malicious behavior for our protocol is simply sending random bits for MACs to other servers upon every request. This is easy to see since if a malicious server

sends a correct MAC for an update upon a request, it will only possibly reduce the diffusion time of the protocol run. For the path verification protocol, we made malicious servers simply fail benignly, replying with empty list of proposals for requests from other servers. For both protocols, updates were discarded twenty five rounds after they were injected, which was well over the diffusion time for most of the updates. For the path-verification protocol, the diffusion strategy chosen was promiscuous youngest diffusion [43] with an age-limit of 10 rounds for a proposal and the sampling strategy chosen was bundle sampling with a maximum bundle size of 12. A value of 11 was chosen for p for our protocol. The following characteristics were studied: (1) diffusion time as a function of the threshold b (2) diffusion time as a function of the actual number of malicious servers f (3) average message size, buffer size and computation time at each server as a function of the frequency at which updates were injected. Last three metrics were measured when the system achieved a steady state and updates were being dropped at the same rate at which fresh updates were being injected.

Table 1 compares the performance of our protocol and other known protocols for Byzantine dissemination. It should be noted here that other protocols are information-theoretically secure while ours is only computationally secure since we rely on cryptographic properties of MACs. As can be seen from the table, our protocol trades off bandwidth and memory resources to improve latency. In subsequent sections, we discuss each of these metrics in some detail, presenting our simulation and experimental results. Although protocols presented in [36] and [40] require considerably less bandwidth and memory, their latency is high ($\Omega(b.log(n/b))$) compared to other protocols. The protocol described in [43] offers the best latency among other protocols. Since our primary focus is on reducing latency, we will compare our protocol to [43] in the following.

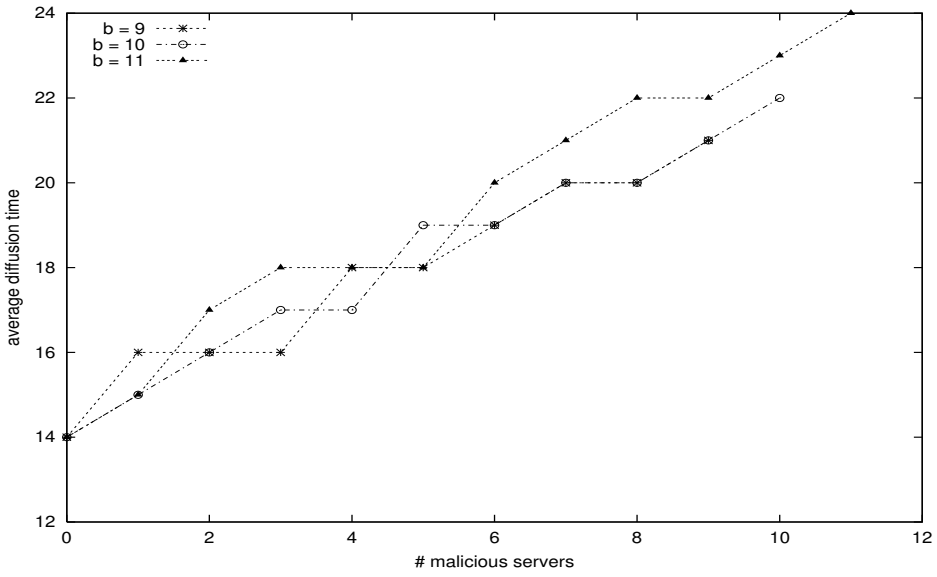
4.6.1 Diffusion Time.

When no server acts maliciously, our protocol takes not more than twice the diffusion time of the best protocol for benign environments to diffuse an update. In this case, every generated MAC takes only $O(log\ n)$ rounds (time taken by best-possible benign-case protocol) to reach

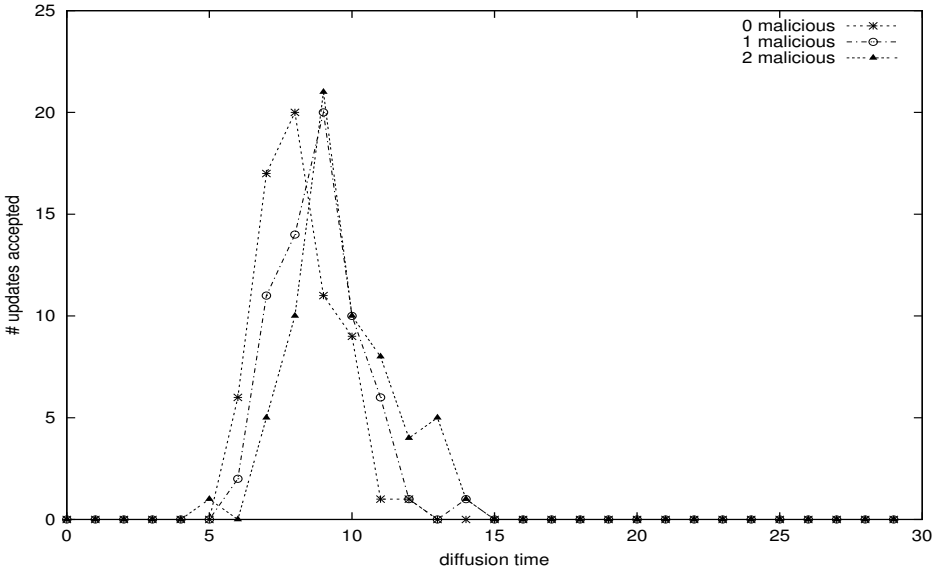
every server. A fraction of servers accept the update and generate MACs using other keys in turn. The newly generated MACs take another $O(log\ n)$ rounds to reach every server. If the size of the initial quorum is appropriately chosen, every server will verify at least $b+1$ MACs either generated by the initial set of servers or generated by servers that accept the update based on MACs generated from initial set of servers. We obtain an upper bound for the size of initial quorum in appendix A.1. Both in our simulation and experimental results, we observed quite often the diffusion time to be less than twice best possible time for a benign setting since (1) some servers accept the update even before the first $log\ n$ rounds and (2) MACs generated in second phase need not reach a lot of servers.

When f servers are acting maliciously, diffusion time of our protocol is $O(log\ n)+f$, independent of the assumed threshold b. We justify this claim by showing in appendix A.2 that in the presence of f malicious servers, each MAC takes $O(log\ n)+f$ rounds to reach a constant fraction of servers. After $O(log\ n)$ rounds, for a particular key, a constant fraction of the servers hold some MAC, whether valid or spurious. Fraction of these servers that hold a valid MAC can be shown to be $1/(f+1)$. Thus after the first $log\ n$ rounds, the probability that a server would obtain a valid MAC is close to $1/(f+1)$. Among the servers that have the key to verify a MAC, the fraction of servers that have not seen a valid MAC decreases by a factor of $f/(f+1)$ every round on the average. It takes about $O(f)$ rounds for this fraction to reduce to a small constant.

Our proof in appendix A.2 holds only when malicious servers are not allowed to disseminate spurious MACs for a key before some non-faulty server starts disseminating the corresponding valid MAC. Since this is true for the first set of MACs generated by the initial quorum, these MACs take $O(log\ n)+f$ rounds to reach a constant fraction of servers. Some of the servers accept an update at the end of this phase. Remaining servers would need only a few more MACs to accept the update and these are obtained in a very short period of time, after having been generated by the servers that accept the update in the first phase. Second set of MACs may take a longer time to reach all the servers. However, this does not affect the latency because at the end of first phase, most of the remaining servers need only a few more MACs and the MACs generated in second phase do not have



(a)

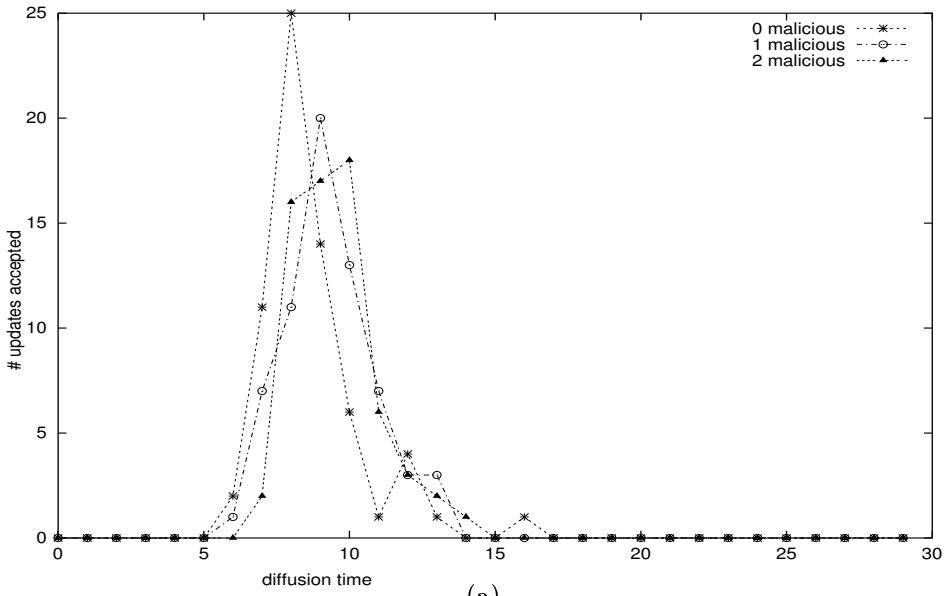


(b)

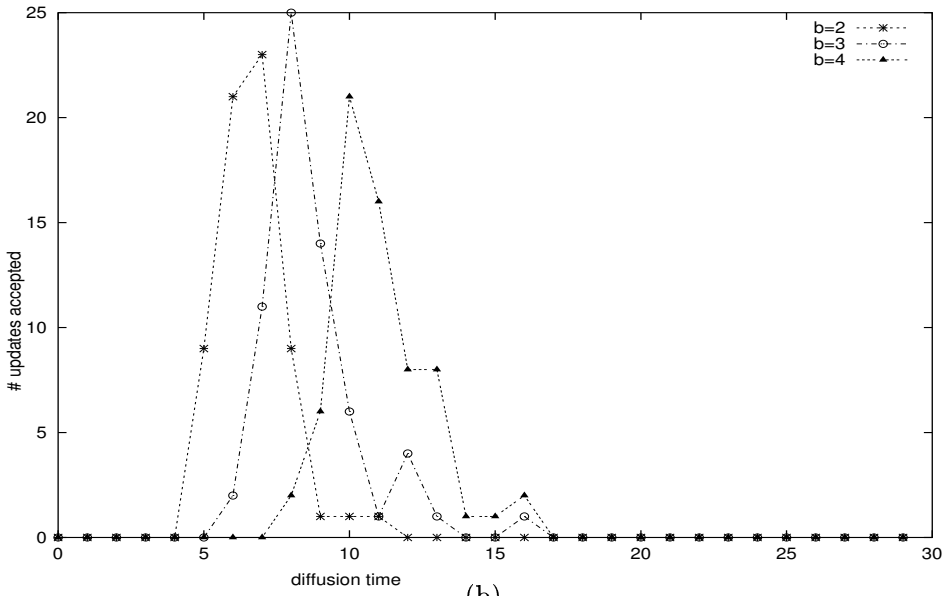
Figure 13: (a) Average diffusion time in number of rounds as a function of f for different values of b for collective endorsement protocol for $n = 1000$ servers, results from simulation, (b) Distribution of diffusion times of updates as a function of f for fixed $b = 3$ for $n = 30$ servers for collective endorsement protocol, experimental result.

to reach a lot of servers.

Our claim is justified by the results we obtained from our simulations and experiments. Our simulations show that as we increase the number of malicious servers by one every time, starting from no-malicious-servers case, diffusion time is increased only by one round every



(a)



(b)

Figure 14: Distribution of diffusion times of updates as a function of f for fixed $b = 3$ and as a function of b for $f = 0$, $n = 30$ servers, for path verification protocol, experimental results

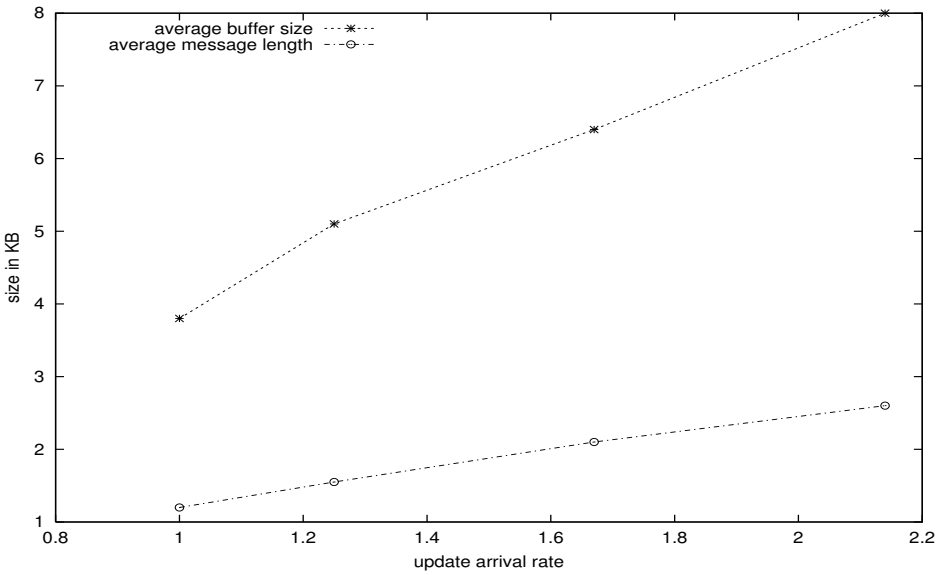
time. Figure 13(a) shows the average diffusion time as a function of f for various values of b as generated by our simulations. This is validated by the diffusion times we observed in our experiments with our implementation. Distribution of diffusion time across updates for various values of f for a b of 3, as seen in our experiments is shown in figure 13(b). The

corresponding plot for our implementation of path verification protocol as measured in our experiments is shown in figure 14. It can be seen from the figures that average diffusion time for path-verification protocol depends on b and f while that of our protocol depends only on f.

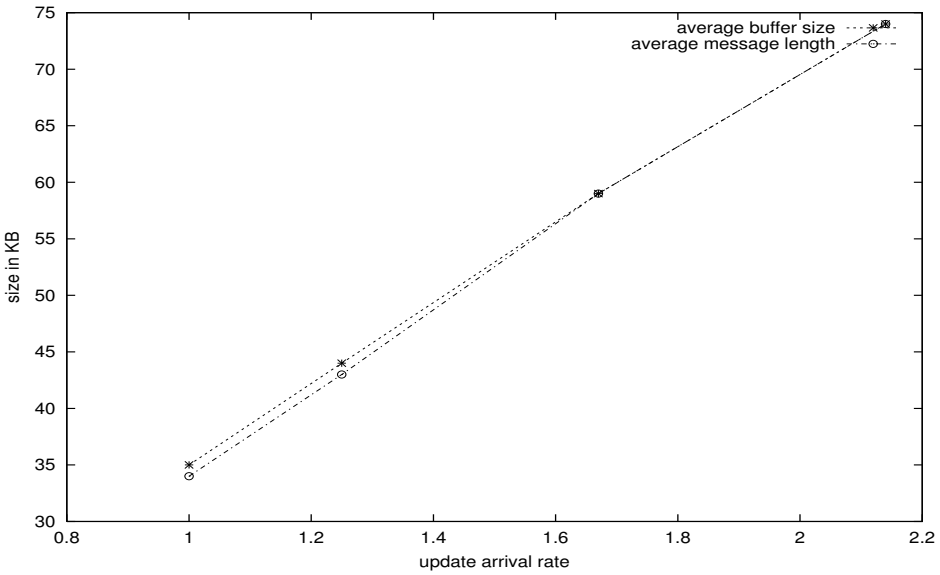
4.6.2 Message Size, Buffer Size and Computation Time.

Average message size and storage requirements per round per host for our protocol is the number of keys times the size of a MAC. Number of keys is $p^2 + p$ where p is greater than $2b + 1$ and $\sqrt{n}$. This is expensive when compared to that of the path-verification protocol, which is $O(b.log\ n)$. However we believe our buffer requirements and message size are scalable to a moderate number of servers. In our implementation, we chose 128bit MACs. Figure 15 shows average buffer size and message size measured in our experiments for our protocol and path verification protocol, as a function of the frequency of arrival of updates. The figures show that our resource requirements are only an order of magnitude higher than those of path-verification protocol for the chosen number of servers(30). However, our protocol would need only the same amount of message size and buffer size even when n is increased to 100 as long as b is less than 6. A higher value of b can be chosen for the same $p(11)$ if consensus on shared keys is guaranteed in the presence of malicious faults.

We observed slightly smaller average computation time for implementation of our protocol over our implementation of path verification protocol. However, for higher values of b, we expect our protocol to take much shorter time compared to path verification protocols. Our protocol requires only about p MAC operations at each server for an update in the whole of an update's dissemination time. Path verification protocols require $O(b^{b+1})$ [43] computation time at each server every round. This is because path-verification protocols involve checking for $b + 1$ non-intersecting paths from a set of paths, which is known to be NP-complete.



(a)



(b)

Figure 15: Message size and buffer size in KB as functions of update arrival rate in numbers per second for (a) path verification and (b) collective endorsement protocols for $b = 3$ and $n = 30$ servers, experimental results

4.7 Summary

In this chapter, we considered the dissemination component of *secure store* in detail. In particular, we posed secure dissemination of verification string as a generic update dissemination problem and presented a gossip-style protocol that securely disseminated an update

in $O(log\ n) + f$ rounds, where f is the actual number of malicious servers. Thus our protocol pays the price of delayed dissemination only in proportion to the amount of malicious activity. When there are no malicious servers, the dissemination latency is comparable to that of the best-possible benign case protocol. The protocol used a special key allocation scheme and a novel technique called collective endorsement, both of which can be used in other secure distributed applications.

We evaluated our protocol through experiments and simulations and compared its performance with other known protocols for secure dissemination. We also present in appendix A proofs for some of the claims we made in this section.

CHAPTER 5

DATA CONSISTENCY

5.1 Overview

In the previous chapters, we introduced the secure store architecture and its dissemination component. Secure store stores data in the form of secret shares and replicates the shares for better availability and performance. Thus, data is both distributed and replicated. The ordering guarantee the store makes for data access requests is an issue to be addressed in such a system. In this chapter, we address this issue. Secure store provides weaker consistency guarantees: monotonic reads and causality. We first revisit our target applications and classify them in terms of their consistency requirements. We then introduce the weaker consistency models that suit our target applications. We then present a protocol that guarantees weaker consistency levels in a purely replicated system, for the sake of clarity. We finally modify the secure store protocols for consistency guarantee. We conclude the chapter with a discussion of some performance benefits that result from using weaker consistency models.

5.2 Target Applications

One of the primary requirements of secure store is to address the needs of a variety of target applications. Our target applications, apart from their security and performance requirements, also vary in terms of their consistency requirements. We identify in this section, three classes of applications that have both security and consistency requirements and can use *secure store* effectively to meet their storage requirements. Most of our target applications fall under one of these three classes and hence we will use these classes to define our consistency models in subsequent sections..

We consider three classes of applications that have security as well as consistency requirements for stored data. The traditional security requirements include confidentiality,

integrity and availability. A variety of weak consistency models have been explored in the past that range from guarantees based on causality to ones that only require that reads of data items return values that are at least as recent as those read in the past.

1. First, we consider applications in which a single user accesses non-shared information. For example, a resident of the Aware Home may save medical records of family members or store tax information in the secure store. Such information is only accessed by the resident. Under certain conditions, copies may be made available to others (e.g., tax preparer or a medical facility) but the information is of non-shared nature.

2. In the second case, we consider applications that involve multiple users. For example, a school may create and store information which may be accessed by many families in the community. In this case, the information is written by a single user but it could be read by many others. This application falls in the general class of applications where a single source disseminates information that is of interest to many others.

3. Finally, we consider applications that manipulate data read and written by multiple users. Many collaborative applications that enable asynchronous interactions across users fall in this category. For example, a group of citizens may collectively develop a plan to address problems in the community over a period of time.

In the non-shared case, which is exemplified by the first class, the private nature of the data clearly implies that it is of confidential nature. The consistency requirement is straightforward because the updates are ordered and we only need to identify the most recent update. High availability for the data such as medical records is a must in an emergency situation. In the second class of applications where data is written by one user and read by others, integrity requirements can be easily seen. In particular, readers must be assured that the data they receive is from the legitimate writer and has not been modified by unauthorized parties. Strong consistency is not necessary. In particular, even if a reader may not see the latest value in the current read operation, it is sufficient that future reads return successively more recent data. Application level information can be used to discern when information may be stale.

For the last application class, where shared data items may be updated and read by multiple users, both confidentiality and integrity could be important and access to the information must be controlled. Consistency based on causality may be sufficient when concurrent updates to shared data items are not common or when such updates can be merged. Strong consistency, enforced by some type of synchronization may be unnecessary because users may externally coordinate their updates.

Although we have given example applications that come from the home and community domain, the secure store will be valuable in addressing the needs of applications from other domains which share similar security and consistency needs. Examples of such applications include access to personal information such as email, inventory management across distributed locations, collaborative software development projects, and educational and entertainment applications.

5.3 Consistency Models

We advocated that the secure store support access to a variety of information with different levels of consistency for shared information. We consider the following consistency models that can meet the needs of many of the applications described earlier.

1. **Monotonic Read Consistency (MRC):** A client that accesses data items using this level of consistency always sees an increasing set of writes to a data item as time proceeds. More specifically, if a client reads a value v for a data item x_i, at a later time when it reads data item x_i again, it is returned v or a value which is newer than v. A value v' is said to be newer than v if the store orders the write that produced v' after the write which stored v. For non-shared data that is accessed by a single client, MRC implies that the client will access the most recent copies of its data items. For shared data, future reads of a reader could return more recent values but are not guaranteed to return the latest value of the object. Thus, consistency guarantees provided by MRC are similar to the monotonic-reads and read-your-writes session guarantees in Bayou [56].

2. **Causal Consistency (CC):** MRC only ensures that for a given data item, a client never receives values older than the ones read in the past. It does not impose restrictions across related data items. Consider the case when a client writes value v_2 to a data item x_2 based on value v_1 of data item x_1 that it has read. Informally, if another client reads value v_2 for x_2, CC ensures that the client is guaranteed not to read a value for x_1 that is older than v_1. The notion of "older" is precisely defined based on the happens before order for read and write operations to shared data items [6]. This relation can order read and write operations across a set of related data items. In particular, if o_1 and o_2 are two writes to data item x which assign values v_1 and v_2 to x respectively, v_1 is said to be causally overwritten by v_2, if o_1 causally precedes o_2. A secure server that supports CC ensures that no read operation returns a causally overwritten value.

The MRC model meets the needs of the first two classes of applications, those that manipulate data belonging to a single user or those in which there is a single writer and multiple readers. The CC consistency model addresses the needs of the third class of applications, where multiple writers and readers interact asynchronously. .

Although we will focus on protocols for MRC and CC, clearly these protocols may not be able to meet the consistency needs of all applications. In particular, some applications may require strong consistency. In this case, existing protocols can be used. For example, the replicated state machine approach based protocol in [35] can be used to ensure that all client operations appear to execute in a total order. MRC and CC do not address how quickly values written by a certain client become available to others. Neither do our definitions for MRC and CC capture the liveness property. Our protocols depend on the dissemination protocol and synchrony assumptions to guarantee liveness property. Although models that address timeliness do exist, their implementation in an asynchronous distributed system is infeasible. We do assume that MRC and CC will return newer values eventually when clients continue to read the data.

5.4 Protocols for Purely Replicated System

For the sake of clarity, we first discuss consistency in the context of a simple system, where data is purely replicated across multiple servers and not secret-shared. We will not address confidentiality in this system and integrity will be guaranteed by using digital signatures. After the technique is illustrated using a purely replicated system, we will modify *secure store* data access protocols to guarantee weaker levels of consistency described earlier.

In this simple system, each data item is written to $b+1$ servers along with an unforgeable digital signature by the writing client. Data items are securely disseminated in the background to other servers. A client that wants to read a data item looks for $b+1$ identical copies, each from a different server, before accepting a value as valid.

5.4.1 Context and Its Management

In traditional storage systems, it has been the responsibility of the store to ensure consistency between reads and writes when clients access data. We take the approach where a considerable portion of the responsibility to ensure consistency in data access is shifted to the clients who access the data. Each client maintains some metadata about its past interaction with the store to help make decisions about consistent data access. We call this metadata the *context* object. Each data item that is stored in the store also has an associated *context* object. We first define the *context* object and give protocols for the clients to store and retrieve *context* that is persistent across multiple sessions.

In replicated data systems, since all copies may not be identical (an update writes new values only at a subset of the servers), version numbers or timestamps are used to determine which copies have the latest values. Such timestamps must monotonically increase as updates are done. They can be read either from clocks that advance between successive updates or can be read from logical clocks that are advanced as read and write operations are executed. When consistency needs to be ensured across a set of related data items and a client accesses several such items, timestamps have to be kept for each of the data items. The *context* of a client includes the unique identifiers of data items accessed in a given session and the timestamps associated with the data items. Although *secure store* may

potentially store a large number of data items, in a given session, we assume that a client only accesses a small number of such items. This implies that the *context* maintained by a client at a given time will not be large.

Consistency dependencies in the CC model can arise because of operations that are executed on several data items that belong to a related group. For example, a client C_i may write value v to data item x based on the value v' of another data item x' that it read in the past. Another client that reads v for x and then requests a copy of x' must see either v' or a more recent value of x'. To ensure this, we need to associate meta-data with values of data items that are stored at the servers. Such meta-data for value v must reflect the fact that it potentially has a causal dependency on the write that produced value v' of item x'. In particular, the meta-data stored with v not only has the timestamp of x but also the timestamps of other related data items to capture causally preceding updates to them. In fact, the set of timestamp values of the data items that are related to x and x' is precisely the information that is needed to maintain CC. This information is captured by a client's *context*. Thus, if $X = \{x_1, x_2, ..x_m\}$ is a related group of data items, the *context* associated with data items in X at client C_i is $\mathcal{X}_i = ((uid(x_1), ts_1), ..., (uid(x_m), ts_m))$. As discussed later, the *context* evolves as ts_j's increase when reads and writes requested by C_i complete. If an x_i is written back to the store, the client's *context* associated with X is written with the data value. Timestamp vectors similar to *context* defined by us are also used in many systems that have been developed for weakly consistent replicated data [56].

A client's *context* reflects the accesses it has completed. Thus, when a new session is initiated by the client, it must initialize its *context* to reflect the interactions it had completed in the last session. A client may deal with a number of *context* objects over a period of time. Clients may be resource poor and information stored at the client site could be compromised easily. Hence, we choose the approach in which a client saves its *context* in the secure store along with a signed digest of the *context*. The signature ensures that a malicious server cannot alter the *context* information.

To ensure that a read of the *context* returns its latest value, strong consistency needs to be guaranteed for the read and write operations that access *context*. To ensure such

consistency and availability in the presence of up to b faulty servers out of a total of n, we use a quorum based approach when *context* is read or written. In particular, we require that the reading or writing of *context* information be done with at least $\lceil (n+b+1)/2 \rceil$ servers. This ensures that at least one non-faulty server, to which the last *context* information was written, will participate in *context* read. The client can choose the most recent *context* value that has a valid signature from the values returned by the servers in the quorum.

Read C_i's *context* on session initiation:

let X be the related group of data items
that C_i wants to access in the session;
request C_i's *context* associated with X and signature
from all servers;
wait for at least $\lceil (n+b+1)/2 \rceil$ responses;
check if a *context* is valid by verifying its signature;
$\mathcal{X}_i$ = latest valid *context* returned by some server;

Store C_i's *context* on completion of its session:

let $\mathcal{X}_i$ be C_i's current *context* for data items in X ;
send $\{\mathcal{X}_i, \{\mathcal{X}_i\}_{K_i^{-1}}\}$ to $\lceil (n+b+1)/2 \rceil$ servers;

Figure 16: Context Acquisition and Storage

As can be seen in Figure 16, *context* read or write can be completed as long as $\lceil (n+b+1)/2 \rceil$ servers participate in the quorum. Since a valid signature is required, faulty servers can only misbehave by either not responding or sending an old value of the *context*. Given that the latest value received from a server is chosen as the client's *context*, the *context* from a non-faulty server that participated in the most recent write will be chosen. Here, latest value is that vector which has the highest timestamp for every data item in the group. Notice that we require only $b+1$ servers in the intersection of two quorums whereas masking quorums require $2b+1$ servers in the intersection. Our optimization is possible because we can choose the latest valid *context* from a single server while masking quorums require that a value appear in the response of at least $b+1$ servers for it to be chosen.

If a client successfully writes its *context* prior to session termination, at least one of the servers that responds to the next *context* acquisition request will return the client's

latest *context*. If the client fails either before this is done or while it is writing the *context*, the quorum intersection is not guaranteed to return the most recent *context*. In fact, the context stored in the meta-data of data items could be more recent than the *context* written by the last successfully terminated session. In this case, a more expensive protocol is used to reconstruct the context. The client will have to read the timestamps associated with all data items in a group X for which *context* needs to be reconstructed. These items must be read from all servers. Only the faulty servers may choose not to respond to the client request. From these values, the latest valid timestamp for each data item is used to reconstruct the client's *context* for data items in X.

5.4.2 Protocols for Non-shared and Single-writer applications

Once a session is started and a client initializes its *context*, it could issue read and write operations for the data items that it is authorized to access. The client is responsible for accessing consistent data based on the *context*. Similar to the protocol in Figure 16, we assume that the operations are executed by client C_i. Since this section considers data that is written by a single client, C_i is the only one that executes write operation on the data items. Other clients can read shared data that is written by C_i. We assume that for a given set of data items, either MRC or CC consistency is specified at the time of their creation. Thus, the same data item cannot be accessed with MRC consistency requirement at one time and CC consistency at another time.

Figure 17 shows the protocols for reading and writing when the data is written by a single client. For monotonic read consistency, only the current version number or timestamp of the data item is stored with its value. Since the timestamp of this data item monotonically increases as values are read and written, successive reads of a client will return newer values. Since a client writes its *context* at the end of a session, a future session will also return the most recent value seen by the client or a newer value.

Since servers simply act as passive stores for signed information, a faulty server cannot modify either the meta-data or the value of the data item in an undetectable way. Thus, it can either not respond to a request, or respond with old data or data that is corrupted.

let $\mathcal{X}_i = ((uid(x_1), t_1), ..., (uid(x_m), t_m))$;

Write(x_j, v)**:**

increment t_j in $\mathcal{X}_i$ to current clock value;
if CC is required then
    write-message := {"write",$uid(x_j)$,
    $\mathcal{X}_i$, v, $\{uid(x_j), \mathcal{X}_i, v\}_{K_i^{-1}}$ };
elseif MRC is required then
    write-message := {"write",$uid(x_j)$, t_j, v,
    $\{uid(x_j), t_j, v\}_{K_i^{-1}}$ };
endif;
send write-message to at least $b+1$ servers;

Read(x_j)**:**

let t_j be the timestamp associated with x_j in $\mathcal{X}_i$;
send ($uid(x_j)$,t_j) to $b+1$ or more servers;
receive replies from these servers that includes
the meta-data of x_j ;
let t_r be the highest timestamp for data item x_j
among the replies ;
if $t_r \geq t_j$ then
    choose the server which sent t_r in its reply;
    send {"read", $uid(x_j), t_r$} to chosen server;
    receive $M = \{t_r, \mathcal{X}_{writer}, v,$
    $\{uid(x_j), t_r, \mathcal{X}_{writer}, v\}_{K_i^{-1}}$ }
    accept v if the signature is valid;
    if MRC consistency is required then
        update t_j in $\mathcal{X}_i$ to t_r when $t_r > t_j$;
    if CC consistency required
        update each timestamp in $\mathcal{X}_i$ to max of value
        in $\mathcal{X}_i$ and the corresponding value in $\mathcal{X}_{writer}$;
else
    contact additional servers or try later

Figure 17: Read and Write Protocols Executed by Client C_i

A client can detect old or corrupted data by verifying the signature and examining the associated meta-data. By writing the data to at least $b+1$ servers, we ensure that at least one non-faulty server receives the data and will store it correctly. However, such a non-faulty server to which the last data value was written may not be among the $b+1$ servers that are contacted by a subsequent read operation. In this case, the data supplied by the non-faulty server may be stale according to the *context* of the requesting client and it will not be

accepted. To increase the likelihood that a client's read request does include a non-faulty server with current value of the data item, we add a dissemination component to the protocol presented so far. Secure dissemination protocols that allow servers to exchange data values were discussed in chapter 4 [36, 37, 43, 28]. New data values could be disseminated to one or more servers at a frequency that can be tuned according to the needs of the clients or the resources available to the servers. We do not discuss further dissemination in this chapter and assume that a secure dissemination protocol is used in the background.

The protocol presented in Figure 17 may not return a value with a timestamp that is greater or equal to the data item's timestamp in the client's *context*. Several options exist for handling this case. For example, additional servers may be contacted or the client can try the operation at a later time when the new value may have been disseminated to the servers that it contacted. Thus, the cost of a read operation will depend on the dissemination protocol as well as the frequency with which data items are updated.

The correctness of the protocol follows from two observations. First, no malicious server can modify any data item since all data items are signed. Second, consistency is enforced by the client accessing the data. Since a single client writes the data (both non-shared and shared data that others only read) and the writer monotonically increases the timestamps on updates, timestamps in client *context*s or in the meta-data stored with object values can always be ordered. In the write protocol, the meta-data is included for computing the signature and non-faulty servers forward the entire write message. Thus, a malicious server can neither successfully disseminate spurious data values nor can it change the meta-data associated with values.

5.4.3 Protocols for Multi-writer Applications

We now consider the case when shared data items are both read and written by multiple clients. Because we only provide MRC and CC consistency, if the writers generate ordered timestamps for their updates, the protocol in Figure 17 will still be correct since the timestamps stored with data values will define an order that will be consistent with causality. However, because the writers can generate values independently, ordered timestamps cannot

be guaranteed. This could create several problems for the earlier protocol. First, without any coordination between writers, two distinct values v_1 and v_2 for data item x could have the same timestamp. In the protocol in Figure 17, it is possible for reads of x by a client to return v_1 followed by v_2 and then v_1 again. This is clearly undesirable. Servers that participate in the dissemination protocols can also get confused when distinct values of x have the same timestamps.

The protocol in Figure 17 can be adapted for multiple writer case by changing how timestamps are associated with data items. In particular, with a time t, we also include the unique identifier of the client that generated t to create the timestamp. Although this dissertation does not address the case of malicious clients, the protocol can presented in figure 17 can defend against a malicious client using the timestamp of another client. This is because the signature associated with writes includes timestamps as well, and the key used to sign should match the *uid* of the client in the timestamp. To prevent a malicious client from using one timestamp for two different values it writes, we also include the digest of the value written in the timestamp. Thus, a timestamp becomes a 3-tuple ($time$,$uid(C_i)$,$d(v)$). Two timestamps $ts_1 = (t_1, uid_1, d_1)$ and $ts_2 = (t_2, uid_2, d_2)$ are first ordered based on the value of the time associated with the timestamp. If t_1 and t_2 are the same, the timestamps are ordered based on the *uid*s of clients contained in them. If the *uid*s are same as well, the digests should be the same. Otherwise, the writer is deemed to be faulty. In this case, clients accessing this data item can be informed that the value cannot be assumed to be correct. The augmented timestamps are associated with data values and the signed digest reflects both the timestamp as well as the value. It should be noted that malicious clients can write garbage values. This cannot be prevented and must be detected at the application level. We have developed more complex protocols that can handle malicious clients which are discussed in [27].

5.5 Modified Versions of Secure Store Protocols

We now present an extended version of the read and write protocols presented in chapter 3 to meet consistency requirements. The technique we use to guarantee consistency would be

Write(x_j, v) by client C_i, $x_j \in X_j$:

1. Let timestamp ts = current clock value concatenated with uid(client). Update $\mathcal{X}_j$ with ts for x_j.
2. Fragment value v into c shares v_1,v_2,..v_c using a (b, c) secret sharing scheme.
3. Compute one-way function of each of the shares, $h(v_i)$.
4. Form the verification string VS and compute signature
   $VS = h(v_1)|h(v_2)...|h(v_c)$. (concatenation).
   $sig = \{uid(x_j), ts, v\}_{K_{c_i}^{-1}}$, where $\{\}_{K_{c_i}^{-1}}$ denotes signature using private key of the client.
5. Choose a row k.
   for ($m = 1$ to c){
   send {"write",uid(x_j),ts,$\mathcal{X}_j$,v_m,VS,sig}
   to server S_{km}.
   }
6. Repeat 5 for a different row until $c - \lfloor b/l \rfloor \geq b + 1$ where l is the number of rows contacted.

Read(x_j) by client C_i, $x_j \in X_j$:

1. Let t_j be the time associated with x_j in $\mathcal{X}_j$.
2. Choose a row k.
   for $m = 1$ to $2b + 1$ {
   send {"read",$uid(x_j)$,t_j} to S_{km}.
   }
3. Receive $\{t_r, VS, \mathcal{X}_{writer}, v_m\}$ from each server.
5. A triplet $\{t_r, VS, \mathcal{X}_{writer}\}$ is said to be "good" if it appears in at least $b + 1$ replies.
   Let t_r be the highest timestamp that appears in a good triplet.
6. If there is no good triplet or if $t_r < t_j$, repeat from 2 for a different k.
7. Pick shares corresponding to t_r. Pick $b + 1$ shares among these that are successfully verified by the verification string. Reconstruct the data value.
8. Check if signature is valid. If valid, return the reconstructed data value. If signature is not valid, repeat from 1 for a different k.
9. If MRC consistency is required then
   Update t_j in $\mathcal{X}_j$ to t_r when $t_r > t_j$;
   If CC consistency is required then
   Update each timestamp in $\mathcal{X}_j$ to max of value in $\mathcal{X}_j$ and the corresponding value in $\mathcal{X}_{writer}$;

Figure 18: Data Access Protocols for *Secure Store* with Consistency Guarantees

the same as the one used in the earlier section. In particular, the definition of the *context* object and the way the *context* object is managed does not change. We retain the same notation we used in the earlier sections for data items, timestamp and the *context* object.

Context captures a client's interaction with the store in the past. A client uses its *context* information to determine what values are acceptable with respect to the consistency level associated with a given data item. A client, before it starts interaction with the store, initializes its *context* for a given set of data items. Initially *context* consists of null timestamps for all the data items in the set. The *context* is continually updated as the client interacts with the store. At the end of the session, the client saves its *context* so that it can be retrieved and used in a later session. We discussed in an earlier section how this *context* is stored. For the rest of this section, we assume that a client always has a valid *context* for a given data set before it starts its interaction with the store for any data item in that set.

Figure 18 shows the extended version of the read and write protocols with consistency guarantees. These protocols differ from the earlier described set of protocols in that *context* information accompanies all reads and writes. Writes of data shares are accompanied by the *context* information of the writer which is stored along with the share at the servers. Data shares retrieved in a read request are also accompanied by the *context* of the writer that is stored along with the share. Upon a successful read, the local *context* of the reading client is updated with the $context_{writer}$ retrieved along with the shares. At any point, *context* stored locally at the client site contains the oldest possible timestamps of the data items that are acceptable for future reads. Thus, a client will never accept a value for a data item that has a timestamp older than the timestamp stored in its local *context*.

The only difference between MRC and CC consistency is in the way the local *context* is updated with the $context_{writer}$ read along with the data shares. For MRC, only the timestamp of the particular data item being read is updated while for CC timestamps associated with all related data items in the local *context* are updated.

For MRC, a client's *context* always captures the timestamp of its latest write or read for all the data items in the corresponding data set. Thus, clients are always guaranteed to

read a value which is not older than the value already read in the past for that data item.

For CC, during a write, *context* of the writing client captures the timestamps of all the reads and writes that causally precede that write. This *context* is stored as $context_{writer}$ along with the data shares. Thus, when a client reads and accepts a data value, it updates its local *context* with timestamps of all reads and writes that causally preceded the write operation that wrote the value. Hence, in future, the reading client will never accept a value for a data item that is older than a value that causally preceded any of its reads and writes.

Since *context* itself is accepted as valid only if $b + 1$ servers return the same context, a malicious server cannot make a client accept a spurious *context* that was not written by any client. While *context* helps clients avoid reading older values for data items, we assume that newly written data values will eventually be accessible to clients either because of background dissemination or because of retries by clients upon failed reads. In the steady state when no writes are taking place, a read by a client with a valid *context* is bound to succeed as long as the earlier writes were completed successfully by non-faulty clients.

For MRC, it is not necessary to send the whole *context* of the data set in a write, just the timestamp of that particular data item alone can be sent. Reading clients would update their local *context* with the timestamp of the value. To ensure that clients can read values for a data item even when a write is going on concurrently, servers can employ versioning mechanisms to store shares and associated meta-data rather than overwrite shares. Goodson et. al. used a similar technique to deal with client crashes in [19]. However, in this case, servers should return a list of shares and meta-data for a read request.

5.6 Performance

In this section, we briefly discuss the performance benefits of using weaker consistency models in distributed storage, when compared to systems that offer stronger consistency levels like atomicity or safe semantics. For the sake of clarity, our discussion in this section will be based on the simple storage scheme described in section 5.4. At the end of the discussion, we will summarize how the metrics change when secret-sharing is used.

We distinguish operations that are used to acquire and store *context* data from those

to read and write other data items. The quorum sizes are different in these two cases.The size of *context* data depends on two factors: (1) the number of related data items in a given group, and (2) the number of groups from which data items are accessed in a session. For example, all documents containing tax related information for a given year could be considered related. An application may access tax documents as well as documents that store information about medical bills. Once the size of *context* data is determined, its acquisition requires round-trip communication with $\lceil (n+b+1)/2 \rceil$ servers, where n is the total number of servers and b is the constant bound on the number of servers that could be faulty. The *context* data also needs to be stored at the same number of servers when it is written on session termination. Thus, the message costs and the response time of *context* read and write operations depends both on the total number of servers and the number of servers that could exhibit malicious behavior. In particular, a total of $2 * \lceil (n+b+1)/2 \rceil$ messages will be exchanged between the client and the servers to retrieve or store the *context*.

The cost of read and write operations for non-context data depends on both the quorum size as well as on the rate at which new values are propagated among servers. A write operation can complete for all types of data (non-shared, shared with MRC or CC consistency) by communicating with $b+1$ servers. This gives a total of $b+1$ messages for write operation. Since the operations can be completed by communicating with only $b+1$ servers, their response time will be better than the response time of *context* operations. In the best case, the message cost and response time of read operations could also be the same as write operations. This will be the case when one of the server that responds to the read quorum request has copy of the data item that is consistent according to the client's *context*. However, if the desired data value has not been propagated to the servers in the read quorum, either additional servers must be contacted or read must block until the needed data value is disseminated to one of the servers in the quorum. The dissemination protocol would require additional communication between the servers. The frequency of such communication will depend on the resources available to servers as well as the read response time desired by the applications.

	atomicity	safe, regular	MRC, CC
access cost in #servers	n	$O(\sqrt{(bn)})$	read- $2b+1$ write- $2b+1$ + dissemination
throughput capacity, ratio to that of atomicity	1	$O(\sqrt{(n/b)})$	read-$(n/(2b+1))$ write - $O(1)$
scalable	no	yes	yes

Table 2: Cost of guaranteeing various consistency levels, tolerating b malicious faults with n servers. Scalability here refers to whether average load per server can be reduced by increasing the number of servers in the system.

Table 2 compares the performance of systems that guarantee various levels of consistency with n servers, tolerating b malicious servers. The communication overheads of our protocols are better than the quorum protocols that provide strong consistency. For example, if majority quorums are used, Byzantine quorums require communication with $\lceil (n+2b+1)/2 \rceil$ servers for both read and write operations. Although improved quorum design can reduce their sizes [38], a minimum quorum size of $\sqrt{n}$ is necessary. Depending on whether reads or writes are expected to be more frequent, asymmetric quorum systems choose optimal sizes for read and write quorums. We can achieve lower overhead even for *context* operations by using an improved quorum design. For non-context data, the message cost of writes is much lower and when writes are infrequent, most reads will access data that has been disseminated to all servers. In this case, the average cost of reads will be close to the costs of writes. Thus, by providing weaker consistency when appropriate, significant savings in communication can be realized. The savings in communication translate to reduced overhead at servers. Since less number of servers are engaged to complete a read in our system, our system can support higher throughput for reads with the same number of servers. For writes, dissemination shifts the load over time and higher throughput cannot be sustained over long periods of time. However, our system can support higher peak

throughputs for writes when dissemination is given a lower priority. For a work load with a mix of read and write requests, the extent to which higher throughput can be sustained depends on the ratio of reads to writes.

More importantly, our weaker consistency system is more scalable than Byzantine quorums. By, adding more servers to the system, improvement in throughput capacity can be better than quorum systems. Adding more servers does not increase the message costs too in our system while these costs go up in quorum systems.

The state machine based implementation of Byzantine fault-tolerance reported in [35] provides better performance than earlier systems. This is primarily due to lower computational overheads of message authentication codes that are used in place of signatures. Although this approach offers computational savings, it has significantly higher message overhead. For example, the multi-phase protocols require $O(n^2)$ messages. This could lead to high response time for operations, specially in an environment where communication latencies are high across the server replicas. Since every server needs to participate in every read or write request, the throughput supported by a system with this approach is much less than weaker consistency and quorum systems for the same number of servers. In addition, a system using this approach cannot be scaled to support better throughput by adding more servers.

By opting for weaker consistency levels, our system's performance is both better and more scalable than systems that offer stronger consistency levels, in terms of the message costs for reads and writes and the system throughput. Using secret sharing mainly introduces additional computation cost that is spent in encoding and decoding the shares. Also, minimum number of servers that need to be contacted for a write becomes $2b+1$. However, the essential benefits of using weaker consistency levels remain, namely reduced message costs, increased system throughput and better scalability.

5.7 Summary

In this section, we dealt with the consistency issue in data access. We discussed two weaker consistency models *secure store* supports, namely monotonic read consistency and causal

consistency. We took the approach of shifting the responsibility of maintaining consistency to clients. Clients make use of *context* objects to make consistency related decisions. We illustrated this technique using a simple system that uses pure replication to store data. We integrated this technique into *secure store* protocols and gave a modified version in this chapter. We also discussed some potential benefits of using weaker consistency levels in distributed storage in the presence of Byzantine faults.

CHAPTER 6

IMPLEMENTATION AND EXPERIMENTAL EVALUATION

6.1 Overview

As a proof of concept, we have implemented a file system based on the techniques discussed in earlier chapters. Our implementation demonstrates the feasibility and practicality of our approach. This chapter describes the design, implementation and evaluation of the file system based on *secure store*. We first give an overview of the implementation, briefly describing various components. We then describe a set of experiments to evaluate the system performance. We illustrate the agility of *secure store* by experimentally demonstrating the tradeoff our store offers between security and performance and justify the theoretical analysis we described in chapter 3. In this chapter, we will concern ourselves with the evaluation of *secure store* as a whole and not consider individual components or techniques. We presented a detailed evaluation of the dissemination protocol in chapter 4.

6.2 Introduction

Although a number of different implementations could be conceived for *secure store*, we chose a file system implementation for two reasons: (1) the file system interface is the most popular interface for storage and generally well understood by every one, and (2) a number of existing applications are file systems ready and we could use them for both demonstrations and bench marking.

The implementation, besides being a proof of concept and demonstrating the feasibility and practicality of the *secure store* design, also helped us develop an understanding of the various costs involved in a realistic system designed using such techniques. More importantly, measurements with our implementation validated the tradeoff graphs we presented in chapter 3. We hope that this implementation would also be useful for further research

in secure distributed storage.

The storage service is offered by a set of data servers. Another service called metadata service is implemented by a separate set of servers. File system service is offered by the metadata service along with the storage service. Both the storage service and the metadata service are designed to tolerate a limited number of compromised servers at any time.

The fault model assumed is Byzantine faults. A faulty server node can behave in any arbitrary fashion. Number of faults (including benign ones) to be tolerated, b, is left as a parameter chosen by the clients for each object that is stored. This parameter, called the threshold value, reflects the maximum number of faults that can occur in any continuous time interval of length T_v seconds before the security of the stored object is compromised. The constant T_v, called the vulnerability window, is a design parameter. The secure store file system guarantees (1) data availability to authorized users, (2) data integrity, and (3) data confidentiality. The file system service makes use of cryptographic primitives, secret sharing schemes and replication techniques to guarantee these properties.

6.3 Implementation Overview

Figure 19 shows an overview of the implementation. The file system is implemented by three components: (1) Client agent, (2) Storage service offered by data servers, and (3) Metadata service.

Client applications access the file system through a client agent. Client agent is a user-level program that runs on the client machine. Client agent is mounted as an NFS server at the client machine. Client agent receives file system calls from client applications as NFS calls via kernel NFS client at the front end. It interacts with data servers and the metadata service at the back end.

Data servers are primarily concerned with storing and retrieving data shares[1]. Each data server authenticates and authorizes each client request independent of other data servers before servicing the request. Authorization is based on a capability or an authorization token

[1] A data block is transformed into a set of data shares using a library that implements the secret sharing algorithm.

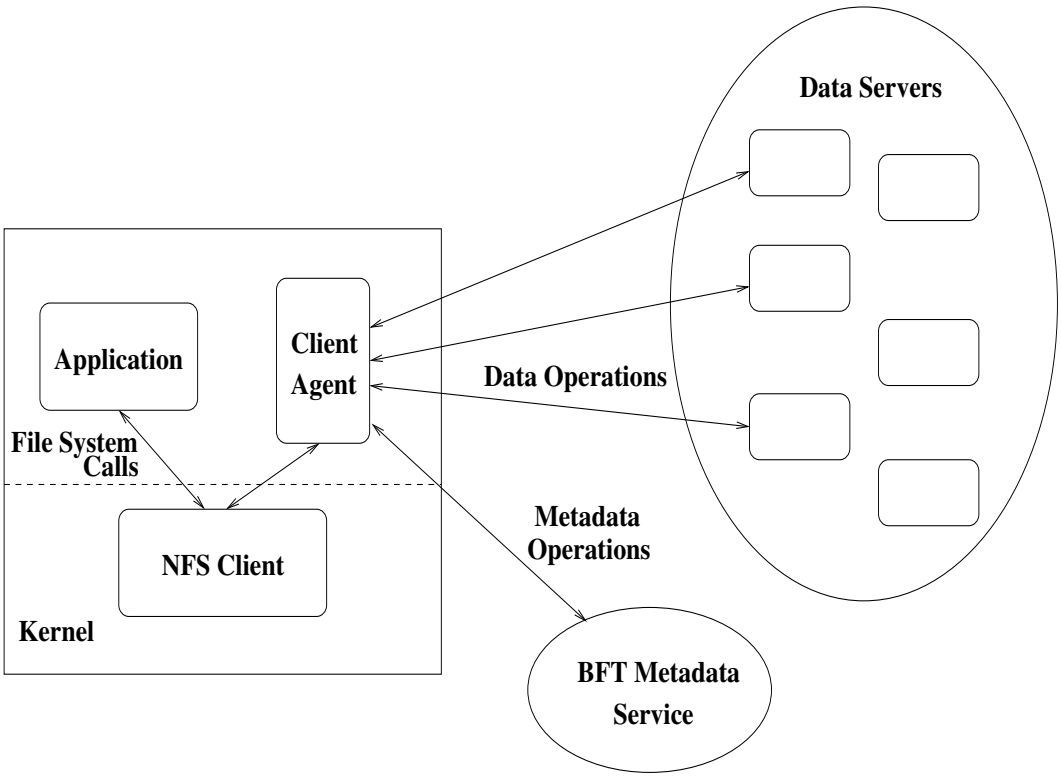


Figure 19: Secure Store File System - Schematic Overview

issued by the metadata service. Data servers also exchange data updates periodically among themselves and renew data shares periodically if a file requires long-term confidentiality. Along with the data, some of the attributes of a file that do not require strong consistency (e.g, last modified time) are stored at the data servers.

Metadata service is a Byzantine fault tolerant state machine implemented using a set of metadata servers. Metadata service is concerned with maintaining directory tree and the metadata associated with each file or directory that require strong consistency, like access permissions, security parameters and other necessary information. Metadata service also issues authorization tokens to client agents, which are presented to data servers for authorization at the time of access.

Client agent employs the read/write protocol presented in earlier chapters 3 and 5 to do data transactions with a subset of data servers. Client agent does all other name-space and meta data related operations with the metadata service.

Metadata service is implemented by a relatively smaller set of metadata servers located

close to each other on a shared LAN. The metadata service is tuned to service high volumes of small requests (not data or computationally intensive), most of which require consensus among the servers. Consensus is achieved typically using multiple rounds of messages (see [35]). Hence metadata servers should be on a high-speed low-latency LAN with preferably a broadcast facility and should be able to process network messages at a high speed. It should be noted that write operations are the ones that are expensive and read operations are both fast and impart less load on the servers. The operations that are frequently carried with metadata service in our file system are read-only (e.g, looking up uid of a file or requesting a capability) while write operations like changing access control lists are less frequent. Frequency of operations that change the directory structure are dependent on the workload and we expect such operations, for the applications we target, to be less frequent. Metadata service is not the focus of our research. Others have looked at implementing such services based on state-machine approach [54, 35, 2]. Hence this service is replaced by a single central metadata server in the prototype with which we did our experiments.

In contrast to metadata servers, data servers are loosely coupled, distributed over a wide area. Data servers do data-intensive storage, retrieval and data transmission operations and hence are tuned for the same. Some servers designated as share renewal servers may also do high volumes of computation.

All communications between any two parties are through a secure channel, authenticated and encrypted whenever necessary. This is accomplished through a key management infrastructure which helps any two nodes negotiate symmetric keys. Such keys may be used for other purposes too, e.g, authorization token. Key management infrastructure makes use of meta data service as certification service or key distribution center, depending on the kind of key management scheme used.

Authorization framework has not been implemented in the current system. Other members of the Agile Store group are looking at this issue that is common to other distributed storage approaches.

6.4 Client Agent

User applications access the *secure store* file system through the client agent. Client agent is a user level program that is mounted on the client machine as an NFS server. File system calls made by client applications are handled by kernel NFS client. Kernel NFS client makes NFS RPC calls to the client agent which is the NFS server. Client agent receives these NFS RPC calls at the front end and services these calls by interacting with the data servers and the metadata service at the back end.

Client agent was chosen to be implemented as a user level program mainly for ease of implementation and testing. This leads to a performance penalty in directing the calls, crossing the user-kernel boundary. However, this overhead is negligible when compared to the total time taken to complete a write and acceptable for a read operation. More importantly running the client agent at the user-level has security implications since any program could run as an NFS server at the user level and receive the file system calls. Thus, client agent should be authenticated if it is run as a user level program. Hence, for both security and performance reasons, the client agent in real systems should be implemented as an in-kernel file system at the VFS layer.

TCP was chosen for communication with the data servers since data access operations typically involve sending or receiving long messages that include one or many data blocks. Besides, connection set up latency is negligible when compared to other costs in the read/write operations. In contrast, UDP was chosen for communication with metadata server since these interactions typically involve short messages, clients requests are implicitly acknowledged with server's reply and flow/error control is not a significant requirement.

6.4.1 File System Issues

Client agent is an intermediary between the client application and the secure store servers. Client agent is implemented as an NFS version 3 server. All the NFS RPC calls listed in RFC 1813 are implemented except a few (calls related to symbolic and hard links). All the calls except read and write are hence forth referred to as metadata operations. Read and write are referred to data access operations.

All metadata operations (directory and access control related operations) are done with the metadata service with the exception of GETATTR and LOOKUP RPC calls. File attributes other than access control lists are stored at the data servers along with the data blocks. All the file attributes associated with a file like size and last modified time require the same level of consistency as the data itself and hence a design decision was made to store these at the data servers. As a consequence, GETATTR call is directed to the data servers rather than the metadata service. GETATTR operation is implemented similar to the read operation, voting on the replies (as opposed to secret share decoding for data blocks). LOOKUP call is serviced by interacting with both metadata service and data servers. LOOKUP call in the NFS specification also returns file attributes and hence attributes are fetched from the data servers.

Data access operations involve the data servers. In any read or write operation, data blocks are also accompanied by the latest attributes of the related file. Client agent maintains a write-though disk-resident data cache using the local file system. Although NFS kernel client does data caching, NFS interface allows for reads and writes to be requested crossing the data block boundary (there is no notion of a block in NFS!). However, *secure store* is a block storage service and hence client agent needs to maintain a data cache.

Client agent also maintains an in-memory attribute cache. NFS client relies on file attributes (mainly last modified time) to ensure data cache validity. For this reason, NFS client expects object attributes of related objects to be returned along with every operation. For this reason, client agent caches attributes. However, client agent does not frequently request attributes from data servers like the NFS client. Client agent assumes that attributes in its cache are always valid. This relies on some mechanism like the one suggested in AFS to ensure cache validity. However, solution adopted by AFS cannot be applied to *secure store* since there is no centralized server. A practical solution would be to offer time-bounded consistency, periodically invalidating the attribute cache. This particular issue has not been addressed in the current implementation.

6.4.2 Data Access Operations

Write and read operations proceed according to the protocol specified in chapters 3 and 5. To write a data block, client agent first timestamps the block, secret shares the data block, computes the verification string, randomly chooses a row of servers and sends one share to each server along with the verification string, timestamp and other associated metadata. To read a data block, client agent chooses a row of servers randomly, requests for shares, collects enough shares, and makes sure that all shares belong to a single write by comparing the timestamps. It then chooses a verification string that appears in $b + 1$ replies, verifies each share using the chosen verification string and finally uses the verified shares to decode data block.

Client agent uses a secret-sharing library to encode and decode data blocks. We implemented a secret-sharing library that operates in the Galois field GF(2^{32}). Although there were standard implementations for this library (e.g, crypto++ library), these libraries work in the integer field modulo a prime which is generally slower. Using Galois field resulted in encoding and decoding operations being at least five times faster than other implementations. Also, writing our own library would allow us to hand-tune the implementation to improve the speed of the function calls. This is important since a significant fraction of time in completing a read or write operation is spent in encoding or decoding the data blocks.

6.5 Data Servers

A data server is implemented as a daemon that runs on a specified TCP port waiting for requests from clients and other data servers (for dissemination requests). A data server stores the blocks in the form of secret shares and the associated meta data as files in the local file system. A data server also maintains a number of in-memory queues meant for dissemination. One data dissemination queue is maintained for each possible value of column. Another queue is maintained for the endorsement protocol, common to all data blocks. Data dissemination queues are implemented as doubly linked lists while the endorsement queue is a hash table with O(1) lookup, insert and delete operations.

Upon a write request, data server verifies that the received data share matches the

accompanied verification string and stores the data share and the verification string on the disk using the local file system. The server also does the following: (1) enqueues the share information in an appropriate queue meant for data dissemination, (2) generates endorsement for the write and stores the verification string and the associated MACs in the endorsement queue. Upon a read request, data server simply retrieves the concerned data share, attributes and verification string from the local file system and returns it to the requesting client.

Each data server also runs two other concurrent threads: (1) data dissemination thread and (2) endorsement thread. Data dissemination thread periodically wakes up, for every possible value for columns, chooses a random server in the same column and requests an update of data shares. The peer data server replies with a list of uids of the data shares from its dissemination queue, along with the timestamp information. The requesting server checks if it has the latest copy of the share for each share id received. The requesting server sends a bit map, indicating the blocks for which it does not have the latest copy. The peer server finally sends all the missing data shares, retrieving from its local file system. This prevents a data share from being sent to a data server multiple times.

Endorsement thread, periodically wakes up and runs the collective endorsement protocol, requesting MACs, attempting verification, keeping count of the number of verified MACs for each block update and finally accepting a valid verification string and generating endorsements in turn. Upon accepting a verification string, its marked as verified in the local disk. If the stored data share does not match the accepted verification string, a share recovery protocol is initiated. The share recovery feature is not implemented in the current system. Periodically old entries in the queue are removed. Thus, if a verification string is not verified within a period of time, that verification string (and the uid of the write) is dropped.

Both endorsement and data dissemination queues are periodically purged of old entries (each entry in all these queues carry a time stamp). Also, data dissemination traffic is limited by a configurable parameter so that data servers can give higher priority to read/write

requests than dissemination requests in utilizing their resources. The main servicing component that attends to requests from clients and other data servers is itself multi-threaded. The servicing component is implemented using the worker-dispatcher model having one dispatcher thread and a fixed pool of configurable number of worker threads.

6.6 Experimental Evaluation

We describe here a set of experiments that were conducted to evaluate the file system based on secure store. We are primarily interested in two metrics: (1) time to complete a read or a write and (2) maximum read/write throughput the system can support. The purpose of this study is to show the agility in the system and to understand the costs involved in such a system. We demonstrate the agility in the system by showing the tradeoffs our data access protocols offer between security and performance. Our experiments also demonstrate the feasibility and scalability of *secure store*. For the purpose of this section, we will consider the fault tolerant threshold parameter b as the primary indicator of security level.

We experimented with our secure store file system implementation in the Emulab testbed environment at Utah [1, 59]. Experiments were conducted in both loaded and unloaded situations. The unloaded situation consists of a single client performing a sequence of read or write operations on files in the store. The relevant performance metric of interest in this situation is the average latency of the read and write operations. In the loaded situation, there are multiple clients performing simultaneous read and write operations and we evaluate the maximum throughput the data servers can support in each of the approaches. Both latency and throughput were measured by varying the fault tolerance threshold. We used a single metadata server for the metadata service in all our experiments.

We used 15 to 30 machines from the testbed for each of our experiments. RedHat Linux 7.3 was loaded on all of the machines. Some of the machines were 600MHz PIII "Coppermine" processors with 256 MB RAM, 13GB 7200RPM IDE hard drive and 5 100Mbps Ethernet cards. Other machines were 850MHz PIII processors with 512MB RAM, 40GB 7200RPM IDE hard drive and 5 Ethernet cards. All the chosen machines were configured to be on a switched Ethernet, with a star topology. All links had a 100Mbps link bandwidth

with negligible network latency.

In all the experiments, a block size of 8KB was chosen for reads and writes. While the fault tolerance threshold parameter was varied between 1 and 7 for a system of about 15 servers, the number of columns was always chosen to be $2b + 1$. Read operations were performed using "grep" on the files stored in the secure store and write operations were performed using "copy" to copy files from a local file system to the Agile Store. Latencies of these operations were measured for 500 files and averaged over 50 runs.

To measure throughput, the file system interface was bypassed and a user level client application made calls directly to the client agent library. Multiple such client instances were started on a number of machines to perform as many reads/writes as possible in a given time. The experiment was repeated five times and the maximum number of operations among the five runs was noted as the achieved throughput. The number of client instances was increased until maximum number of operations performed stopped to increase or started to decline.

6.6.1 Cost of Writes

b	writes			reads		
	encoding latency	total latency	percentage computation	decoding latency	total latency	percentage computation
1	17.4 ms	33.7 ms	51.61%	4.4 ms	19.4 ms	22.78%
2	41.1 ms	60.9 ms	67.46%	6.4 ms	23.9 ms	26.98%
3	74.6 ms	96.7 ms	77.15%	10.3 ms	28.5 ms	36.26%
4	120.1 ms	144.7 ms	83.10%	14.5 ms	35.1 ms	41.36%

Table 3: Cost of secret sharing computation: time spent in encoding and decoding shares as a percentage of total time taken to complete a write/read for different values of b

Figure 20 shows the latency of completing a write of a 8KB block for different values of fault tolerance threshold b. It takes about 33 ms to write a 8KB for a b of 1 and increases to about 145 ms when b is 4. The latency can be split into three parts: (1) time spent in

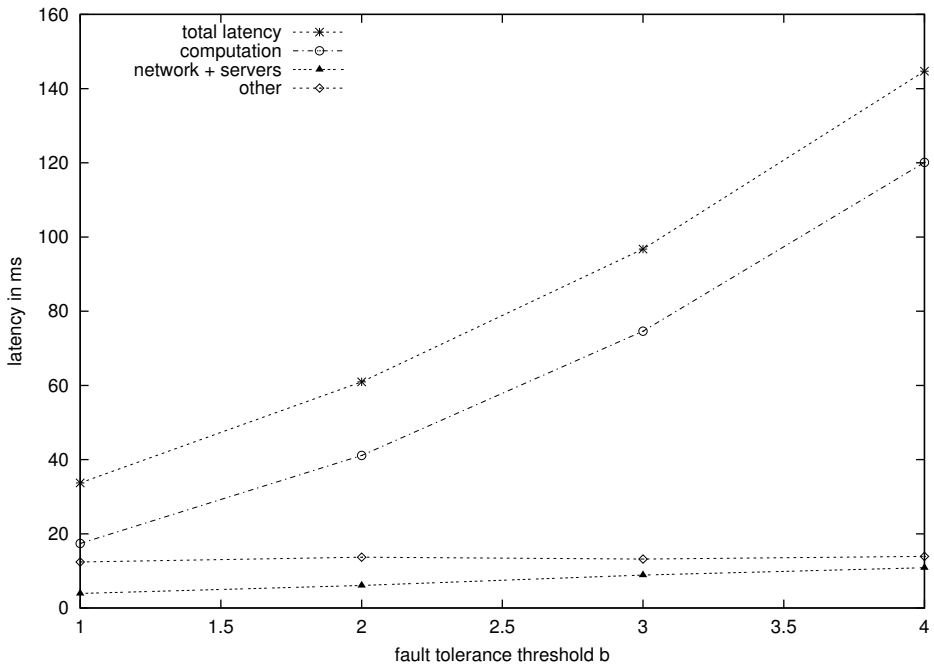


Figure 20: Cost of writes in *secure store*: Time taken in milliseconds to complete a write of an 8KB block as a function of the fault tolerance threshold parameter b. Latency due to computation, network and server side latency and other costs are also shown.

computing the shares, (2) time spent sending and receiving messages, network latency and latency at the servers before they reply, and (3) others, e.g, time spent in forwarding the call from the application to the client agent and forwarding the reply back to the application. Figure 20 also shows how the total latency is distributed among the three components.

As can be seen, the dominating cost is the time spent in computing the shares. Table 3 summarizes this fact by showing the percentage of total latency spent in computing the shares. From 51% for a b of 1, the amount of time spent in encoding raises to 83% of total latency for a b of 4. With our implementation of the secret sharing library, it takes about 17 ms to compute the shares for a b of 1 and increases to 120 ms when b is 4. It can be seen from the figure that both the total latency and the computation time display a quadratic behaviour in b. This is because secret-sharing encoding is $O(b^2)$. Using Shamir's polynomial interpolation, encoding involves computing $2b + 1$ shares and computing each share involves evaluating a b^{th} degree polynomial at a specific point.

The time elapsed between sending the first request message to a server and receiving

the last reply increases linearly from about 4 ms to 11 ms. This is because write involves communicating with $2b + 1$ servers, sending to each server a share which is of same size as the data block itself, $8KB$. This metric also includes the time taken by each server to reply to the message with an acknowledgement after checking if the received share can be written to the disk. However, the servers reply to the write message even before writing the shares to the disk. Processing at servers is done in parallel and hence time spent at the servers does not depend on the number of servers.

Other factors that contribute to the latency include (1) transporting the application call to the client agent and back, (2) on-disk caching by the client agent, and (3) accessing the metadata service. Latency due to these factors is a constant that does not depend on b. It was measured to be about 12 ms. Although this latency is greater than latency due to communication over the network, it is negligible when compared to the total latency of writes, particularly for higher values of b.

6.6.2 Cost of Reads

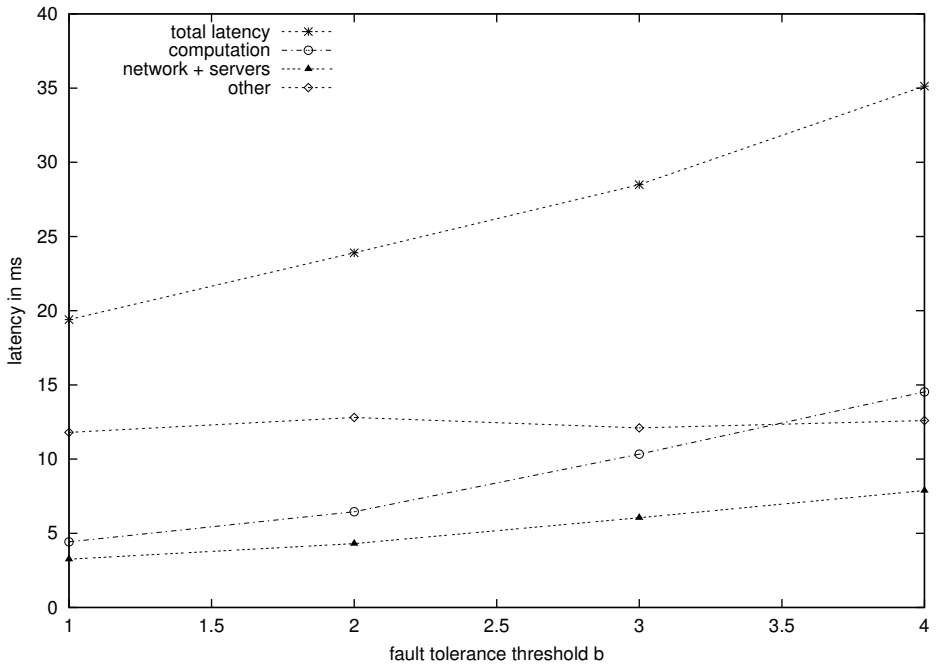


Figure 21: Cost of reads in *secure store*: Time taken in milliseconds to complete a read of an 8KB block as a function of the fault tolerance threshold parameter 'b'. Latency due to computation, network and server side latency and other costs are also shown.

Figure 21 shows the total time taken to complete a read of a 8KB block for various values of b. The total latency increases from $20ms$ for a b of 1 to about 35 ms for a b of 4. The figure also shows the time spent (1) in decoding the shares, (2) between sending requests to the servers and receiving the shares and (3) due to other factors.

Reads are considerably cheaper in our system mainly because secret sharing decoding is significantly faster than encoding. While encoding the shares involves $O(b^2)$ computation, decoding involves finding the y-intercept of the polynomial from the values the polynomial takes at $b+1$ points. This can be computed in $O(b)$ time. Secret sharing decoding takes 4.4 ms for a b of 1 to about 14.5 ms for a b of 4.

Latency due to communication with the servers, sending requests and receiving replies, is linear in b, as in the case of writes. This latency increases from 3.25 ms for a b of 1 to about 7.9 ms for a b of 4. Although this latency includes the time spent by servers reading the shares from theirs disks, the total communication latency is slightly less than that for writes because of the following reason: For reads, clients send the request to $2b+1$ servers but may complete the read when first $b+1$ servers reply. Attempting to complete the read with $b+1$ servers will fail only if there is at least one of the servers contacted is malicious or faulty. We did not simulate any malicious behavior at the servers in our experiments.

Latency due to other factors is almost a constant as in the case of writes. In fact, this cost is almost the same as in writes, around 12 ms. However, this cost is a significant percentage of the total latency for values of b that we consider in this section (1-4). Thus, an in-kernel implementation would significantly reduce the latency of reads. However, for higher values of b, the time spent in computation would become dominant. This can be seen in table 3 which shows that latency due to decoding of shares becomes more and more significant as b increases.

6.6.3 Throughput

Throughput of the system primarily depends on the number of servers engaged in completing a read or a write operation. Figure 22 shows the maximum read and write throughput that the system can support with 15 servers for a block size of $8KB$ for various fault tolerance

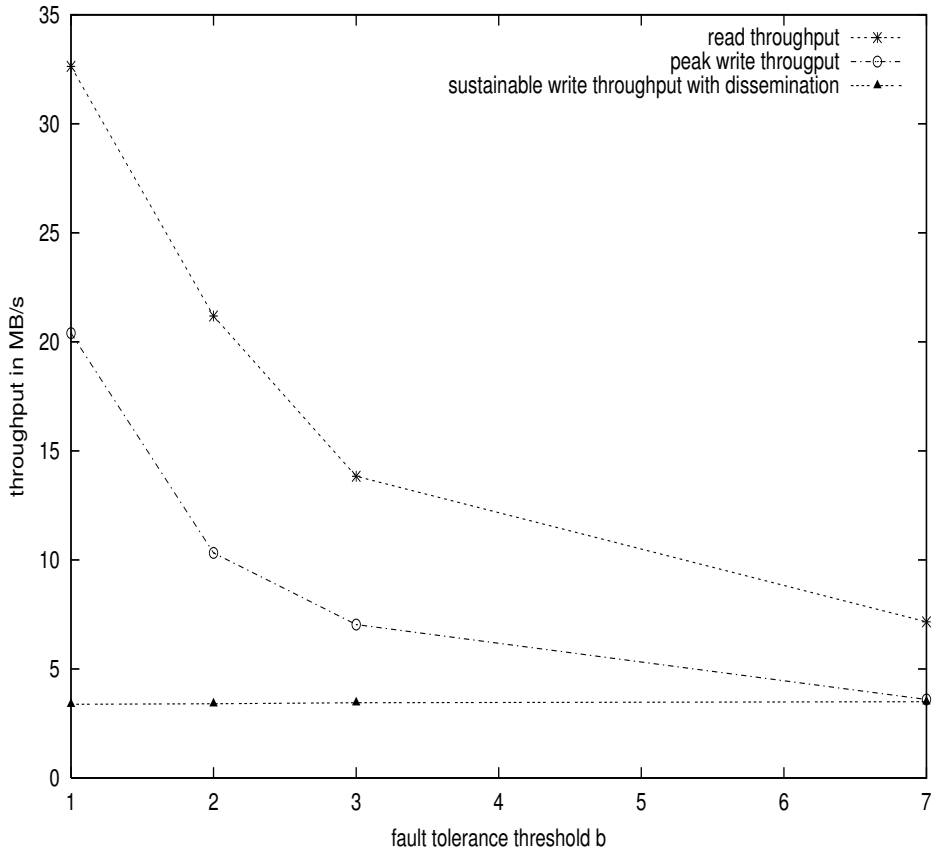


Figure 22: Throughput capacity: Maximum achievable read throughput, achievable peak write throughput and write throughput sustainable over a period of time in the presence of dissemination. All throughputs are shown in MB/s as a function of the fault tolerance threshold parameter b for a system of 15 servers and a block size of 8KB.

threshold values for b ranging from 1 to 7.

Maximum achievable read throughput for a b of 1 was measured to be 32.6 MB/s and dropped to about 7.2 MB/s when b was increased to 7. Since each read involves communicating with a row of servers, as many reads as the number of rows in the system can be completed simultaneously by the clients. Hence, the read throughput is directly proportional to the number of rows. Table 4 justifies this claim by summarizing the experimentally observed numbers.

There are two kinds of write throughputs that characterize weaker consistency systems like *secure store* that replicate the stored data using background dissemination: (1) peak throughput achievable and (2) sustainable throughput. Peak throughput is the maximum number of write operations the system can support in a short period of time in the absence of

b	no. of rows $n/(2b+1)$	read		write	
		throughput in MB/s	throughput/rows	peak-throughput in MB/s	throughput/rows
1	5	32.6	6.52	20.4	4.8
2	3	21.2	7.67	10.3	3.43
3	2	13.8	6.9	7.0	3.5
7	1	7.2	7.2	3.6	3.6

Table 4: Read throughput and peak momentary write throughput achievable in the absence of dissemination, as proportional to the number of rows, $n/(2b+1)$.

dissemination. The dissemination component competes with reads and writes for resources like network bandwidth, memory and disk time. Dissemination is done only periodically and hence at other times when dissemination is not in progress, all system resources can be devoted to completing writes. Alternatively, during times of high load, dissemination component can be given a lower priority (or totally stalled for a small period of time) and system resources can be fully used up to support a higher write throughput for a short period of time. However, such a high write throughput is not sustainable over a long period of time. If the dissemination component is not alloted enough resources to sustain dissemination at the same rate as writes introduce new traffic, dissemination queues would grow arbitrarily large and data items would not be updated sufficiently rapidly to guarantee high availability and spatial locality for reads. Sustainable write throughput is the throughput that the system can support when the dissemination component maintains a bound on the queue lengths.

Figure 22 shows both peak achievable write throughput and sustainable write throughput. Peak achievable write throughput was measured to be 20.4 MB/s when b is 1 and dropped to 3.6 MB/s when b was increased to 7. Similar to read throughput, peak achievable write throughput is directly proportional to the number of rows.

While the read throughput was limited by the network bandwidth, the peak achievable write throughput was limited by the data rate the servers could support in writing the data

to the disk. At peak read throughput, each server served data at a rate of about 56 Mb/s on a 100 Mb/s link. At peak write throughput, each server accepted writes at a rate ranging from 3.5 MB/s to 4 MB/s. In a separate experiment, maximum data rate that a server could achieve simply writing data blocks to the disks as separate $8KB$ files was measured to be about 6.6 MB/s.

While the peak achievable write throughput is comparable to read throughput and is proportional to the number of rows, the sustainable write throughput is expected to be almost a constant irrespective of the parameter b or the number of rows. Except the servers that receive a write directly from the client, replication through dissemination simply shifts the write load on the remaining servers over time. Thus, each server receives each data block (in the form of a share) and has to devote its resources like network and disk time to store the data block, although at a later point of time. The overhead due to dissemination protocol will be discussed in the subsequent section. The dissemination overhead is mostly network overhead (additional bytes exchanged over the network as control information) and is negligible when compared to the data block size. Thus, the dissemination overhead should not affect the sustainable write throughput, at least not significantly. In this case, sustainable write throughput should be the same as the peak achievable write throughput when all servers are in one row ($b = 7$) since all servers receive a write directly from the client in this case. Thus, peak achievable write throughput should remain fairly constant at about 3.5 MB/s.

6.6.4 The Dissemination Overhead

The dissemination component replicates the data written to a subset of servers to other servers in the system in the background. There are two components to dissemination: (1) dissemination of data shares, and (2) the collective endorsement component that disseminates verification strings securely.

Both dissemination components incur network overhead at each server in addition to the actual data share and the verification string that is exchanged in the gossip messages. These overheads are additional bytes sent or received as control information and metadata

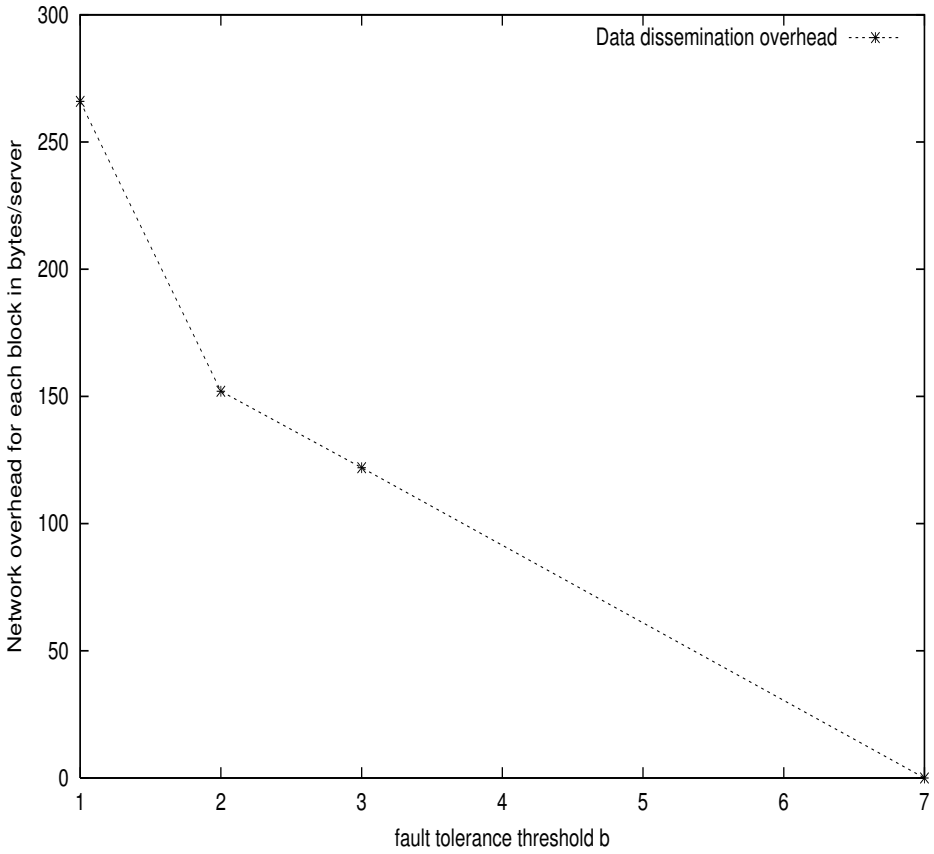


Figure 23: The network overhead incurred in disseminating data shares via dissemination, extra bytes each server sends/receives for every 8KB block for different values of b for a system of 15 servers. This is in addition to the basic 2KB network overhead each server incurs for every round of dissemination.

for the protocol. Figures 23 and 24 present these overheads. Apart from the overhead incurred per block at each server, each server also incurs a constant overhead per round for both the components, even in the absence of any write. These constant overheads are due to the messages that are sent or received to check if there are any updates to be exchanged between the gossip partners.

The data dissemination protocol is a simple gossip protocol where each server randomly picks another server from the same column and pulls updates. This is done once every 10 seconds. The constant overhead for this protocol is about 2 KB per round. In addition to this, there is an additional per block overhead. This overhead is about 266 bytes per block at each server for a b of 1 and drops to about 122 bytes per block per server when b is 3. As b increases, the number of rows decreases and the data dissemination component

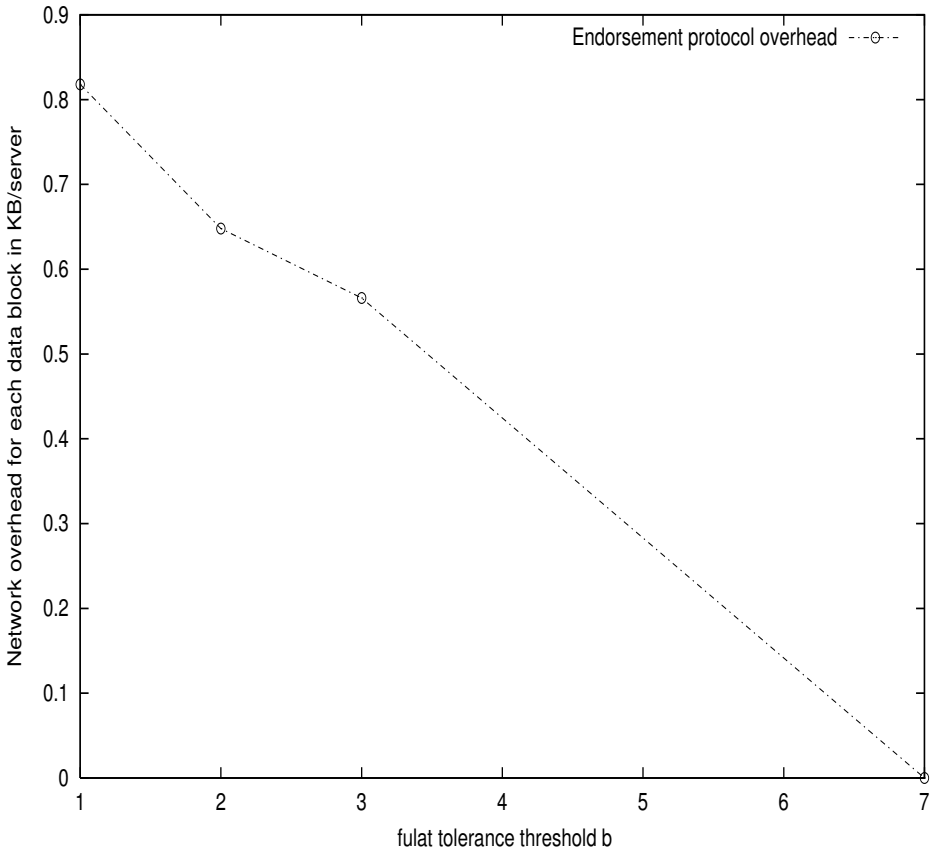


Figure 24: The network overhead incurred in the collective endorsement protocol, extra bytes each server sends/receives for every 8KB block for different values of b for a system of 15 servers. This is in addition to the basic 115 bytes of network overhead each server incurs for every round of endorsement related gossip.

disseminates the shares among smaller groups of servers. When the group among which data is disseminated is large, each pair of servers exchange extra bytes to check for data shares that might have already reached the servers in earlier rounds. With smaller groups, such instances are small. Also, our data dissemination protocol keeps each write in the dissemination queue for smaller number of rounds when the number of peers to disseminate to is less. Hence the network overhead is small when the dissemination group (number of rows) is small. . When b is set to 7, all servers receive the writes directly from the client and there is no per block overhead since dissemination does not need to transport any block between the servers.

The collective endorsement protocol implemented used a p of 7 for the prime number and thus 49 symmetric keys. The MACs were 64 bits long. The endorsement gossip was

done once every second. The constant overhead per round in the endorsement component was about 115 bytes. The per block overhead was about 820 bytes at each server for a b of 1 and dropped to 560 bytes when b was increased to 3. The reduced overhead for higher b is due to the fact that for a higher b, the initial update is introduced at a larger number of servers. When the update is introduced at a larger number of servers, there are fewer servers to disseminate the verification string to and our endorsement protocol keeps each update in the queue for a smaller number of rounds. Finally, when b is set to 7, the endorsement protocol does not keep the update in the endorsement queue and the only overhead incurred in endorsement is the constant overhead per round to check for updates.

The constant per round overhead for data dissemination is considerably higher than that of the endorsement protocol. This is because, every round the dissemination component checks for updates for other servers in the same column for every possible value of column ranging from 1 through 7. The per-block overhead in the endorsement component is mostly due to the large number of 64 bit MACs exchanged in the gossip. As can be seen from these figures, the network overheads due to dissemination and endorsement are within negligible to acceptable limits when compared to the chosen block size of 8 KB. We expect this network overhead not to affect the sustainable write throughput since limiting factor in case of sustainable write throughput is the data rate of disk writes and the unused network bandwidth can certainly support the network overhead due to dissemination.

6.7 Comparison with NFS

	latency		throughput	
	read	write	read	write
NFS	2.4 ms	5.2 ms	6.4 MB/s	3.01 MB/s
secure store	17.4 ms	31.7 ms	13.8 MB/s	7.04 MB/s

Table 5: Comparing *secure store* with Linux NFS. *Secure Store* was run with 6 servers with a b of 1.

In this section, we compare the performance of our file system with the NFS file system with standard Linux kernel NFS server. Table 5 shows the latency and throughput of our file system and the NFS file system, both measured in the emulab testbed. The measurements

for secure store are for 6 servers with the fault tolerance threshold b set to 1. Latencies are for a 8KB file. On-disk caching at the client side was disabled for these measurements. For NFS, operations were done in synchronous mode with no write delay and caching was disabled both at the client and server side. It is necessary to run the NFS server in synchronous mode to achieve a level of reliability that is comparable with secure store in which data is written to multiple servers.

The latencies for secure store are only an order of magnitude higher than those of NFS. The overhead in secure store is mostly due to secret sharing encoding and decoding. The other overhead is the constant factor of about 10 to 12 ms in secure store due to call forwarding to the client agent and a lookup call to the metadata server. These overheads become negligible for larger file sizes and an in-kernel implementation of client agent.

Throughput in secure store is better (little more than twice) than NFS because with 6 servers and a b of 1, there are two rows of servers that can service reads and writes in a parallel fashion. Still, NFS throughput is marginally smaller than secure store's throughput for one row. This is because Linux's NFS server currently supports a maximum of only 4KB of data for a single read or write operation while secure store was run with a block size of 8KB.

6.8 Performance Benefits of Using Weaker Consistency

	latency		throughput	
	read	write	read	write
secure store without secret sharing (causal consistency)	8.25 ms	8.9 ms	32.6 MB/s	3.38 MB/s
Byzantine Quorums (safe semantics)	9.4 ms	12.3 ms	5.0 MB/s	4.8 MB/s

Table 6: Throughput and latency of systems that offer two different consistency levels: causal consistency(*secure store*) and safe semantics(recoverable Byzantine quorums [24]) for a system of 15 servers with a b of 1.

To make a fair comparison between secure store that guarantees weaker consistency

levels and other systems that guarantee stronger forms of consistency, we disabled secret-sharing in secure store and used pure replication. Thus, to write a data item, a client writes replicated copies of the data to one row of servers. Also, on-disk caching by client agent was disabled. Table 6 compares the read/write latency and throughput for three systems: (1) secure store which offers causal consistency, (2) Byzantine quorums which offer safe semantics and (3) state machine which offers atomic consistency.

The numbers presented for Byzantine quorums are for the recoverable Byzantine quorum system described in [24] and [5], with proxy and without further optimizations like using digest computation or multicast. The numbers reported in [5] have been interpolated for a system size of 15 servers. When proxies are not used, the throughput can improve by a factor of two to three. Slightly higher latency for writes is mainly due to an additional round trip communication involved in reading time stamps in the quorum protocol.

The strongest form of consistency, which is atomicity, is offered by the state machine approach. The numbers for state machine can be predicted based on Castro's thesis [11]. Castro predicts a slow down of about 1.5 for reads and 2 for writes for a system when compared to a non-replicated case in a fast LAN (1 Gbps networks with 1GHz processors in his analysis). Taking the NFS latencies for the non-replicated case, the read and write latencies of the state machine approach would be 3.2 ms and 8.4 ms respectively for a system of 7 servers with a b of 2. The thesis also predicts a throughput that is about 20% more for read operations and 30% less for write operations when compared to the non-replicated case. The read and write throughputs we measured for NFS in the emulab testbed were 6.4 MB/s and 3.0 MB/s respectively. Increase in read throughput is because of the read-only optimizations where each server returns only a digest for a read request while only one of the servers returns the actual file content. Lower latency in read operations is also due to the read-only optimizations in the system.

Read throughput in secure store is particularly high because of the parallelism possible with five rows of servers. However, the sustainable write throughput reported here is about the same as that of NFS. Latencies of both secure store and Byzantine quorums reported here are both significantly high in spite of the reduced number of servers to communicate

with. This is due to the latency involved in forwarding the file system call from the kernel to the user level client agent.

6.9 Summary

This chapter described the implementation of a file system based on *secure store*. We presented an experimental evaluation of the system demonstrating the feasibility and practicality of the approach presented in this thesis. We also evaluated the overheads of secret sharing and dissemination and showed that overheads are tolerable. Our evaluations also demonstrate the performance benefits of using weaker consistency levels. We compared our file system with NFS file system and showed that our system can be used in practice with limited number of servers and a small fault tolerance threshold. The performance degradation in using high values of b is the price paid for a higher level of security and could be useful for applications that require strong security guarantees.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis addressed the problem of building an agile distributed storage service that can guarantee integrity, availability and confidentiality of stored data even when a limited number of servers are compromised. In this final chapter, we summarize our research contributions and briefly describe some directions for future research.

7.1 Research Summary

With the considerable exposure of computer systems to outside attackers over the internet, systems are being constantly compromised. Vulnerabilities remain even in carefully built systems that are being regularly exploited to gain unauthorized access. There arises a need to build robust and secure systems that can continue to provide reliable service in spite of such compromises.

Of particular interest is protecting sensitive data that is being constantly created and exchanged. New applications in environments like home and pervasive computing infrastructures create and manipulate sensitive information that needs to be protected against accidental disclosures and active intruders. The environment and the requirements of such applications call for a secure storage service that is both highly available and secure.

This thesis addressed the problem of building a secure distributed storage service that can meet the performance and security requirements of a wide class of applications. The inherent tradeoff between security and performance implies that security always comes at a price. The agility approach advocates keeping the overhead of security as low as possible in the common case and incurring the overhead only at times of high malicious activity. Any service that strives to meet the requirements of a large class of applications needs to be agile and should offer the tradeoff between security and performance explicitly to the clients. This thesis illustrates the principles of agility by exploring the tradeoff space

between performance, security and consistency in the context of distributed storage.

A survey of the existing work in secure distributed storage shows that agility has not been one of the main design principles in the earlier systems although some ongoing projects do consider the tradeoff between security and performance. In particular, there is a strong need to integrate schemes and techniques developed for enhancing performance and those meant for security. None of the earlier approaches in secure distributed storage consider weaker consistency models that would allow far more efficient implementations and still would meet the needs of a class of applications that require data confidentiality. Also, the notion of paying the price of security only when there is some malicious activity is a new principle in agility that has not been considered by others.

We illustrate the principles of agility in a number of ways in the context of a secure distributed storage service. We designed a storage architecture that makes use of secret sharing schemes and replication techniques to guarantee availability, integrity and confidentiality of stored data in the presence of a limited number of compromised servers. By controlling the degree of replication and degree of secret sharing, *secure store* provides an explicit tradeoff between security and performance. We showed this qualitatively by assuming a probabilistic model for server compromises. The parameters that determine the performance and security could be customizable on a per-object basis, either by the client or by a separate security management service, thus making *secure store* a truly agile service.

Secure store replicates data lazily in the background using a gossip-style dissemination protocol. We addressed the data dissemination problem in *secure store* in detail. The dissemination is done in two parts: (1) secure dissemination of verification strings, and (2) dissemination of data shares. We gave a novel protocol for secure dissemination of verification strings based on a new approach called collective endorsement. The dissemination latency was shown to be comparable to best possible benign case protocol and the overhead for security was proved to be in proportion to the amount of malicious activity at the time of dissemination. We validated our claims with a set of experiments and presented our experimental evaluation. The dissemination problem is generic enough that our solution could be used in lot of other contexts like peer-to-peer environments and sensor networks.

The key allocation scheme and the endorsement technique in itself are valuable tools that can be used in other secure distributed applications.

Replication in secure store introduces the problem of consistency in data access. We revisited our target applications and observed that most applications that require high levels of integrity or confidentiality do not require strong consistency. We presented two consistency models that can meet the requirements of most target applications: (1) monotonic read consistency and (2) causal consistency. We modified the *secure store* protocols to ensure consistency in data access. The novelty in the approach is in shifting most of the responsibility in maintaining consistency to the clients and making use of *context* objects to capture consistency-related history. We summarized the potential performance benefits that can result from using weaker consistency models when compared to stronger ones like safe semantics or atomicity.

Finally, we have built a file system based on *secure store* to demonstrate the feasibility and practicality of the approach and techniques introduced in this thesis. We presented experimental results showing the tradeoff the system offers between the performance metrics, throughput and latency, and security, validating what was observed with our analytical model. The results show that the overheads involved in background data dissemination and the endorsement protocol are negligible. We also compared our system with NFS and showed that most of the increase in latency is due to secret-sharing related computation. We also compared our experimental results with other systems that offer stronger consistency to demonstrate the benefits of using weaker consistency models.

7.2 Thesis Contributions

In summary, the contributions of this thesis include:

- A distributed storage service that exploits the inherent tradeoff between security and performance and makes this tradeoff explicitly available to the client. The service combines replication and secret-sharing techniques to guarantee integrity, availability and confidentiality of data in the presence of a limited number of compromised servers. When complemented by intrusion detection, fault diagnosis and storage management

services, our service can be truly agile, the overhead of tolerating compromises being incurred only by those applications that require such high levels of security and only at times of high threat level.

- A novel gossip-style dissemination protocol to disseminate information available at a small set of servers securely to other servers in the system. The dissemination latency is comparable to best possible benign case protocol in the absence of any malicious activity and the delay in dissemination is in proportion to the amount of malicious activity. This claim has been proved theoretically and validated empirically. The protocol is based on a novel key allocation scheme and a technique called collective endorsement which can be used in other secure distributed applications. This is the first protocol to our knowledge where price paid for security is directly dependent on the amount of malicious activity.

- A technique to guarantee two kinds of weaker consistency, namely monotonic read consistency and causal consistency, in secure distributed storage services. The technique shifts most of the responsibility in maintaining consistency in data access to the clients and makes use of *context* objects for consistency-related decisions.

- An implementation of a file system based on *secure store* demonstrating the feasibility and practicality of the approach. Experimental results demonstrate the tradeoff between security and performance and also illustrate the performance benefits that can be obtained by using weaker consistency models in secure distributed storage.

7.3 Directions for Future Work

This thesis has not addressed all issues that should be considered to build a system that can be deployable in a realistic environment. For example, existing authentication and authorization mechanisms are not light-weight and hence cannot be used in an environment where sensors and cameras are the clients. New light-weight security infrastructure mechanisms have to be developed. Similarly, share renewal protocol requires using an one-way function that can be updated in the course of the protocol. Such an one-way function should

be quickly computable to make the secret-sharing scheme a practical and useful approach. Also, to be truly agile, the *secure store* service should be complemented by sophisticated intrusion detection and fault diagnosis techniques. Some of the issues are being considered by others, possibly in the context of other applications while others remain to be explored in future.

This thesis addressed only a subset of the issues related to secure distributed storage and the agility paradigm. We now highlight some of the issues that need further investigation.

7.3.1 Guaranteeing Diversity Among Servers

One of the assumptions we made in the analysis of *secure store*'s security is that server compromises are independent of each other. This implies that if an attacker successfully compromises one of the servers, the task of compromising another server is not any easier. However this assumption has not been realized in practice. Diversity among servers in terms of the operating systems and the application code (and possibly diversity in the hardware architecture) is very important to realize this assumption. Such diversity would make attacks that successfully break into multiple systems less likely, if not impossible. Creating diverse servers in an automated fashion would readily make the threshold-based Byzantine fault tolerance approach a robust and useful technique that can be deployed in practice with confidence.

N-version programming that has been proposed to address this issue has been found to be inadequate in practice [8]. There are some on-going efforts to achieve diversity among servers using other approaches. For example, one possible approach is to specialize each instance of a server by changing the order in which parameters are passed to function calls. This and other approaches need to be explored in depth, deployed and tested over time. Apart from diversity among different servers, diversity in the same server over time is also essential to limit the power of an adversary.

7.3.2 Dynamic Set of Servers

Our thesis has essentially focused on a static set of servers. However, for security, ease of deployment and other reasons, it would be useful to allow new servers to be added

and existing servers removed from the system. A robust, secure and light-weight group management scheme that keeps all nodes (clients and servers) informed of the set of servers currently in service, importing state to new servers from existing servers and a seamless security infrastructure (authentication and authorization) that supports such dynamism are some of the issues to be addressed to enable dynamic set of servers. As an additional benefit, if managing a dynamic set can be made scalable to a large number of nodes, *secure store* can be implemented as a peer-to-peer service in the internet. Such a service would be of immense use to common users in the internet for storing their personal files and documents reliably and securely.

There has been considerable work addressing secure group management [60, 42, 14]. Importing state has also been addressed in benign environments [49]. Oceanstore [3] achieves scalability by including servers in a hierarchical fashion. None of the proposed solutions are readily usable in the context of *secure store*. For example, if a malicious server is allowed to join the system twice under different identities, possibly being in two different columns for the same object, such a server would be able to collect more shares than should be allowed. Hence techniques proposed by others have to be carefully adapted and new techniques need to be developed for *secure store* to make the set of servers dynamic.

7.3.3 Applying the Agility Approach to Other Services

This thesis used the agility approach in the context of a secure distributed storage service. The agility paradigm is quite general and can be applied to other secure services and even to some services where security is not a concern. Agility paradigm can be quite useful in practice in keeping the overheads of offering specialized services low until the demand arises due to environmental conditions or application needs.

Identifying dimensions along which a service can be differentiated, the tradeoffs that exist between various metrics that characterize the service, identifying the environmental conditions that can potentially affect the quality of the service, and adapting to changing conditions are generic issues that would need to be addressed in building any agile service. A number of distributed services can be made agile. Some examples of secure services that

can use the agility paradigm are a threshold certification service, a secure directory service, web server cluster and other autonomic systems.

7.3.4 Using the Collective Endorsement Technique in Other Applications

The collective endorsement technique and the associated key allocation technique described in this thesis are contributions in themselves that can be used in other contexts. For example, the collective endorsement technique could be used to implement a threshold authorization service that is secure against a limited number of compromised servers. However, the technique described is for necessarily a small static set of servers. Scaling this technique to a large number of servers cutting across different administrative domains can lead to a viable and cheap alternative to public key based schemes for various applications. For example, Naor and others describe a scheme to replace public key signatures in multicast using a probabilistic key allocation scheme [16, 10]. Keeping the computation cost low is particularly useful in pervasive computing environments and sensor networks where computational power is limited and information about events is constantly exchanged.

7.3.5 Adaptive Secure Protocols

The gossip-style secure dissemination protocol described in this thesis pioneers a new principle that we believe should become one of the design goals for future secure systems. In the normal case when there is no malicious activity, the performance should be comparable to that of best possible benign case system and the overhead for overcoming attacks should be in proportion to the amount of malicious activity. While the agility principle in general strives to achieve this with the help of a security management module and intrusion detection and other services that monitor for malicious activity and provide feedback about current threat level, adaptive protocols achieve this without any kind of feedback mechanism and without changing any parameter that would affect the operation of the system. Such adaptive protocols allow the safety parameter to be set to a very high level without having to pay a price when there is no little or no malicious activity. In other words, a user can afford to be a paranoid if the system and the protocols are adaptive.

There are already some efforts in the agile store group to design round-based quorum

protocol that incur the overhead of tolerating compromises only when there are malicious servers. We expect that many more such adaptive protocols would be designed in future.

7.4 Conclusions

This thesis addressed the problem of building a secure distributed storage service according to the principles of agility. We described the principles of agility and presented the requirements of a secure storage service that should meet the requirements of a large class of target applications. We designed a storage scheme that combines two well known techniques, namely replication and secret sharing, to offer to the clients an explicit tradeoff between security and performance. We presented a novel gossip style protocol to replicate data lazily in the background, the dissemination latency of which is comparable to best-possible benign case protocol in the absence of any malicious activity. The overheads involved in the dissemination protocol was shown to be acceptable and the delay in dissemination latency in the presence of malicious activity was shown both theoretically and empirically to be in proportion to the amount of malicious activity. We exploited the performance benefits possible with weaker consistency models and designed a novel technique to guarantee weaker levels of consistency. We observed that such weaker levels of consistency meet the requirements of a large class of applications. Finally we presented a file system implementation of *secure store* and demonstrated the practicality and feasibility of our approach and also empirically showed the tradeoffs between performance and security and performance and consistency. We believe that this thesis would form a basis for further research in secure distributed storage and agility in secure services.

APPENDIX A

PROOFS FOR COLLECTIVE ENDORSEMENT PROTOCOL

A.1 All servers accept an update in two phases when the initial quorum size $q \geq 4b+3$.

In the following, we prove that if $p \geq q \geq 4b+3$, all servers will accept an update in two phases of MAC generation for any random choice of initial quorum of size q.

Model : Let p be a prime. In the field Z_p define a straight line $L = (\alpha, \beta)$ as the set of points (i, j) such that $i = \alpha j + \beta$. Any such line has p points. Define the universal set of lines U to be the set of all possible lines, $0 \leq \alpha \leq p-1, 0 \leq \beta \leq p-1$. Define two integer constants b and q such that $p >= q >= 4b+3$.

For any two lines L_1 and L_2, we define their intersection to be the point that is on both lines. If two lines are parallel, their α being the same, we define their intersection to be a special point at infinity along the direction of the two lines. For a line L and a set of lines S, we define the intersection of L and S as the union of points of intersection between L and every line in S.

For a set of lines S, we define $\mathcal{D}(S)$ as the set of all lines L in U such that $\mid L \cap S \mid$ is at least $2b+1$. S is contained in $\mathcal{D}(S)$.

Claim 1 : If $p >= q >= 4b+3$, for a randomly chosen set of lines Q of size q, $U = \mathcal{D}(\mathcal{D}(Q))$.

Before we prove claim 1, we will prove the following claim.

Claim 2 : Let θ be a point through which none of the lines in Q pass. Then there exists a line L passing through θ that is contained in $\mathcal{D}(Q)$.

Proof for claim 2 :

Define $\mathcal{C}$ to be a multiset of points containing all intersection points between every pair of lines in Q. Some of these intersection points may be the same, if more than two lines in Q are concurrent. We will however consider these as different elements in $\mathcal{C}$. Thus $\mathcal{C}$ has exactly $\binom{q}{2}$ points. Since θ is not on any line in Q, $\theta \notin \mathcal{C}$.

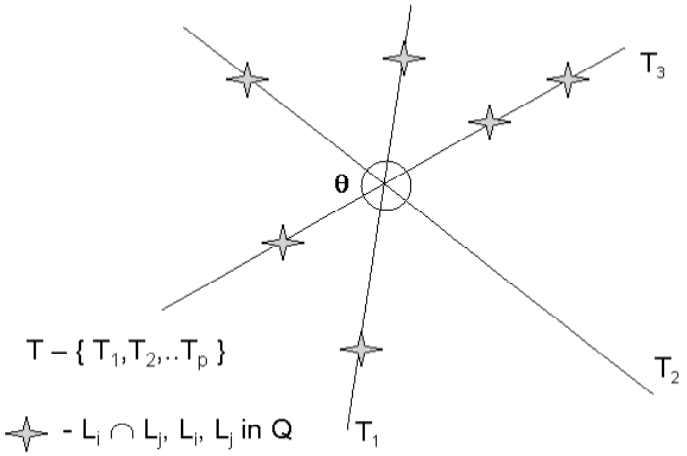


Figure 25: Distribution of $\mathcal{C}$ over lines passing through θ.

Define T to be the set of lines that pass through the point θ. Size of T is p. Since none of the lines in Q pass through θ, Q and T do not overlap.

We claim that there exists at least one line L in T such that not more than $\binom{q}{2}/p$ points in $\mathcal{C}$ are on L. This is because, even assuming the worst case that all points in $\mathcal{C}$ are on some line of T (none of them can be on more than one line in T), there are $\binom{q}{2}$ points distributed among p lines. At least one line in T should get fewer or same as the average number of points per line.

Let us get a lower bound on the number of points L shares with Q. L shares q points with Q. Some of these may be the same (concurrent points) and one of them could be the point at infinity. Concurrent points are those that belong in $\mathcal{C}$. Subtracting one from q for every concurrent point, the distinct number of points shared between L and Q should be at least $q - \binom{q}{2}/p$. This expression is at least $2b+2$ if $p >= q >= 4b+3$. Even if one point is the point at infinity, we have $2b+1$ distinct and valid points shared between L and Q. Thus L is contained in $\mathcal{D}(Q)$.

Proof for claim 1 : From claim 2, it follows that if a point does not belong to any line in Q, then it belongs to some line in $\mathcal{D}(Q)$. Thus, for any line L, the number of points of intersection between L and $\mathcal{D}(Q)$ is p which is greater than $2b+1$. This proves that every line in U belongs to $\mathcal{D}(\mathcal{D}(Q))$.

A.2 A valid MAC takes $O(log\ N) + f$ rounds to reach a constant fraction of servers

Model : Among a group of N servers, f servers are malicious. G of the N servers share a common symmetric key k, not overlapping with the f malicious servers. In round 0, one of the G servers starts with an update (that no other server knows), calculates digest(update) and computes MAC(digest,timestamp,k). In synchronous rounds of gossip, each server chooses a gossip partner for each round and pulls (1)digest of the update, (2)timestamp of the update and (3)MAC of the update. The f faulty servers always send a spurious MAC for the update to any requesting server. Servers other than the f faulty ones that do not have the key to verify the MAC ($N - G - f$ servers) will simply accept any incoming MAC

with a timestamp in the past and store it in its buffer, to be sent to other servers in future rounds of gossip. On the other hand, G servers that have the key k reject an update if either timestamp is in the future or if MAC is not valid.

Assumption : For the sake of this proof, we assume that all servers have their clocks perfectly synchronized and make their gossip at the same time. Synchrony assumption prevents the faulty servers from spreading spurious MACs for an update even before the actual source gets a chance to disseminate.

Claim : In $O(log\ N) + f$ rounds, a constant fraction of the G servers that have the key get the valid MAC that the source started with in round 0.

Proof :

We will divide the set of N servers into three groups : (1) Group A containing the G servers that know the key k, (2) Group B containing f faulty servers and (3) Group C containing the remaining $N - G - f$ servers. With some misuse of notation, we will use C to also denote the constant $N - G - f$, the number of servers in group C.

Notations :

- $b[r]$ - the number of servers in C in r^{th} round that have a spurious MAC. (b - bad)
- $l[r]$ - the number of servers in C in r^{th} round that have a valid MAC. (l - lucky)
- $g[r]$ - the number of servers in A in r^{th} round that have a valid MAC. (g - good)

Observations :

1. $g[r] \geq 1$ for all r.

2. A server in C which has a valid MAC in r^{th} round will continue to have a valid MAC in $(r+1)^{st}$ round unless the server chooses as its gossip partner for round $r+1$ a server that has a spurious MAC in that round. In the latter case, the server accepts a spurious MAC in that round.

3. A server in C which has a spurious MAC in r^{th} round will continue to have a spurious MAC in $(r+1)^{st}$ round unless the server chooses as its gossip partner for round $r+1$ a server that has a valid MAC in that round. In the latter case, the server accepts a valid MAC in that round.

Last two points can be summarized in the following equations characterizing the expected growth of $b[r]$ and $l[r]$:

$$l[r+1] = l[r] * (1 - \frac{b[r]+f}{N}) + (C - l[r]) * \frac{l[r]+g[r]}{N}. \quad (1)$$

$$b[r+1] = b[r] * (1 - \frac{l[r]+g[r]}{N}) \ + \ (C - b[r]) * \frac{b[r]+f}{N}. \quad (2)$$

We will replace $g[r]$ in equations 1 and 2 by the constant 1. Since $g[r]$ is always at least 1, this would give us a lower bound on the expected value for $l[r]$ and an upper bound on the expected value of $b[r]$.

$$l[r+1] \ = \ l[n] * (1 - \frac{b[r]+f}{N}) \ + \ (C - l[r]) * \frac{l[r]+1}{N}. \quad (3)$$

$$b[r+1] = b[r] * (1 - \frac{l[r]+1}{N}) \ + \ (C - b[r]) * \frac{b[r]+f}{N}. \quad (4)$$

From equations 3 and 4, if $l[r]/b[r] \ = \ 1/f$, it can be shown that $l[r+1]/b[r+1] \ = \ 1/f$. Since $l[0]/b[0] \ = \ 1/f$, by induction,

$$\frac{l[r]}{b[r]} \ = \ \frac{1}{f}, \ \forall r. \quad (5)$$

The system reaches a state of dynamic equilibrium when when $l[r] = C/(f+1)$ and $b[r] = f * C/(f+1)$, and $l[r]$ and $b[r]$ remain the same for all future rounds.

Let's denote by $T[r]$, the expected number of servers in round r that have some MAC, either valid or spurious. $T[r]$ is the sum of $b[r]$,$l[r]$,f and $g[r]$ which we will approximate to one. By adding equations 3 and 4 and adding $f+1$ on either side, it can be seen that

$$T[r] = T[r] * (1 + C/N - \frac{T[r]}{N}).$$

Approximating C/N to one, this equation can be rewritten as

$$T[r+1] = T[r] * (2 - \frac{T[r]}{N}). \tag{6}$$

Equation 6 is a typical equation that characterizes the traditional pull-based gossip protocol for benign settings. It can be shown that

$$1 - (\frac{T[r]}{N}) \ = \ (1 - \frac{T[0]}{N})^{2^r}.$$

Thus, after $O(log\ N)$ rounds, a constant fraction of the N servers would have some MAC or the other, 1/(f+1) of which will have valid MACs.

Let us assume a minimum of $\alpha * (1/(f+1)) * C$ servers in group C have valid MACs after $O(log\ N)$ rounds, for some constant α. Let's look at the growth of $g[r]$ from this point on.

Growth of g[n] is characterized by the following equation.

$$g[r+1] = g[r] \ + \ (G - g[r]) * (\frac{l[r] + g[r]}{N}). \tag{7}$$

Assuming $G, f << N$, we can approximate $(l[r] + g[r])/N$ to $l[r]/C$ which is $\alpha/(f+1)$. Equation 7 becomes

$$g[r+1] = g[r] \ + \ (G - g[r]) * (\frac{\alpha}{f+1}).$$

or

$$1 - (\frac{g[r+1]}{G}) \;=\; (1 - \frac{g[r]}{G}) * (1 - \frac{\alpha}{f+1}).$$

For a constant fraction β among the G servers to get the valid MAC, it takes $O(log\ N)+k$ rounds where k is given by

$$(1 - \frac{1}{G}) * (1 - \frac{\alpha}{f+1})^k = 1 - \beta.$$

Taking log on both sides and approximating $log(1 - \alpha/(f+1))$ to $-\alpha/(f+1)$, we get

$$k = \frac{f+1}{\alpha} \;*\; (log(1 - 1/G) - log(1 - \beta)). \tag{8}$$

The constant β can be adjusted to yield a constant factor of 1 for $O(f)$ on the right hand side. The β thus obtained would be sufficiently large to guarantee that all servers will accept an update in $O(log\ N) + f$ rounds. This is because each MAC does not have to be disseminated to all the servers. A server would accept an update even if it misses some MACs provided it receives $b+1$ other valid MACs. This justifies our claim that each MAC takes $O(log\ N) + f$ rounds to reach a constant fraction of servers.

REFERENCES

[1] "Emulab." http://www.emulab.net.

[2] "Farsite: Federated, available and reliable storage for an incompletely trusted environment." http://research.microsoft.com/sn/Farsite.

[3] "The oceanstore project." http://oceanstore.cs.berkeley.edu/.

[4] "The aware home research initiative." http://www.cc.gatech.edu/fce/ahri/, 2000.

[5] Ahamad, M., Blough, D. M., Fogla, P., Kong, L., Lakshmanan, S., Lee, W., Manohar, D. J., Subbiah, A., and Sun, M., "Agility in security guarantees: The agile store project." under preparation.

[6] Ahamad, M., Neiger, G., Burns, J., Hutto, P., and Kohli, P., "Causal memory: Definitions, implementations and programming," *Distributed Computing Journal*, August 1995.

[7] Alvisi, L., Malkhi, D., Pierce, E., Reiter, M., and Wright, R., "Dynamic byzantine quorum systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.

[8] Avizienis, A., "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1491–1501, December 1985.

[9] Blakley, G., "Safeguarding cryptographic keys," in *Proceedings of the National Computer Conference*, pp. 313–317, American Federation of Information Processing Societies, 1979.

[10] Canetti, R., Garay, J., Itkis, G., Micciancio, D., Naor, M., and Pinkas, B., "Multicast security: A taxonomy and some efficient constructions," in *Proceedings of INFOCOM*, (New York), March 1999.

[11] Castro, M., *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, January 2001.

[12] Chan, H., Perrig, A., and Song, D., "Random key predistribution schemes for sensor networks," in *IEEE Symposium on Security and Privacy*, May 2003.

[13] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D., "Epidemic algorithms for replicated database maintenance," in *Proceedings of the 6th Symposium on Principles of Distributed Computing*, pp. 1–12, 1987.

[14] Dutertre, B., Saidi, H., and Stavridou, V., "Intrusion-tolerant group management in enclaves," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN '01)*, (Goteborg, Sweden), pp. 203–212, July 2001.

[15] Feldman, P., "A practical scheme for non-interactive verifiable secret sharing," in *Proceedings of the 28th IEEE Symposium on the Foundations of Computer Science*, pp. 427–437, IEEE press, 1987.

[16] Fiat, A. and Naor, M., "Broadcast encryption," in *Proceedings of Crypto '93*, pp. 480–491, 1993.

[17] Fray, J., Deswarte, Y., and Powell, D., "Intrusion-tolerance using fine-grain fragmentation-scattering," in *Proceedings of the 1986 Symposium on Security and Privacy*, (Oakland, CA), April 1986.

[18] Golding, R. A., *Weak-Consistency Group Communication and Membership.* PhD thesis, University of California at Santa Cruz, December 1992.

[19] Goodson, G., Wylie, J., Ganger, G., and Reiter, M., "Decentralized storage consistency via versioning servers," Tech. Rep. CMU-CS-02-180, Carnegie Mellon University, September 2002.

[20] Herlihy, M. and Wing, J., "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages*, vol. 12, July 1992.

[21] Herlihy, M. P. and Tygar, J. D., "How to make replicated data secure," in *Proceedings of Advances in Cryptology: Crypto 87*, 1987.

[22] Herzberg, A., Jarecki, S., Krawczyk, H., and Yung, M., "Proactive secret sharing," *Advances in Cryptology, Crypto '95*, 1995.

[23] Iyengar, A., Cahn, R., Gray, J. A., and Jutla, C., "Design and implementation of a secure distributed data repository," in *Proceedings of the 14th IFIP International Information Security Conference*, September 1998.

[24] Kong, L., Subbiah, A., Ahamad, M., and Blough, D. M., "A reconfigurable byzantine quorum approach for the agile store," in *Proceedings of the International Symposium on Reliable Distributed Systems*, 2003.

[25] Krawczyk, H., "Distributed fingerprints and secure information dispersal," in *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, (Ithaca, NY), 1993.

[26] Krawczyk, H., "Secret sharing made short," in *Proceedings of Advances in Cryptology: Crypto 93*, 1993.

[27] Lakshmanan, S., Ahamad, M., and Venkateswaran, H., "A secure and highly available distributed store for meeting diverse data storage needs," Tech. Rep. GIT-CC-00-41, Georgia Institute of Technology, 2000.

[28] Lakshmanan, S., Manohar, D. J., Ahamad, M., and Venkateswaran, H., "Collective endorsement and the dissemination problem in byzantine environments," in *Proceedings of the 2004 Internation Conference on Dependable Systems and Networks*, (Florence, Italy), July 2004.

[29] Lamport, L., "Time, clocks and the ordering of events in distributed systems," *Coomunications of the ACM*, vol. 21, pp. 558–565, July 1978.

[30] LAMPORT, L., "How to make a multiprocessor computer that correctly executes multiprocessor programs," *IEEE Transactions on Computers*, vol. 28, no. 9, 1979.

[31] LAMPORT, L., "On interprocess communication," *Distributed Computing*, vol. 1, pp. 77–101, 1986.

[32] LAMPORT, L., SHOSTAK, R., and PEASE, M., "The byzantine general's problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, 1982.

[33] LANDWEHR, C. E., BULL, A. R., MCDERMOTT, J. P., and CHOI, W., "A taxonomy of computer program security flaws, with examples," *ACM Computing Surveys*, vol. 26, pp. 211–254, September 1994.

[34] LISKOV, B. and CASTRO, M., "Authenticated byzantine fault tolerance without public-key cryptography," Tech. Rep. /LCS/TM-595, Massachusetts Institute of Technology, 1999.

[35] M. CASTRO, B. L., "Practical byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans*, February 1999.

[36] MALKHI, D., MANSOUR, Y., and REITER, M., "On diffusing updates in a byzantine environment," in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, October 1999.

[37] MALKHI, D., PAVLOV, E., and SELLA, Y., "Optimal unconditional information diffusion," in *Proceedings of the 15th International Symposium on Distributed Computing*, (Lisbon, Portugal), pp. 63–67, 2001.

[38] MALKHI, D. and REITER, M., "Byzantine quorum systems," in *Proceedings of the 29th ACM Symposium on Theory of Computing*, May 1997.

[39] MALKHI, D. and REITER, M., "Secure and scalable replication in phalanx," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[40] MALKHI, D., REITER, M., RODEH, O., and SELLA, Y., "Efficient update diffusion in byzantine environments," in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, (New Orleans), pp. 90–98, 2001.

[41] MALKHI, D., REITER, M., TULONE, D., and ZISKIND, E., "Persistent objects in the fleet system," in *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, 2001.

[42] MCHUGH, J. and MICHAEL, J. B., "Secure group management in large distributed dystems: What is a group and what does it do?," in *Proceedings of the 1999 workshop on New Security Paradigms*, (Ontario, Canada), 1999.

[43] MINSKY, Y. and SCHNEIDER, F., "Tolerating malicious gossip," *Distributed Computing*, vol. 16, no. 1, pp. 49–68, 2003.

[44] NAOR, M. and WOOL, A., "Access control and signatures via quorum secret sharing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 9, 1998.

[45] Ozkasap, O., Renesse, R., Birman, K., and Xiao, Z., "Efficient buffering in reliable multicast protocols," in *Proceedings of the First International Workshop on Networked Group Communication*, (Pisa, Italy), pp. 188–203, November 1999.

[46] Pederson, T. P., "Non-interactive and information-theoretic secure verifiable secret sharing," in *Proceedings of Crypto '91*, 1991.

[47] Pennington, A., Strunk, J., Griffin, J., Soules, C., Goodson, G., and Ganger, G., "Storage-based intrusion detection: Watching storage activity for suspicious behavior," in *12th USENIX Security Symposium*, (Washington, D.C.), August 2003.

[48] Perrig, A., Szewczyk, R., Wen, V., Culler, D., and Tygar, J. D., "SPINS: Security protocols for sensor networks," in *Seventh Annual International Conference on Mobile Computing and Networks (MobiCOM 2001)*, (Rome, Italy), July 2001.

[49] Peterson, K., Spreitzer, M. J., Terry, D., Theimer, M., and Demers, A., "Flexible update propogation for weakly consistent replication," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 288–301, 1997.

[50] Rabin, M., "Efficient dispersal of information for security, load balancing and fault tolerance," *Journal of the ACM*, vol. 36, no. 2, pp. 335–348, 1989.

[51] Renesse, R., "Scalable and secure resource location," in *Proceedings of the Hawai International Conference on System Sciences*, January 2000.

[52] Rivest, R. and Lampson, B., "A simple distributed security infrastructure(sdsi)." http://theory.lcs.mit.edu/ cis/sdsi.html.

[53] Santis, A. D. and Masucci, B., "Multiple ramp schemes," *IEEE Transactions on Information Theory*, pp. 1720–1728, July 1999.

[54] Schneider, F., "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, December 1990.

[55] Shamir, A., "How to share a secret," *Communications of the ACM*, vol. 22, pp. 612–613, 1979.

[56] Terry, D., Demers, A., Peterson, K., Spreitzer, J., Theimer, M., and Welch, B., "Session guarantees for weakly consistent repicated data," in *Proceedings of the International Conference on Parallel and Distributed Information Systems*, (Austin, TX), September 1994.

[57] Tompa, M. and Woll, H., "How to share a secret with cheaters," *Journal of Cryptology*, February 1988.

[58] Wang, F., Gong, F., Sargor, C., Goseva-Popstojanova, K., Trivedi, K., and Jou, F., "Sitar: A scalable intrusion tolerance architecture for distributed services," in *IEEE 2nd SMC Information Assurance Workshop*, 2001.

[59] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A., "An integrated experimental environment for distributed systems and networks," in *Operating Systems Design and Implementation*, December 2002.

[60] Wong, C. K., Gouda, M., and Lam, S., "Secure group communication using key graphs," in *Proceedings of ACM SIGCOMM*, (Vancouver, British Columbia), September 1998.

[61] Woo, T. Y. C. and Lam, S. S., "A framework for distributed authorization," in *Proceedings of the ACM Conference on Computer and Communications Security*, (Fairfax, VA), November 1993.

[62] Wylie, J. J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Kiliccote, H., and Khosla, P. K., "Survivable information storage systems," *IEEE Computer*, vol. 33, pp. 61–68, August 2000.

[63] Yu, H. and Vahdat, A., "Design and evaluation of a continuous consistency model for replicated services," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, October 2000.

[64] Zhou, L., Schneider, F. B., and Van Renesse, R., "Coca: A secure distributed online certification authority," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 329–368, 2002.