Building Trustworthy Storage Services out of Untrusted Infrastructure

Jinyuan Li

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

September 2006

Prof. David Mazières

To my parents Shuzhong Li, Yulan Zhou

and my sister Jinyang Li

Acknowledgments

I am greatly indebted to Prof. David Mazières for his supervision throughout my five years in graduate school. Without his steady encouragement and insightful inspiration, I would never have been able to complete this work. Also, his enthusiastic approach to life has deeply impressed and transformed everyone around.

I would like to thank Prof. Vijay Karamcheti, Prof. Lakshminarayanan Subramanian, Prof. Dawson Engler and Prof. Mendel Rosenblum for their suggestive comments to the work and for kindly serving on the committee. I have also learned a lot from Prof. Dennis Shasha, when working with him on SUNDR.

There are few other places more exciting to work in other than our Secure Computer Systems Group. Without Maxwell Krohn's high performance *bstor*, SUNDR would be a less convincing story; Siddhartha Annapureddy is now interested in delivering jitterless on-demand video to large numbers of users; Michael Freedman develops and manages Coral, a CDN used world-wide; Antonio Nicolosi is always keen to answer our cryptography questions.

I am also grateful to my friends for making my life in New York City and California more enjoyable – Zilin Du, Zhihua Wang, Junfeng Yang, Feng Qin, Lu Zhang, Yuanxing Dong, Peng Zhang, Ziyang Wang, Xiaojian Zhao, Feng Ning, Yuxing Yang, Chie Mishiro, Min Luo, Xin Pu, Chingching Shih, Ya-Yunn Su, Ying Xu, Nina Liu and Xiang Jin.

Abstract

As the Internet has become increasingly ubiquitous, it has seen tremendous growth in the popularity of online services. These services range from online CVS repositories like *sourceforge.net*, shopping sites, to online financial and administrative systems, etc. It is critical for these services to provide correct and reliable execution for clients. However, given their attractiveness as targets and ubiquitous accessibility, online servers also have a significant chance of being compromised, leading to Byzantine failures.

Designing and implementing a service to run on a machine that may be compromised is not an easy task, since infrastructure under malicious control may behave arbitrarily. Even worse, as any monitoring facility may also be subverted at the same time, there is no easy way for system behavior to be audited, or for malicious attacks to be detected.

We propose our solution to the problem by reducing the trust needed on the server side in the first place. In the other words, our system is designed specifically for running on untrusted hosts. In this thesis, we realize this principle by two different approaches. First, we design and implement a new network file system – SUNDR. In SUNDR, malicious servers cannot forge users' operations or tamper with their data without being detected. In the worst case, attackers can only conceal users' operations from each other. Still, SUNDR is able to detect this misbehavior whenever users communicate with each other directly.

The limitation of the approach above lies in that the system cannot guarantee ideal

consistency with even one single failure. In the second approach, we use replicated state machines to tolerate some fraction of malicious server failures, which is termed Byzantine Fault Tolerance (BFT) in the literature. Classical BFT systems assume less than $1/3$ of the replicas are malicious, to provide ideal consistency. In this thesis, we push the boundary from $1/3$ to $2/3$. With fewer than $1/3$ of replicas faulty, we provide the same guarantees as classical BFT systems. Additionally, we guarantee weaker consistency, instead of arbitrary behavior, when between $1/3$ and $2/3$ of replicas fail.

Contents

Acknowledgments **iv**

Abstract **v**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Motivations 1

1.2 Organization 6

2 Fork consistency and Fork* consistency **7**

2.1 Fetch-modify consistency 8

2.2 Fork consistency 10

2.2.1 Limitations of fork consistency 14

2.2.2 Impossibility Proof 14

2.3 A two-round protocol 16

2.4 Fork* consistency 18

3 SUNDR **20**

3.1 Setting . . . 21

3.2 The SUNDR protocol . . . 23

3.2.1 A straw-man file system . . . 24

3.2.2 Implications of fork consistency . . . 26

3.2.3 Serialized SUNDR . . . 27

3.2.3.1 Data structures . . . 27

3.2.3.2 Protocol . . . 29

3.2.4 Concurrent SUNDR . . . 32

3.2.4.1 Update certificates . . . 32

3.2.4.2 Update conflicts . . . 34

3.2.4.3 Example . . . 36

3.3 Discussion . . . 37

3.4 File system implementation . . . 39

3.4.1 File system client . . . 40

3.4.2 Signature optimization . . . 41

3.4.3 Consistency server . . . 42

3.5 Block store implementation . . . 42

3.5.1 Interface . . . 43

3.5.2 Index . . . 44

3.5.3 Data management . . . 45

3.6 Performance . . . 46

3.6.1 Experimental setup . . . 46

3.6.2 Microbenchmarks . . . 47

3.6.2.1 *bstor* . . . 47

3.6.2.2 Cryptographic overhead . . . 49
3.6.3 End-to-end evaluation . . . 49
3.6.3.1 LFS small file benchmark . . . 50
3.6.3.2 Group contention . . . 52
3.6.3.3 Real workloads . . . 53
3.6.3.4 CVS on SUNDR . . . 54

4 BFT2F **56**

4.1 Background . . . 58
4.1.1 PBFT in a nutshell . . . 59
4.2 BFT2F Algorithm . . . 60
4.2.1 BFT2F Variables . . . 61
4.2.2 BFT2F Node Behavior . . . 62
4.2.3 Garbage Collection . . . 64
4.2.4 Server View Change . . . 65
4.2.5 An Example . . . 67
4.3 Proof Sketch . . . 69
4.3.1 Normal case operations . . . 69
4.3.2 Servers view change . . . 72
4.4 Discussion . . . 74
4.5 Performance . . . 75
4.5.1 Implementation . . . 75
4.5.2 Micro benchmark . . . 75
4.5.3 Application-level benchmark . . . 76

5 Related Work **77**

5.1 Cryptographic file systems 77

5.2 Byzantine Fault Tolerant systems 80

5.3 Encapsulation of hostile behavior 81

5.4 Anonymity systems 82

6 Conclusion **84**

6.1 Future work 84

6.2 Conclusion 85

References **88**

List of Figures

1.1 An example of an integrity attack. Suppose a client first stores a file F to the server. Later on when the client fetches the file, the server returns a modified one. 2

1.2 An example of a freshness attack. Suppose a client first stores a file F^1 to the server, and updates the same file to a new version F^2 afterwards. Later on when the client fetches the file, the server returns with the old version of the file F^1. 3

2.1 An example of fork consistency. Since the server deceives client b about a's operation op^2, both client a's result list $\langle op^1, op^2 \rangle$, and b's result list $\langle op^1, op^3 \rangle$ are fork consistent only (Strictly speaking, client a still has fetch-modify consistency at this moment, should op^2 be ordered before op^3. However, client a will miss fetch-modify consistency hereafter.). 12

2.2 In a one-server system, the formation of two forked sets implies the server has been malicious. In a replicated state system, the intersection of two fork sets can only consists of *provable malicious* servers, while the partition that excludes the intersection part might have honest, and "malicious", but not yet misbehaving servers, which we call *probable malicious* servers. 13

2.3 Two malicious servers (p and q) wear different hats when talking to distinct honest servers (u or w). In this way, p and q, with u, return result list $\langle op_b, op_a \rangle$ to client a; p and q, with w, return $\langle op_a, op_b \rangle$ to client b. 15
2.4 A two-round protocol. 16
2.5 Pseudocode for a two-round protocol. 17

3.1 Basic SUNDR architecture. 21
3.2 User and group i-handles. An *i-handle* is the root of a hash tree containing a user or group *i-table*. (H denotes SHA-1, while H^* denotes recursive application of SHA-1 to compute the root of a hash tree.) A *group i-table* maps group inode numbers to user inode numbers. A *user i-table* maps a user's inode numbers to i-hashes. An *i-hash* is the hash of an inode, which in turn contains hashes of file data blocks. 28
3.3 A version structure containing a group i-handle. 29
3.4 Signed version structures with a forking attack. 31
3.5 i-table for group g, showing the change log. t'_g is a recent i-table; applying the log to t'_g yields t_g. 32
3.6 Concurrent updates to `/sundr/tmp/` by different users. 35
3.7 A pending update by user u_1, reflected in user u_2's version structure. 37
3.8 *bstor* throughput measurements with the block cache disabled. 48
3.9 Single client LFS Small File Benchmark. 1000 operations on files with 1 KB of random content. 50
3.10 Concurrent LFS Small File Benchmark, **create** phase. 1000 creations of 1 KB files. (Relative standard deviation for SUNDR in 3 concurrent clients case is 13.7%) 52

3.11 Installation procedure for `emacs_20.7` 53
3.12 Concurrent *untar* of `emacs_20.7.tar` 54

4.1 An example of two forked result lists. The timeline in the middle shows the result list that would have been executed by a non-faulty (never-forked) system. The timeline above it shows in a forked system, one fork set has executed a forked result list, which does not reflect operation op'^2. The timeline below it shows another forked result list that omits operation op^2. . 67
4.2 An example of join-at-most-once property. Suppose op^3 has been used to join two forked result lists as in Figure 4.1. Diagram (a) shows the commit and reply stages for operation op^3. Since the result lists of the two fork sets are already forked, the HCD field of sequence number 3 for non-faulty replicas in different fork sets is different: HCD^3 for u and HCD'^3 for w respectively. Client a accepts the reply from fork set FS_α, and updates its V_a accordingly. Notice a cannot receive the reply from (non-faulty) replicas (e.g., w) in FS_β simultaneously, without w being able to detect the system faulty. Diagram (b) shows the impossibility of any future operation by c appearing in both fork sets' result lists again. For example, if the next operation op^4 reaches non-faulty replicas in both fork sets, then the HCD check in pre-prepare stage could only succeed at one of them. 68
4.3 From section 4.3.1, either SL_v or SL_{v+1} delivers fork* consistency, respectively, within its own view. We only need to show, when considering completed operations from both sub result lists, it also preserves fork* consistency. 72

List of Tables

3.1 Run times for CVS experiments (in seconds). 55

4.1 Performance comparison of different file system implementation (in seconds). 76

Chapter 1

Introduction

1.1 Motivations

People rely on online storage systems to save and share various types of critical data nowadays. For example, millions of users use Yahoo's online email service [6] for their daily email communications. Many people publish Blogs to share personal journals and solicit comments from friends and others. Amazon recently launched S3 [1] that lets users store, retrieve, and share their personal data conveniently via a web-based interface.

A well-behaved storage service guarantees the integrity and freshness (also called consistency) of the data it stores on behalf of its clients. Despite their popularity and importance, it is very difficult for today's online storage systems to provide users any security guarantees. This is because in the current usage model for storage systems, clients trust the storage servers completely to faithfully store data. This trust model is fundamentally flawed because it is hard to secure a server in practice. First, a dependable server requires a trustworthy administrator who does not tamper with stored data. Second, the server requires a competent administrator who promptly patches known security holes as they are

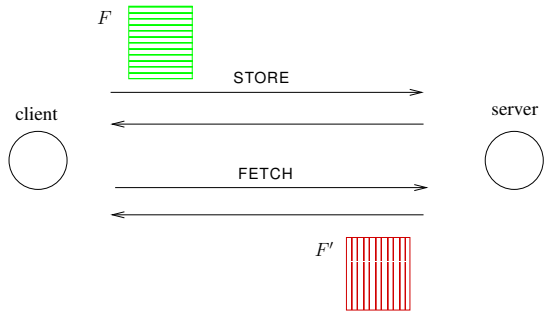


Figure 1.1: An example of an integrity attack. Suppose a client first stores a file F to the server. Later on when the client fetches the file, the server returns a modified one.

discovered. Both of these requirements are hard to meet in the real world. Being widely accessible, the server in an online storage system is much more likely to become the victim of an attack, which leads to arbitrary failures. Furthermore, given their importance in serving possibly millions of users, online storage services also become attractive targets of attacks. Consequently, a compromised server results in the total loss of any guarantee of the entire online storage service.

Because servers are trusted in the current storage system model, they lack the ability to attest to clients whose data they have received and stored. Hence, compromised storage servers can return arbitrary result to clients, because they are not expected to present such proofs for data integrity and freshness anyway. Figure 1.1 and Figure 1.2 show two example attacks that a compromised server can carry out on trusting clients. In Figure 1.1, a malicious server returns to the client a modified version of the file which is different from what has been stored previously. In Figure 1.2, a malicious server returns to the client a valid, but outdated version of the file. In both examples, since the client completely trusts the server, there is no mechanism for the server to demonstrate to the client the correctness of the data it provides. As a result, a compromised server can cause the client to accept arbitrary data.

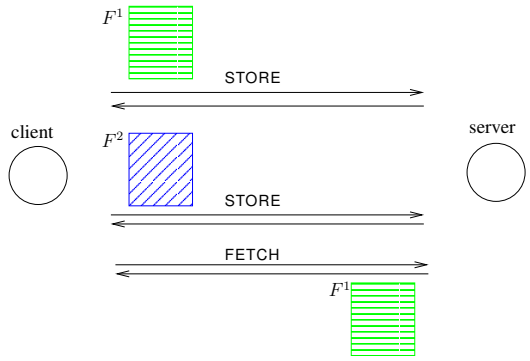


Figure 1.2: An example of a freshness attack. Suppose a client first stores a file F^1 to the server, and updates the same file to a new version F^2 afterwards. Later on when the client fetches the file, the server returns with the old version of the file F^1.

These concerns are no mere academic exercise, as we have seen server compromises due to malicious system administrators or software vulnerabilities. Recently, a former UBS/ PaineWebber network administrator was convicted for deleting files on over 1,000 machines, and this caused about a $3M loss to the company [10]. Furthermore, according to an FBI statistic [9], discontented employees and former employees account for up to 65% of such security breaches.

Even with trustworthy administrators, systems might still fail. For example, the Debian GNU/Linux development cluster was compromised in November 2003 [11]. An unauthorized attacker used a sniffed password and a kernel vulnerability to gain superuser access to Debian's primary CVS and Web servers. After detecting the break-in, administrators were forced to freeze development for several days, as they employed manual and ad-hoc sanity checks to assess the extent of the damage. Finally, the bug was fixed, and the kernel was patched up. Unfortunately, this does not prevent Debian CVS from suffering the same fate again: it was broken into in July 2006 [13], and the reason still remains unidentified at this moment. Similar attacks have also succeeded in the past against Apache [8], Gnome [12], and other popular projects.

Currently, there lack systematic solutions to building secure storage systems. Most existing work keeps with the current trust model, but aims to secure the server better by using firewalls, intrusion detection systems, and other defensive approaches. This approach has two main drawbacks. First, experience shows that people usually do not build high enough fences (or sometimes entrust fences to administrators who are not completely trustworthy). Second and more importantly, higher fences are inconvenient: they restrict the ways in which people can access, update, and manage data.

In this thesis, we take a new direction; instead of requiring clients to trust the servers, we propose ways to construct secure storage systems out of untrusted servers. Unlike traditional systems, our design requires untrusted servers to explicitly demonstrate to clients the integrity and freshness of the data they provide.

When all servers fail arbitrarily, it is impossible to guarantee data freshness. However, this does not mean that the overall system should necessarily behave arbitrarily. For example, in Figure 1.2, if the client is required to remember the latest version of the data block it has previously provided to the server, it will be able to detect this particular freshness attack by the server immediately. The focus of our work is to constrain the implications of servers' (mis)behavior, and to extract the strongest possible weak guarantees for a storage system built out of untrusted servers. Such weak guarantees are useful because they enable clients to audit and prove the misbehavior of servers and detect server compromises.

In this thesis, we formalize two new weak consistency models – fork consistency and fork* consistency – which are achievable even when all or a majority of servers are malicious. In both models, malicious servers can cause clients to accept an older version of the data. However, once servers misbehave, evidence of the attack cannot be subsequently erased. This feature makes fork consistency useful in detecting any past consistency violations by the servers.

We first demonstrate our ideas in the context of a distributed file system by designing and building SUNDR (Secure UNtrusted Data Repository). SUNDR provably achieves fork consistency in the case of one unreplicated server. SUNDR does not require any trusted component on the server side. Clients make requests to the server and can detect any consistency or integrity failures injected by a compromised server, as long as they see other peers' latest operations.

Unfortunately, fork consistency requires heavy protocols that need at least two rounds of communication between a client and server for each operation. The extra round of communication leads to higher latency, increased message overhead and additional server state. Furthermore, it complicates the system design because we need to handle the case when the client fails prematurely after finishing just the first round. To simplify the design, we propose a slightly weaker consistency model, fork* consistency, that can be realized with just one round of communication for each request.

We then design and build BFT2F, a distributed file system in the context of multiple replicated servers. BFT2F takes the same replicated state machine approach as used by the classical Castro-Liskov Practical Byzantine Fault Tolerance model (PBFT) [25]. Instead of relying on one single honest or compromised server, PBFT replicates servers and assumes more than two thirds of the replicas honest, to achieve ideal consistency [21]. However, when this requirement is not met, PBFT guarantees nothing. Building from PBFT, BFT2F further reduces the trust needed in servers by exploiting the design space beyond this assumption. Specifically, BFT2F provides fork* consistency when one third or more, but fewer than two thirds of the replicas fail. Though BFT2F cannot guarantee liveness in this case, it is still preferable to arbitrary behavior in most situations.

This thesis develops two practical secure storage systems: SUNDR and BFT2F. In short, they share the principle of *"eliminating, or reducing the need to trust storage servers"*.

We demonstrate how to design client-server protocols such that clients are aware in the first place that servers might be malicious, or potentially corrupted. They also can reveal any past violation of the expected storage semantics. We implemented and evaluated both systems. Measurements show performance is only slightly worse than comparable, but less secure systems. Yet by reducing the amount of trust placed in the server, both systems increase people's options for managing data and significantly improve the security of their files.

1.2 Organization

The thesis is organized as follows: First, we formalize fork consistency and fork* consistency, and discuss their difference in detail in chapter 2. In particular, we prove the impossibility of achieving fork consistency with one round of communication between clients and servers for each request. We then describe SUNDR, which achieves fork consistency, in chapter 3. In chapter 4, we propose BFT2F, an extension to PBFT. BFT2F guarantees fork* consistency with more than f, but no more than $2f$ faulty replicas out of a $3f + 1$ replicas system. We give a brief survey of related work in chapter 5, discuss future work and conclude in chapter 6.

Chapter 2

Fork consistency and Fork* consistency

It is not always possible to achieve ideal consistency in the presence of arbitrary server failures. For example, if the only server in a non-replicated online storage system is compromised, there is no way of preventing it from concealing one client's newly updated file from other clients' subsequent fetch requests of the same file because clients do not always communicate among each other directly. However, ideally, the server should only be allowed to misbehave in a certain way, and the evidence of such attacks should not be erased. We define a weaker consistency model, fork consistency, to capture this desired property; the new weak consistency models make it as easy as possible to detect whether there has been any consistency failure in the past, and to limit servers only to consistency violations, as opposed to arbitrary attacks.

2.1 Fetch-modify consistency[1]

We first discuss the ideal consistency guarantees of a storage system. In the distributed file system literature, *close-to-open consistency* [45] is widely accepted to define the consistency of related operations. Loosely speaking, a newly "opened" file should reflect all modifications by other clients that previously "closed" the file. The advantage of close-to-open consistency is that a client can read from local cache, or delay writing to the server when the file is opened, but not yet closed. Unfortunately, not all file system operations incur file opens or closes: for example, the truncate system call modifies a file synchronously, without opening or closing it, and the effect of file truncate should be immediately visible to other users. Also, storage systems besides file systems often do not have the notion file open and close. Consequently, to be more generally applicable, we will speak of *fetch* and *modify* operations in this thesis, rather than opens and closes. A fetch operation is when a client either validates its local cached copy of a file or downloads a new version from the server. A modify operation is when a client makes new file contents visible to other clients. Higher-level filesystem calls can be translated to fetch and modify operations accordingly.

Definition 1 Each request has two wall-clock times associated with it – *issue time* and *completion time*.

Conceptually, the issue time corresponds to the time at which the fetch or modify operation is invoked, which would return at some later time, called completion time. We do not assume clients have synchronized clocks, so a client will not know the issue or completion time of its own operation.

[1] This and the next section borrows heavily from the definition given by Mazières and Shasha in [60].

Definition 2 A set of fetch and modify operations in a system is *orderable*, if there is a partial order, happens before ($\prec$), on the operations such that:

1. If the completion time of operation op^1 is earlier than the issue time of operation op^2, then op^1 happens before op^2.
2. Happens before orders any two operations by the same client.
3. Happens before totally orders all modifications to a given file.
4. Happens before orders any fetch of a file with respect to all modifications to the same file.

Orderability does not fully dictate the order of operations. For concurrent, overlapping operations, the system is free to choose any order, as long as all dependent operations have a definite order.

Definition 3 A set of fetch and modify operations is *fetch-modify consistent* iff the operations are orderable and any fetch f of a file F returns the contents of the file produced by exactly the modifications that happened before f, in the order specified by the happens before relation.

In any system, the result of each operation should be determined by a list of previously executed operations. We call these lists *result lists*. In a fetch-modify consistent (well-behaved) system, A *well-formed* result list is an ordered list of *all* previous operations: $L^{all} = \langle op^1, op^2, ..., op^n \rangle$, and L^{all} should be fetch-modify consistent. While in a faulty system, the result list for an operation might not include all the operations that happen before it, because the malicious server might conceal it. Even worse, the malicious server could return to distinct clients different result lists for the same operation. Nevertheless, we

assume malicious servers can not make up operations that clients have never issued This can be implemented by having clients sign their issued operations with their private keys.

Fetch-modify consistency is stronger than one-copy serializability [18], in the sense that two externally non-overlapping operations must be temporally ordered. The notion of fetch-modify consistency is not completely new, it is equivalent to linearizability for concurrent objects [44], and 1SR+EXT in TACT [85]. In this thesis, we will also use the terms *fetch-modify consistency* and *ideal consistency* interchangeably.

2.2 Fork consistency

Fetch-modify consistency may be violated when servers become malicious. However, it is still possible to achieve a weaker consistency that allows clients to detect any past consistency violation conducted by the server.

In defining fork consistency, we make a distinction between operations issued by *correct* clients, which obey the protocol, and *malicious* clients that don't. We use the term result lists only for replies to operations issued by correct clients, as it makes little sense to talk about results seen by clients that do not implement the protocol.

Definition 4 A system is *fork consistent* if and only if it satisfies the following requirements:

- **Sublist Property:** At any given time, every result list L is a sublist of one specific well-formed result list L^{all}. [2]

  Since L is a sublist of L^{all}, this implies operations in L should have the same ordering relations as those in L^{all}, which should preserve the temporal order of non-

[2] For example, both $\langle op^1, op^3 \rangle$, and $\langle op^1, op^2 \rangle$ are sublists of $\langle op^1, op^2, op^3 \rangle$, but $\langle op^3, op^2 \rangle$ is not.

overlapping operations by honest clients,

- **Self-consistency Property:** Each honest client sees all past operations from itself: if the same client a issues an operation op^i before op^j, then op^i also appears in op^j's result list.

  This property ensures that a client has a consistent view w.r.t. its own operations. For example, in a file system, a client always sees the effect of its own writes.

- **No-join Property:** Every result list that contains an operation op by an honest client is identical up to op.

  As a corollary, the no-join property implies that when two clients see each others' latest operation, they have also seen *all* of each other's past operations in the same order. This makes it possible to audit the system; if we can collect all clients' current knowledge and check that all have seen each others' latest operations, we can be sure that the system has *never* violated fetch-modify consistency in the past.

We say two result lists are *forked* if neither one is an (improper) prefix of the other. For instance, result list $\langle op^1, op^2 \rangle$ and result list $\langle op^1, op^3 \rangle$ are forked, while $\langle op^1, op^2 \rangle$ and $\langle op^1, op^2, op^3 \rangle$ are not. Informally speaking, a forked result list represents a failure of the server to deliver ideal consistency. In the example above, forked result lists $\langle op^1, op^2 \rangle$ and $\langle op^1, op^3 \rangle$ indicate the server's failure to deliver ideal consistency immediately after op^1.

If the system consists of multiple servers which replicate the system state, it is also useful to categorize servers by the responses they return. We say servers are in different *fork sets* if they reflect forked result lists. A fork set consists of a set of servers who return the *same* result lists to the client, although, in an asynchronous system, some of them might not reach the client before the operation completes. Intuitively, any correct server cannot

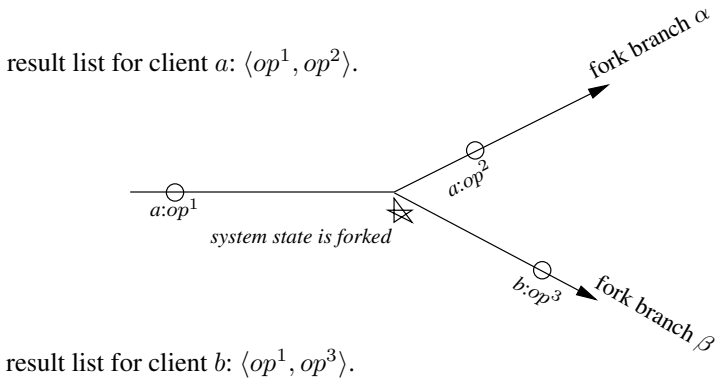


Figure 2.1: An example of fork consistency. Since the server deceives client b about a's operation op^2, both client a's result list $\langle op^1, op^2 \rangle$, and b's result list $\langle op^1, op^3 \rangle$ are fork consistent only (Strictly speaking, client a still has fetch-modify consistency at this moment, should op^2 be ordered before op^3. However, client a will miss fetch-modify consistency hereafter.).

be in more than one fork set. A malicious server, however, can simultaneously be in multiple fork sets – presenting different system states to different clients, and might even return results that do not reflect any well-formed result-list. To achieve ideal consistency, all correct servers must be in the same fork set, and execute the protocol faithfully. When correct servers are forked into different fork sets, clients can no longer achieve ideal consistency and may accept results from different fork sets.

Figure 2.1 shows an example where fork consistency differs from ideal consistency. Initially, all clients have the same result list $\langle op^1 \rangle$. Client a issues a request op^2, and gets back the result list $\langle op^1, op^2 \rangle$ from fork set FS_α; after that, client b's operation op^3 returns with the result list $\langle op^1, op^3 \rangle$ from fork set FS_β. This means the server deceives b about a's completed operation op^2. Therefore, client a's result list and client b's are forked. At this moment, one can verify that the system has delivered fork consistency, but failed to provide fetch-modify consistency.

Figure 2.2 shows an oracle's view of the system at this time. We mean *oracle* by an

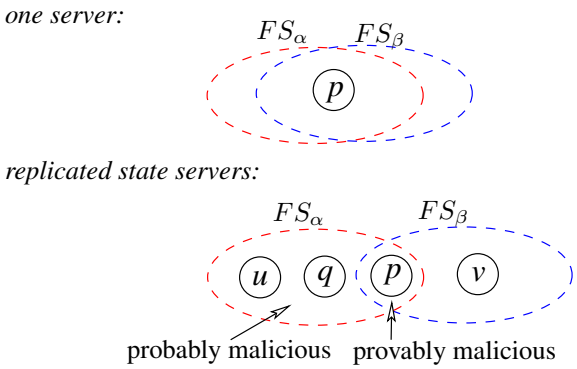


Figure 2.2: In a one-server system, the formation of two forked sets implies the server has been malicious. In a replicated state system, the intersection of two fork sets can only consists of *provable malicious* servers, while the partition that excludes the intersection part might have honest, and "malicious", but not yet misbehaving servers, which we call *probable malicious* servers.

external entity who can globally observe the current system state. However, in a real system, such knowledge usually cannot be collected. If the system only has one server p, which implies $FS_\alpha = FS_\beta = \{p\}$, then p must be malicious. If the system consists of multiple servers as replicated state machines, then $FS_\alpha \bigcap FS_\beta$ [3] must have *only* malicious servers that have attacked (e.g., p). $FS_\alpha \backslash (FS_\alpha \bigcap FS_\beta)$ [4] and $FS_\beta \backslash (FS_\alpha \bigcap FS_\beta)$ could include either honest servers (e.g., u, v), or malicious servers who have not misbehaved so far (e.g., q), i.e., indistinguishable from honest servers, but might attack later. However, once they do so, they will be classified in $FS_\alpha \bigcap FS_\beta$, as observed by the oracle, and can not move back again.

[3] $FS_\alpha \bigcap FS_\beta$ means the intersection of FS_α and FS_β.
[4] $FS_\alpha \backslash (FS_\alpha \bigcap FS_\beta)$ means the partition of FS_α which excludes $FS_\alpha \bigcap FS_\beta$.

2.2.1 Limitations of fork consistency

A practical consistency protocol requires that both the number of messages sent and the latency of each operation be small. To complete an operation, a client issues requests to servers and waits for the corresponding replies. In a one-round protocol, message exchange between a client and the servers happens exactly once for every operation. The number of rounds in a protocol affects its performance; each additional round results in increased latency and extra messages sent between clients and servers. Additionally, a protocol of more than one round needs to keep extra state about clients, which results in extra complexity. Finally, a client that fails prematurely after just the first round of the protocol poses significant difficulty to the system, since other clients who depend its operation completing would be blocked indefinitely. [5] Consequently, most consistency protocols designed for replicated state systems are one-round protocols, such as PBFT [25].

Unfortunately, in a system without sufficient trust on the server side, it is impossible to achieve fork consistency with one round of communication for each operation. For instance, when more than f servers fail in an asynchronous BFT system of $3f + 1$ servers, at least two rounds of communication are required to achieve fork consistency. We prove this below.

2.2.2 Impossibility Proof

Theorem: When more than f out of $3f + 1$ servers fail in an asynchronous system, no one-round protocol can achieve fork consistency.

Proof Sketch: In a single-round protocol, an operation sends a single message which, finally convinces honest replicas to alter their state by executing an operation. Consider the

[5] Further discussion on this can be found in Section 3.2.4.

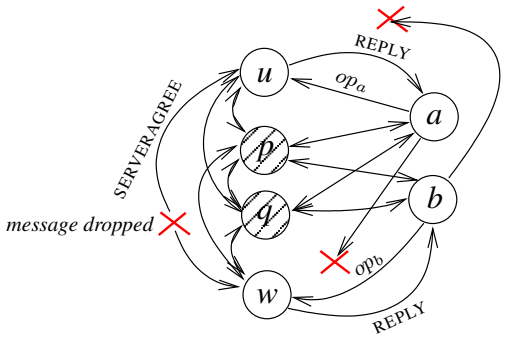


Figure 2.3: Two malicious servers (p and q) wear different hats when talking to distinct honest servers (u or w). In this way, p and q, with u, return result list $\langle op_b, op_a \rangle$ to client a; p and q, with w, return $\langle op_a, op_b \rangle$ to client b.

case when two clients, a and b, issue two requests, op_a and op_b concurrently. Neither client could have known about the other's request when issuing its operation. Thus, either op_a or op_b is capable of being executed before the other.

If, for instance, the network delays op_a, op_b could have been executed before op_a arrives, and vice versa. Moreover, because of liveness, request op_a must be capable of executing if both client b and replica w are unreachable, as long as the remaining three replicas respond to the protocol. Figure 2.3 illustrates this case, where client a was eventually returned with result list $\langle op_b, op_a \rangle$.

The same reasoning also applies to client b, who might get result list $\langle op_a, op_b \rangle$, when replica u is unreachable. These two out-of-order result lists reflect the (malicious) servers' ability to reorder concurrent requests at will, if a single message is allowed to change the system state *atomically*. This clearly violates no-join property of fork consistency. ∎

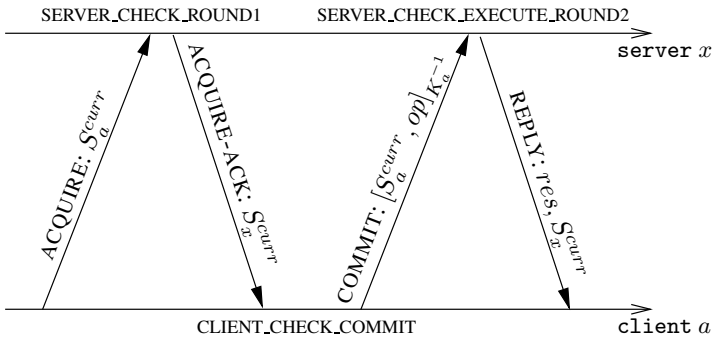


Figure 2.4: A two-round protocol.

2.3 A two-round protocol

We proceed to explain why the above contradiction does not arise with a two-round protocol. Figure 2.4 shows the generic structure of a two-round protocol; a client sends an `acquire` request to acquire the system's latest state in the first round, and a server replies to the client with its current state. In the second round the client commits the request op, servers execute op and send back the reply. For simplicity of discussion, we assume clients' two rounds of accesses to be atomic, i.e., servers would queue other concurrent requests until they finish the one in progress. Figure 2.5 shows how server x generates the `acquire-ack` message in round one and how client a generates a corresponding `commit` message upon the receipt of `acquire-ack`. S_n^{curr} represents node n's latest knowledge of the system state.

Here, S^{curr} could be realized by a log of all previous requests that have led to the current state. Each log entry has the format of $[\texttt{commit}, S_a^{curr}, op]_{K_a{}^{-1}}$, signed by client a's private key ${K_a}^{-1}$. S_a^{curr} is client a's knowledge of the system state before a commits message op. Since a updates S_a^{curr} in the first round of communication with servers, S_a^{curr} is guaranteed to be up-to-date.

//executed by server x in the first round upon receiving `acquire`.
procedure SERVER_CHECK_ROUND1(S_a^{curr})
if $S_a^{curr} \prec S_x^{curr}$
send [`acquire-ack`, S_x^{curr}] to client a;
else
//either a is faulty, or x has been forked

//executed by client a to collect `acquire-acks` and generate `commit`.
procedure CLIENT_CHECK_COMMIT(op)
decide on the system's current state S^{curr} based on $2f + 1$
matching `acquire-acks` from servers
$S_a^{curr} \leftarrow S^{curr}$
sends $[\texttt{commit}, S_a^{curr}, op]_{K_a{}^{-1}}$ to all servers

//executed by server x in the second round upon receiving `commit`.
procedure SERVER_CHECK_EXECUTE_ROUND2(op, S_a^{curr})

(S_x^{curr}, op) $\leftarrow$ SERVERAGREE(S_x^{curr}, op)

if $S_a^{curr} = S_x^{curr}$
//x's knowledge is the same as a's, execute op, update state
(S_x^{curr}, res) $\leftarrow$ EXECUTE(S_x^{curr}, op)
send [`reply`, S_x^{curr}, res] to client a;
else
//either a is faulty, or x has been forked

Figure 2.5: Pseudocode for a two-round protocol.

In the example above: if, for client b, op_a is ordered before op_b by the servers, op_a's corresponding log entry would be $[\texttt{commit}, S^0, op_a]_{K_a{}^{-1}}$. S^0 is the system state before client a and client b issue their operations. However, if op_b is ordered before op_a in client a's view, op_a's log entry would be $[\texttt{commit}, S^1, op_a]_{K_a{}^{-1}}$, where S^1 is the system state after applying op_b to initial state S^0. An honest client would not sign different log entries for the same

operation. Consequently, so long as it is possible to determine the relative ordering of two states like S^0 and S^1 by examining them, even malicious servers cannot order op_a and op_b differently for two clients.

As should be clear from the above discussion, the benefit of a two-round protocol is the ability to improve clients' knowledge of the system state in the first round, and thus to prevent malicious servers from deceiving oblivious clients. If the system state is ever forked before a (non-faulty) client makes its request, malicious servers can present at most *one* current state to the client, so the client's operation cannot be reflected in more than one fork branch.

2.4 Fork* consistency

Since a two-round protocol is expensive and complicated, we propose a slightly weaker consistency model, fork* consistency, which can be achieved with a one-round protocol. Essentially, fork* consistency relaxes the no-join property in fork consistency to join-at-most-once, i.e., two forked result lists can be joined by at most one operation from the *same* correct client.

Join-at-most-once Property: If two result lists contain two operations op' and op from the same client a, and op' precedes op, then the two result lists are identical up to op'.

As shown in the previous section, with a one-round protocol, client a's operation op might be applied to servers in both fork sets, if a has no clue that the system state has been forked when she issues op. Then the servers in FS_α or FS_β would update their system state S_α or S_β respectively, which violates the no-join property of fork consistency. However, client a is only going to accept the reply from one of the fork sets, e.g., FS_α, and adopt

its new state S_α, as her freshest knowledge of the system state. If the protocol can prohibit states in one forked branch from appearing to happen before those in other branches, as a consequence, all future operations from client a can be only applied to servers in fork set FS_α, preserving the join-at-most-once property of fork* consistency.

The join-at-most-once property is still useful for system auditing. We can periodically collect all clients' current knowledge to check that all have seen each others' latest operations. Suppose we have made a sequence of checks at times t^1, t^2 ... t^n, where t^n is the time of the latest check, then the join-at-most-once property guarantees that, up to time t^{n-1}, the system has *never* violated ideal consistency.

As in the example in section 2.2, after client a finishes with result list $\langle op^1, op^2 \rangle$ and client b with $\langle op^1, op^3 \rangle$, there might come along another operation op^4 from client c that shows up in both a and b's result lists. Now, for client a, the new result list is $\langle op^1, op^2, op^4 \rangle$, and for b, it is $\langle op^1, op^3, op^4 \rangle$. Yet, the system still delivers fork* consistency at this moment, but not fork consistency, nor fetch-modify consistency. However, fork* consistency prevents malicious servers from showing c's future operations to both a and b from this moment on.

Chapter 3

SUNDR

In this chapter, we describe a new network file system, SUNDR, that achieves fork consistency while placing minimal trust in the storage infrastructure. SUNDR needs much less trust on the server side than previous approaches. In particular, SUNDR does not trust any of its server component. Through the SUNDR protocol, clients can detect any unauthorized operation and most data tampering immediately. In the worst case, malicious servers can conceal users' operations from each other; SUNDR still can detect this misbehavior whenever the invloved users communicate with each other directly.

We discuss the system setting for SUNDR in section 3.1. Then we describe the SUNDR protocol in stages in section 3.2. First, we show a naïve design that achieves fork consistency, albeit at a prohibitive cost. Second, we give a preliminary SUNDR protocol, in which all file system operations are serialized by a global (though untrusted) lock. Finally, we present the full-fledged SUNDR protocol. We also discuss the implementation and evaluate it afterwards.

3.1 Setting

SUNDR provides a file system interface to remote storage, like NFS [74] and other network file systems. To secure a source code repository, for instance, members of a project can mount a remote SUNDR file system on directory `/sundr` and use `/sundr/cvsroot` as a CVS repository. All checkouts and commits then take place through SUNDR, ensuring users will detect any attempts by the hosting site to tamper with repository contents.

Figure 3.1 shows SUNDR's basic architecture. When applications access the file system, the client software internally translates their system calls into a series of *fetch* and *modify* operations, where fetch means retrieving a file's contents or validating a cached local copy, and modify means making new file system state visible to other users. Fetch and modify, in turn, are implemented in terms of SUNDR protocol RPCs to the server. Section 3.2 explains the protocol, while Section 3.4 describes the server design.

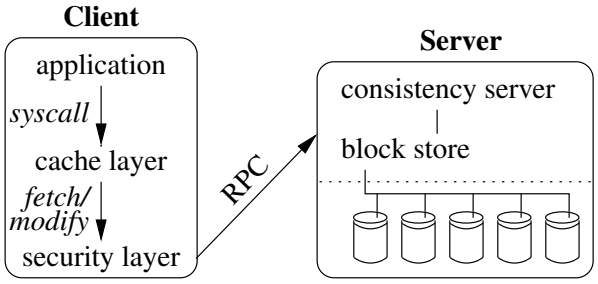


Figure 3.1: Basic SUNDR architecture.

To set up a SUNDR server, one runs the server software on a networked machine with dedicated SUNDR disks or partitions. The server can then host one or more file systems. To create a file system, one generates a public/private *superuser* signature key pair and gives the public key to the server, while keeping the private key secret. The private key provides exclusive write access to the root directory of the file system. It also directly or indirectly allows access to any file below the root. However, the privileges are confined to that one file

system. Thus, when a SUNDR server hosts multiple file systems with different superusers, no single person has write access to all files.

Each user of a SUNDR file system also has a signature key. When establishing accounts, users exchange public keys with the superuser. The superuser manages accounts with two superuser-owned file in the root directory of the file system: `.sundr.users` lists users' public keys and numeric IDs, while `.sundr.group` designates groups and their membership. To mount a file system, one must specify the superuser's public key as a command-line argument to the client, and must furthermore give the client access to a private key. (SUNDR could equally well manage keys and groups with more flexible certificate schemes; the system only requires some way for users to validate each other's keys and group membership.)

Throughout this paper, we use the term *user* to designate an entity possessing the private half of a signature key mapped to some user ID in the `.sundr.users` file. Depending on context, this can either be the person who owns the private key, or a client using the key to act on behalf of the user. However, SUNDR assumes a user is aware of the last operation he or she has performed. In the implementation, the client remembers the last operation it has performed on behalf of each user. To move between clients, a user needs both his or her private key and the last operation performed on his or her behalf (concisely specified by a version number). Alternatively, one person can employ multiple user IDs (possibly with the same public key) for different clients, assigning all file permissions to a personal group.

SUNDR's architecture draws an important distinction between the administration of servers and the administration of file systems. To administer a server, one does not need any private superuser keys.[1] In fact, for best security, key pairs should be generated on

[1]The server does actually have its own public key, but only to prevent network attackers from "framing" honest servers; the server key is irrelevant to SUNDR's security against compromised servers.

separate, trusted machines, and private keys should never reside on the server, even in memory. Important keys, such as the superuser key, should be stored off line when not in use (for example on a floppy disk, encrypted with a passphrase).

3.2 The SUNDR protocol

SUNDR's protocol lets clients detect unauthorized attempts to modify files, even by attackers in control of the server. When the server behaves correctly, a fetch reflects exactly the authorized modifications that happened before it. This property is *fetch-modify consistency*, as defined in section 2.1.

If the server is dishonest, clients enforce a slightly weaker property, namely *fork consistency*. Intuitively, under fork consistency, a dishonest server could cause a fetch by a user A to miss a modify by B. However, either user will detect the attack upon seeing a subsequent operation by the other. Thus, to perpetuate the deception, the server must fork the two user's views of the file system. Put equivalently, if A's client accepts some modification by B, then at least until B performed that modification, both users had identical, fetch-modify-consistent views of the file system.

We have formally specified fork consistency in chapter 2, and, assuming digital signatures and a collision-resistant hash function, proven SUNDR's protocol achieves it in [61]. Therefore, a violation of fork consistency means the underlying cryptography was broken, the implementation deviated from the protocol, or there is a flaw in our mapping from high-level Unix system calls to low-level fetch and modify operations.

3.2.1 A straw-man file system

In the roughest approximation of SUNDR, the straw-man file system, we avoid any concurrent operations and allow the system to consume unreasonable amounts of bandwidth and computation. The server maintains a single, untrusted global lock on the file system. To fetch or modify a file, a user first acquires the lock, then performs the desired operation, then releases the lock. So long as the server is honest, the operations are totally ordered and each operation completes before the next begins.

The straw-man file server stores a complete, ordered list of every fetch or modify operation ever performed. Each operation also contains a digital signature from the user who performed it. The signature covers not just the operation but also *the complete history of all operations that precede it*. For example, after five operations, the history might appear as follows:

fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	mod(f_2) user A sig	fetch(f_2) user B sig

To fetch or modify a file, a client acquires the global lock, downloads the entire history of the file system, and validates each user's most recent signature. The client also checks that its own user's previous operation is in the downloaded history (unless this is the user's very first operation on the file system).

The client then traverses the operation history to construct a local copy of the file system. For each modify encountered, the client additionally checks that the operation was actually permitted, using the user and group files to validate the signing user against the file's owner or group. If all checks succeed, the client appends a new operation to the list, signs the new history, sends it to the server, and releases the lock. If the operation is a modification, the appended record contains new contents for one or more files or directories.

Now consider, informally, what a malicious server can do. To convince a client of a file modification, the server must send it a signed history. Assuming the server does not know users' private keys and cannot forge signatures, any modifications clients accept must actually have been signed by an authorized user. The server can still trick users into signing inappropriate histories, however, by concealing other users' previous operations. For instance, consider what would happen in the last operation of the above history if the server failed to show user B the most recent modification to file f_2. Users A and B would sign the following histories:

user A:	fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	mod(f_2) user A sig
user B:	fetch(f_2) user A sig	mod(f_3) user B sig	fetch(f_3) user A sig	fetch(f_2) user B sig

Neither history is a prefix of the other. Since clients always check for their own user's previous operation in the history, from this point on, A will sign only extensions of the first history and B will sign only extensions of the second. Thus, while before the attack the users enjoyed fetch-modify consistency, after the attack the users have been forked.

Suppose further that the server acts in collusion with malicious users or otherwise comes to possess the signature keys of compromised users. If we restrict the analysis to consider only histories signed by honest (i.e., uncompromised) users, we see that a similar forking property holds. Once two honest users sign incompatible histories, they cannot see each others' subsequent operations without detecting the problem. Of course, since the server can extend and sign compromised users' histories, it can change any files compromised users can write. The remaining files, however, can be modified only in honest users' histories and thus continue to be fork consistent.

3.2.2 Implications of fork consistency

Fork consistency is the strongest notion of integrity possible without on-line trusted parties. Suppose user A comes on line, modifies a file, and goes off line. Later, B comes on line and reads the file. If B doesn't know whether A has accessed the file system, it cannot detect an attack in which the server simply discards A's changes. Fork consistency implies this is the only type of undetectable attack by the server on file integrity or consistency. Moreover, if A and B ever communicate or see each other's future file system operations, they can detect the attack.

Given fork consistency, one can leverage any trusted parties that are on line to gain stronger consistency, even fetch-modify consistency. For instance, as described later in Section 3.4, the SUNDR server consists of two programs, a block store for handling data, and a consistency server with a very small amount of state. Moving the consistency server to a trusted machine trivially guarantees fetch-modify consistency. The problem is that trusted machines may have worse connectivity or availability than untrusted ones.

To bound the window of inconsistency without placing a trusted machine on the critical path, one can use a "time stamp box" with permission to write a single file. The box could simply update that file through SUNDR every 5 seconds. All users who see the box's updates know they could only have been partitioned from each other in the past 5 seconds. Such boxes could be replicated for Byzantine fault tolerance, each replica updating a single file.

Alternatively, direct client-client communication can be leveraged to increase consistency. Users can write login and logout records with current network addresses to files so as to find each other and continuously exchange information on their latest operations. If a malicious server cannot disrupt network communication between clients, it will be unable to fork the file system state once on-line clients know of each other. Those who deem

malicious network partitions serious enough to warrant service delays in the face of client failures can conservatively pause file access during communication outages.

3.2.3 Serialized SUNDR

The straw-man file system is impractical for two reasons. First, it must record and ship around complete file system operation histories, requiring enormous amounts of bandwidth and storage. Second, the serialization of operations through a global lock is impractical for a multi-user network file system. This subsection explains SUNDR's solution to the first problem; we describe a simplified file system that still serializes operations with a global lock, but is in other respects similar to SUNDR. Subsection 3.2.4 explains how SUNDR lets clients execute non-conflicting operations concurrently.

Instead of signing operation histories, as in the straw-man file system, SUNDR effectively takes the approach of signing file system snapshots. Roughly speaking, users sign messages that tie together the complete state of all files with two mechanisms. First, all files writable by a particular user or group are efficiently aggregated into a single hash value called the *i-handle* using *hash trees* [62]. Second, each i-handle is tied to the latest version of every other i-handle using *version vectors* [66].

3.2.3.1 Data structures

Before delving into the protocol's details, we begin by describing SUNDR's storage interface and data structures. Like several recent file systems [39, 63], SUNDR names all on-disk data structures by cryptographic handles. The block store indexes most persistent data structures by their 20-byte SHA-1 [35] hashes, making the server a kind of large, high-performance hash table. It is believed to be computationally infeasible to find any two different data blocks with the same SHA-1 hash. Thus, when a client requests the block

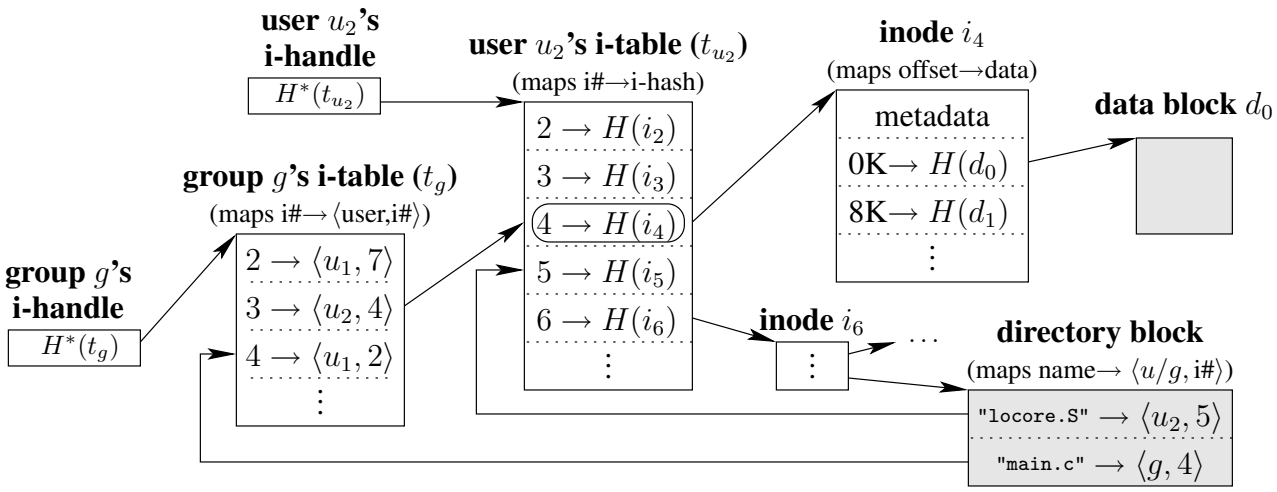


Figure 3.2: User and group i-handles. An *i-handle* is the root of a hash tree containing a user or group *i-table*. (H denotes SHA-1, while H^* denotes recursive application of SHA-1 to compute the root of a hash tree.) A *group i-table* maps group inode numbers to user inode numbers. A *user i-table* maps a user's inode numbers to i-hashes. An *i-hash* is the hash of an inode, which in turn contains hashes of file data blocks.

with a particular hash, it can check the integrity of the response by hashing it. An incidental benefit of hash-based storage is that blocks common to multiple files need be stored only once.

SUNDR also stores messages signed by users. These are indexed by a hash of the public key and an index number (so as to distinguish multiple messages signed by the same key).

Figure 3.2 shows the persistent data structures SUNDR stores and indexes by hash, as well as the algorithm for computing i-handles. Every file is identified by a $\langle \text{principal}, \text{i-number} \rangle$ pair, where principal is the user or group allowed to write the file, and i-number is a per-principal inode number. Directory entries map file names onto $\langle \text{principal}, \text{i-number} \rangle$ pairs. A per-principal data structure called the *i-table* maps each i-number in use to the corresponding inode. User i-tables map each i-number to a hash of the corresponding inode, which we call the file's *i-hash*. Group i-tables add a level of indirection, mapping a group i-number onto a user i-number. (The indirection allows the same user to perform multi-

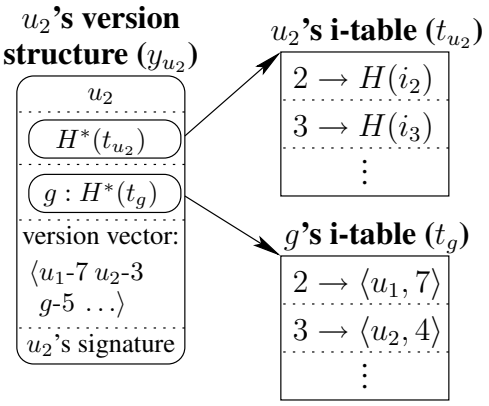


Figure 3.3: A version structure containing a group i-handle.

ple successive writes to a group-owned file without updating the group's i-handle.) Inodes themselves contain SHA-1 hashes of file data blocks and indirect blocks.

Each i-table is stored as a B+-tree, where internal nodes contain the SHA-1 hashes of their children, thus forming a hash tree. The hash of the B+-tree root is the i-handle. Since the block store allows blocks to be requested by SHA-1 hash, given a user's i-handle, a client can fetch and verify any block of any file in the user's i-table by recursively requesting the appropriate intermediary blocks. The next question, of course, is how to obtain and verify a user's latest i-handle.

3.2.3.2 Protocol

i-handles are stored in digitally-signed messages known as *version structures*, shown in Figure 3.3. Each version structure is signed by a particular user. The structure must always contain the user's i-handle. In addition, it can optionally contain one or more i-handles of groups to which the user belongs. Finally, the version structure contains a version vector consisting of a version number for every user and group in the system.

When user u performs a file system operation, u's client acquires the global lock and

downloads the latest version structure for each user and group. We call this set of version structures the *version structure list*, or VSL. (Much of the VSL's transfer can be elided if only a few users and groups have changed version structures since the user's last operation.) The client then computes a new version structure z by potentially updating i-handles and by setting the version numbers in z to reflect the current state of the file system.

More specifically, to set the i-handles in z, on a fetch, the client simply copies u's previous i-handle into z, as nothing has changed. For a modify, the client computes and includes new i-handles for u and for any groups whose i-tables it is modifying.

The client then sets z's version vector to reflect the version number of each VSL entry. For any version structure like z, and any principal (user or group) p, let $z[p]$ denote p's version number in z's version vector (or 0 if z contains no entry for p). For each principal p, if y_p is p's entry in the VSL (i.e., the version structure containing p's latest i-handle), set $z[p] \leftarrow y_p[p]$.

Finally, the client bumps version numbers to reflect the i-handles in z. It sets $z[u] \leftarrow z[u] + 1$, since z always contains u's i-handle, and for any group g whose i-handle z contains, sets $z[g] \leftarrow z[g] + 1$.

The client then checks the VSL for consistency. Given two version structures x and y, we define $x \leq y$ iff $\forall p\ x[p] \leq y[p]$. To check consistency, the client verifies that the VSL contains u's previous version structure, and that the set of all VSL entries combined with z is totally ordered by $\leq$. If it is, the user signs the new version structure and sends it to the server with a COMMIT RPC. The server adds the new structure to the VSL and retires the old entries for updated i-handles, at which point the client releases the file system lock.

Figure 3.4 revisits the forking attack from the end of Section 3.2.1, showing how version vectors evolve in SUNDR. With each version structure signed, a user reflects the highest version number seen from every other user, and also increments his own version number

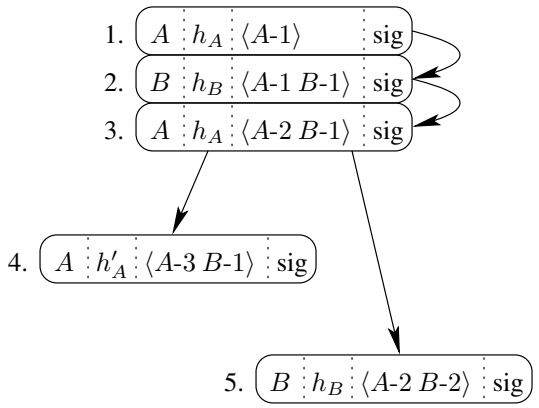


Figure 3.4: Signed version structures with a forking attack.

to reflect the most recent i-handle. A violation of consistency causes users to sign *incompatible* version structures—i.e., two structures x and y such that $x \not\leq y$ and $y \not\leq x$. In this example, the server performs a forking attack after step 3. User A updates his i-handle from h_A to h'_A in 4, but in 5, B is not aware of the change. The result is that the two version structures signed in 4 and 5 are incompatible.

Just as in the straw-man file system, once two users have signed incompatible version structures, they will never again sign compatible ones, and thus cannot ever see each other's operations without detecting the attack (as proven in [60]).

One optimization worth mentioning is that SUNDR amortizes the cost of recomputing hash trees over several operations. As shown in Figure 3.5, an i-handle contains not just a hash tree root, but also a small log of changes that have been made to the i-table. The change log furthermore avoids the need for other users to fetch i-table blocks when re-validating a cached file that has not changed since the hash tree root was last computed.

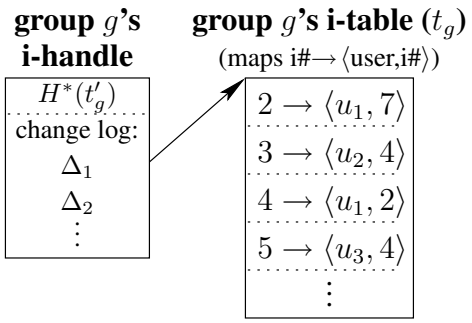


Figure 3.5: i-table for group g, showing the change log. t'_g is a recent i-table; applying the log to t'_g yields t_g.

3.2.4 Concurrent SUNDR

While the version structures in SUNDR detect inconsistency, serialized SUNDR is too conservative in what it prohibits. Each client must wait for the previous client's version vector before computing and signing its own, so as to reflect the appropriate version numbers. Instead, we would like most operations to proceed concurrently. The only time one client should have to wait for another is when it reads a file the other is in the process of writing.[2]

3.2.4.1 Update certificates

SUNDR's solution to concurrent updates is for users to pre-declare a fetch or modify operation before receiving the VSL from the server. They do so with signed messages called *update certificates*. If y_u is u's current VSL entry, an update certificate for u's next operation contains:

- u's next version number $(y_u[u] + 1$, unless u is pipelining multiple updates$)$,
- a hash of u's VSL entry $(H(y_u))$, and

[2]One might wish to avoid waiting for other clients even in the event of such a read-after-write conflict. However, this turns out to be impossible with untrusted servers. If a single signed message could atomically switch between two file states (as shown in the proof in section 2.2.2), the server could conceal the change initially, then apply it long after forking the file system, when users should no longer see each others' updates.

- a (possibly empty) list of modifications to perform.

Each modification (or *delta*) can be one of four types:

- Set file $\langle \text{user}, \text{i\#} \rangle$ to i-hash h.
- Set group file $\langle \text{group}, \text{i\#} \rangle$ to $\langle \text{user}, \text{i\#} \rangle$.
- Set/delete entry *name* in directory $\langle \text{user/group}, \text{i\#} \rangle$.
- Pre-allocate a range of group i-numbers (pointing them to unallocated user i-numbers).

The client sends the update certificate to the server in an UPDATE RPC. The server replies with both the VSL and a list of all pending operations not yet reflected in the VSL, which we call the *pending version list* or PVL.

Note that both fetch and modify operations require UPDATE RPCs, though fetches contain no deltas. (The RPC name refers to updating the VSL, not file contents.) Moreover, when executing complex system calls such as *rename*, a single UPDATE RPC may contain deltas affecting multiple files and directories, possibly in different i-tables.

An honest server totally orders operations according to the arrival order of UPDATE RPCs. If operation op^1 is reflected in the VSL or PVL returned for op^2's UPDATE RPC, then op^1 happened before op^2. Conversely, if op^2 is reflected in op^1's VSL or PVL, then op^2 happened before op^1. If neither happened before the other, then the server has mounted a forking attack.

When signing an update certificate, a client cannot predict the version vector of its next version structure, as the vector may depend on concurrent operations by other clients. The server, however, knows *precisely* what operations the forthcoming version structure must reflect. For each update certificate, the server therefore calculates the forthcoming version structure, except for the i-handle. This unsigned version structure is paired with its update

certificate in the PVL, so that the PVL is actually a list of ⟨update certificate, unsigned version structure⟩ pairs.

The algorithm for computing a new version structure, z, begins as in serialized SUNDR: for each principal p, set $z[p] \leftarrow y_p[p]$, where y_p is p's entry in the VSL. Then, z's version vector must be incremented to reflect pending updates in the PVL, including u's own. For user version numbers, this is simple; for each update certificate signed by user u, set $z[u] \leftarrow z[u] + 1$. For groups, the situation is complicated by the fact that operations may commit out of order when slow and fast clients update the same i-table. For any PVL entry updating group g's i-table, we wish to increment $z[g]$ if and only if the PVL entry happened after y_g (since we already initialized $z[g]$ with $y_g[g]$). We determine whether or not to increment the version number by comparing y_g to the PVL entry's unsigned version vector, call it ℓ. If $\ell \not\leq y_g$, set $z[g] \leftarrow z[g] + 1$. The result is the same version vector one would obtain in serialized SUNDR by waiting for all previous version structures.

Upon receiving the VSL and PVL, a client ensures that the VSL, the unsigned version structures in the PVL, and its new version structure are totally ordered. It also checks for conflicts. If none of the operations in the PVL change files the client is currently fetching or group i-tables it is modifying, the client simply signs a new version structure and sends it to the server for inclusion in the VSL.

3.2.4.2 Update conflicts

If a client is fetching a file and the PVL contains a modification to that file, this signifies a read-after-write conflict. In this case, the client still commits its version structure as before but then waits for fetched files to be committed to the VSL before returning to the application. (A FETCHPENDING RPC lets clients request a particular version structure from the server as soon as it arrives.)

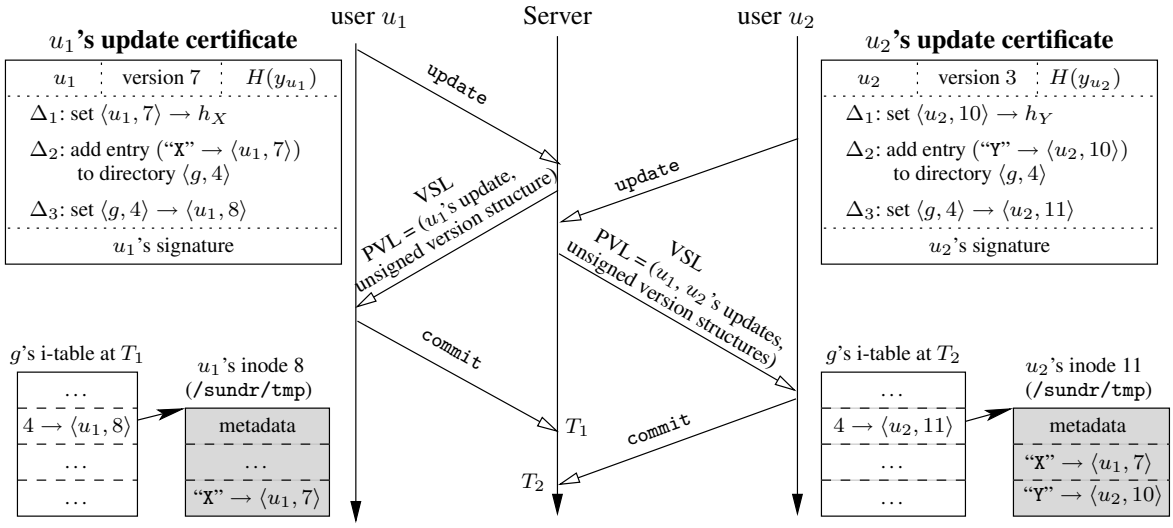


Figure 3.6: Concurrent updates to `/sundr/tmp/` by different users.

A trickier situation occurs when the PVL contains a modification to a group i-handle that the client also wishes to modify, signifying a write-after-write conflict. How should a client, u, modifying a group g's i-table, t_g, recompute g's i-handle, h_g, when other operations in the PVL also affect t_g? Since any operation in the PVL happened before u's new version structure, call it z, the handle h_g in z must reflect all operations on t_g in the PVL. On the other hand, if the server has behaved incorrectly, one or more of the forthcoming version structures corresponding to these PVL entries may be incompatible with z. In this case, it is critical that z not somehow "launder" operations that should have alerted people to the server's misbehavior.

Recall that clients already check the PVL for read-after-write conflicts. When a client sees a conflicting modification in the PVL, it will wait for the corresponding VSL entry even if u has already incorporated the change in h_g. However, the problem remains that a malicious server might prematurely drop entries from the PVL, in which case a client could incorrectly fetch modifications reflected by t_g but never properly committed.

The solution is for u to incorporate any modifications of t_g in the PVL not yet reflected in y_g, and also to record the current contents of the PVL in a new field of the version structure. In this way, other clients can detect missing PVL entries when they notice those entries referenced in u's version structure. Rather than include the full PVL, which might be large, u simply records, for each PVL entry, the user performing the operation, that user's version number for the operation, and a hash of the expected version structure with i-handles omitted.

When u applies changes from the PVL, it can often do so by simply appending the changes to the change log of g's i-handle, which is far more efficient than rehashing the i-table and often saves u from fetching uncached portions of the i-table.

3.2.4.3 Example

Figure 3.6 shows an example of two users u_1 and u_2 in the group g modifying the same directory. u_1 creates file `X` while u_2 creates `Y`, both in `/sundr/tmp/`. The directory is group-writable, while the files are not. (For the example, we assume no other pending updates.)

Assume `/sundr/tmp/` is mapped to group g's i-number 4. User u_1 first calculates the i-hash of file `X`, call it h_X, then allocates his own i-number for `X`, call it 7. u_1 then allocates another i-number, 8, to hold the contents of the modified directory. Finally, u_1 sends the server an update certificate declaring three deltas, namely the mapping of file $\langle u_1, 7\rangle$ to i-hash h_X, the addition of entry ("`X`" $\rightarrow \langle u_1, 7\rangle$) to the directory, and the re-mapping of g's i-number 4 to $\langle u_1, 8\rangle$.

u_2 similarly sends the server an update certificate for the creation of file `Y` in `/sundr/tmp/`. If the server orders u_1's update before u_2's, it will respond to u_1 with the VSL and a PVL containing only u_1's update, while it will send u_2 a PVL reflecting both updates. u_2 will therefore apply u_1's modification to the directory before computing the i-handle

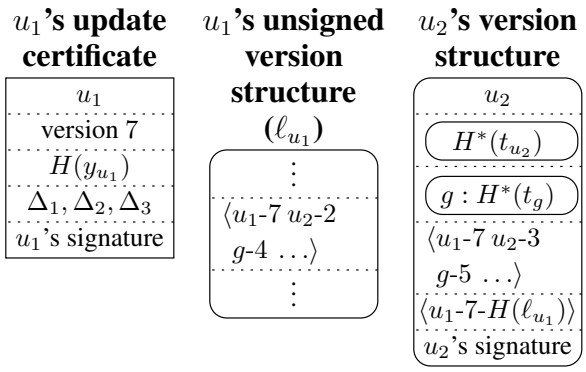


Figure 3.7: A pending update by user u_1, reflected in user u_2's version structure.

for g, incorporating u_1's directory entry for `X`. u_2 would also ordinarily incorporate u_1's re-mapping of the directory $\langle g, 4\rangle \rightarrow \langle u_1, 7\rangle$, except that u_2's own re-mapping of the same directory supersedes u_1's.

An important subtlety of the protocol, shown in Figure 3.7, is that u_2's version structure contains a hash of u_1's forthcoming version structure (without i-handles). This ensures that if the server surreptitiously drops u_1's update certificate from the PVL before u_1 commits, whoever sees the incorrect PVL must be forked from both u_1 and u_2.

3.3 Discussion

SUNDR only detects attacks; it does not resolve them. Following a server compromise, two users might find themselves caching divergent copies of the same directory tree. Resolving such differences has been studied in the context of optimistic file system replication [50, 77], though invariably some conflicts require application-specific reconciliation. With CVS, users might employ CVS's own merging facilities to resolve forks.

SUNDR's protocol leaves considerable opportunities for compression and optimization. In particular, though version structure signatures must cover a version vector with all

users and groups, there is no need to transmit entire vectors in RPCs. By ordering entries from most- to least-recently updated, the tail containing idle principals can be omitted on all but a client's first UPDATE RPC. Moreover, by signing a hash of the version vector and hashing from oldest to newest, clients could also pre-hash idle principals' version numbers to speed version vector signatures. Finally, the contents of most unsigned version structures in the PVL is implicit based on the order of the PVL and could be omitted (since the server computes unsigned version structures deterministically based on the order in which it receives UPDATE RPCs). None of these optimizations is currently implemented.

SUNDR's semantics differ from those of traditional Unix. Clients supply file modification and inode change times when modifying files, allowing values that might be prohibited in Unix. There is no time of last access. Directories have no "sticky bit." A group-writable file in SUNDR is not owned by a user (as in Unix) but rather is owned by the group; such a file's "owner" field indicates the last user who wrote to it. In contrast to Unix disk quotas, which charge the owner of a group-writable file for writes by other users, if SUNDR's block store enforced quotas, they would charge each user for precisely the blocks written by that user.

One cannot change the owner of a file in SUNDR. However, SUNDR can copy arbitrarily large files at the cost of a few pointer manipulations, due to its hash-based storage mechanism. Thus, SUNDR implements *chown* by creating a copy of the file owned by the new user or group and updating the directory entry to point to the new copy. Doing so requires write permission on the directory and changes the semantics of hard links (since *chown* only affects a single link).

Yet another difference from Unix is that the owner of a directory can delete any entries in the directory, including non-empty subdirectories to which he or she does not have write permission. Since Unix already allows users to rename such directories away, additionally

allowing delete permission does not appreciably affect security. In a similar vein, users can create multiple hard links to directories, which could confuse some Unix software, or could be useful in some situations. Other types of malformed directory structure are interpreted as equivalent to something legal (e.g., only the first of two duplicate directory entries counts).

SUNDR does not yet offer read protection or confidentiality. Confidentiality can be achieved through encrypted storage, a widely studied problem [20, 42, 47, 82].

In terms of network latency, SUNDR is comparable with other polling network file systems. SUNDR waits for an UPDATE RPC to complete before returning from an application file system call. If the system call caused only modifies, or if all fetched data hit in the cache, this is the only synchronous round trip required; the COMMIT can be sent in the background (except for *fsync*). This behavior is similar to systems such as NFS3, which makes an ACCESS RPC on each open and writes data back to the server on each close. We note that callback- or lease-based file systems can actually achieve zero round trips when the server has committed to notifying clients of cache invalidations.

3.4 File system implementation

The SUNDR client is implemented at user level, using a modified version of the **xfs** device driver from the ARLA file system [81] on top of a slightly modified FreeBSD kernel. Server functionality is divided between two programs, a consistency server, which handles update certificates and version structures, and a block store, which actually stores data, update certificates, and version structures on disk. For experiments in this paper, the block server and consistency server ran on the same machine, communicating over Unix-domain sockets. They can also be configured to run on different machines and communicate over an authenticated TCP connection.

3.4.1 File system client

The **xfs** device driver used by SUNDR is designed for whole-file caching. When a file is opened, **xfs** makes an upcall to the SUNDR client asking for the file's data. The client returns the identity of a local file that has a cached copy of the data. All reads and writes are performed on the cached copy, without further involvement of SUNDR. When the file is closed (or flushed with *fsync*), if it has been modified, **xfs** makes another upcall asking the client to write the data back to the server. Several other types of upcalls allow **xfs** to look up names in directories, request file attributes, create/delete files, and change metadata.

As distributed, **xfs**'s interface posed two problems for SUNDR. First, **xfs** caches information like local file bindings to satisfy some requests without upcalls. In SUNDR, some of these requests require interaction with the consistency server for the security properties to hold. We therefore modified **xfs** to invalidate its cache tokens immediately after getting or writing back cached data, so as to ensure that the user-level client gets control whenever the protocol requires an UPDATE RPC. We similarly changed **xfs** to defeat the kernel's name cache.

Second, some system calls that should require only a single interaction with the SUNDR consistency server result in multiple kernel vnode operations and **xfs** upcalls. For example, the system call "`stat ("a/b/c", &sb)`" results in three **xfs** GETNODE upcalls (for the directory lookups) and one GETATTR. The whole system call should require only one UPDATE RPC. Yet if the user-level client does not know that the four upcalls are on behalf of the same system call, it must check the freshness of its i-handles four separate times with four UPDATE RPCs.

To eliminate unnecessary RPCs, we modified the FreeBSD kernel to count the number of system call invocations that might require an interaction with the consistency server. We increment the counter at the start of every system call that takes a pathname as an argument

(e.g., `stat`, `open`, `readlink`, `chdir`). The SUNDR client memory-maps this counter and records the last value it has seen. If **xfs** makes an upcall that does not change the state of the file system, and the counter has not changed, then the client can use its cached copies of all i-handles.

3.4.2 Signature optimization

The cost of digital signatures on the critical path in SUNDR is significant. Our implementation therefore uses the ESIGN signature scheme,[3] which is over an order of magnitude faster than more popular schemes such as RSA. All experiments reported in this paper use 2,048-bit public keys, which, with known techniques, would require a much larger work factor to break than 1,024-bit RSA.

To move verification out of the critical path, the consistency server also processes and replies to an UPDATE RPC before verifying the signature on its update certificate. It verifies the signature after replying, but before accepting any other RPCs from other users. If the signature fails to verify, the server removes the update certificate from the PVL and and drops the TCP connection to the forging client. (Such behavior is acceptable because only a faulty client would send invalid signatures.) This optimization allows the consistency server's verification of one signature to overlap with the client's computation of the next.

Clients similarly overlap computation and network latency. Roughly half the cost of an ESIGN signature is attributable to computations that do not depend on the message contents. Thus, while waiting for the reply to an UPDATE RPC, the client precomputes its next signature.

[3] Specifically, we use the version of ESIGN shown secure in the random oracle model by [64], with parameter $e = 8$.

3.4.3 Consistency server

The consistency server orders operations for SUNDR clients and maintains the VSL and PVL as described in Section 3.2. In addition, it polices client operations and rejects invalid RPCs, so that a malicious user cannot cause an honest server to fail. For crash recovery, the consistency server must store the VSL and PVL to persistent storage *before* responding to client RPCs. The current consistency server stores these to the block server. Because the VSLs and PVLs are small relative to the size of the file system, it would also be feasible to use non-volatile RAM (NVRAM).

3.5 Block store implementation

A block storage daemon called *bstor* handles all disk storage in SUNDR. Clients interact directly with *bstor* to store blocks and retrieve them by SHA-1 hash value. The consistency server uses *bstor* to store signed update and version structures. Because a SUNDR server does not have signature keys, it lacks permission to repair the file system after a crash. For this reason, *bstor* must synchronously store all data to disk before returning to clients, posing a performance challenge. *bstor* therefore heavily optimizes synchronous write performance.

bstor's basic idea is to write incoming data blocks to a temporary log, then to move these blocks to Venti-like storage in batches. Venti [68] is an archival block store that appends variable-sized blocks to a large, append-only IDE log disk while indexing the blocks by SHA-1 hash on one or more fast SCSI disks. *bstor*'s temporary log relaxes the archival semantics of Venti, allowing short-lived blocks to be deleted within a small window of their creation. *bstor* maintains an archival flavor, though, by supporting periodic file system snapshots.

The temporary log allows *bstor* to achieve low latency on synchronous writes, which under Venti require an index lookup to ensure the block is not a duplicate. Moreover, *bstor* sector-aligns all blocks in the temporary log, temporarily wasting an average of half a sector per block so as to avoid multiple writes to the same sector, which would each cost at least one disk rotation. The temporary log improves write throughput even under sustained load, because transferring blocks to the permanent log in large batches allows *bstor* to order index disk accesses.

bstor keeps a large in-memory cache of recently used blocks. In particular, it caches all blocks in the temporary log so as to avoid reading from the temporary log disk. Though *bstor* does not currently use special hardware, in Section 4.5 we describe how SUNDR's performance would improve if *bstor* had a small amount of NVRAM to store update certificates.

3.5.1 Interface

bstor exposes the following RPCs to SUNDR clients:

`store` (*header*, *block*)

`retrieve` (*hash*)

`vstore` (*header*, *pubkey*, *n*, *block*)

`vretrieve` (*pubkey*, *n*, [*time*])

`decref` (*hash*)

`snapshot` ()

The `store` RPC writes a block and its header to stable storage if *bstor* does not already have a copy of the block. The header has information encapsulating the block's owner and creation time, as well as fields useful in concert with encoding or compression. The

`retrieve` RPC retrieves a block from the store given its SHA-1 hash. It also returns the first header `stored` with the particular block.

The `vstore` and `vretrieve` RPCs are like `store` and `retrieve`, but for signed blocks. Signed blocks are indexed by the public key and a small index number, n. `vretrieve`, by default, fetches the most recent version of a signed block. When supplied with a timestamp as an optional third argument, `vretrieve` returns the newest block written before the given time.

`decref` (short for "decrement reference count") informs the store that a block with a particular SHA-1 hash might be discarded. SUNDR clients use `decref` to discard temporary files and short-lived metadata. *bstor*'s deletion semantics are conservative. When a block is first stored, *bstor* establishes a short window (one minute by default) during which it can be deleted. If a client `store`s then `decref`s a block within this window, *bstor* marks the block as garbage and does not permanently store it. If two clients store the same block during the dereference window, the block is marked as permanent.

An administrator should issue a `snapshot` RPC periodically to create a coherent file system image that clients can later revert to in the case of accidental data disruption. Upon receiving this RPC, *bstor* simply immunizes all newly-stored blocks from future `decref`'s and flags them to be stored in the permanent log. `snapshot` and `vretrieve`'s *time* argument are designed to allow browsing of previous file system state, though this functionality is not yet implemented in the client.

3.5.2 Index

bstor's index system locates blocks on the permanent log, keyed by their SHA-1 hashes. An ideal index is a simple in-memory hash table mapping 20-byte SHA-1 block hashes to 8-byte log disk offsets. If we assume that the average block stored on the system is 8 KB, then

the index must have roughly $1/128$ the capacity of the log disk. Although at present such a ratio of disk to memory is possible with commodity components, we are not convinced that memory will keep up with hard disks in the future.

We instead use Venti's strategy of striping a disk-resident hash table over multiple high-speed SCSI disks. *bstor* hashes 20-byte SHA-1 hashes down to $\langle$*index-disk-id*, *index-disk-offset*$\rangle$ pairs. The disk offsets point to sector-sized on-disk data structures called *buckets*, which contain 15 *index-entries*, sorted by SHA-1 hash. *index-entries* in turn map SHA-1 hashes to offsets on the permanent data log. Whenever an index-entry is written to or read from disk, *bstor* also stores it in an in-memory LRU cache.

bstor accesses the index system as Venti does when answering `retrieve` RPCs that miss the block cache. When *bstor* moves data from the temporary to the permanent log, it must access the index system sometimes twice per block (once to check a block is not a duplicate, and once to write a new index entry after the block is committed the permanent log). In both cases, *bstor* sorts these disk accesses so that the index disks service a batch of requests with one disk arm sweep. Despite these optimizations, *bstor* writes blocks to the permanent log in the order they arrived; randomly reordering blocks would hinder sequential read performance over large files.

3.5.3 Data management

To recover from a crash or an unclean shutdown, the system first recreates an index consistent with the permanent log, starting from its last known checkpoint. Index recovery is necessary because the server updates the index lazily after storing blocks to the permanent log. *bstor* then processes the temporary log, storing all fresh blocks to the permanent log, updating the index appropriately.

Venti's authors argue that archival storage is practical because IDE disk capacity is

growing faster than users generate data. For users who do not fit this paradigm, however, *bstor* could alternatively be modified to support mark-and-sweep garbage collection. The general idea is to copy all reachable blocks to a new log disk, then recycle the old disk. With two disks, *bstor* could still respond to RPCs during garbage collection.

3.6 Performance

The primary goal in testing SUNDR was to ensure that its security benefits do not come at too high a price relative to existing file systems. In this section, we compare SUNDR's overall performance to NFS. We also perform microbenchmarks to help explain our application-level results, and to support our claims that our block server outperforms a Venti-like architecture in our setting.

3.6.1 Experimental setup

We carried out our experiments on a cluster of 3 GHz Pentium IV machines running FreeBSD 4.9. All machines were connected with fast Ethernet with ping times of 110 μs. For block server microbenchmarks, we additionally connected the block server and client with gigabit Ethernet. The machine running *bstor* has 3 GB of RAM and an array of disks: four Seagate Cheetah 18 GB SCSI drives that spin at 15,000 RPM were used for the index; two Western Digital Caviar 180 GB 7200 RPM EIDE drives were used for the permanent and temporary logs.

3.6.2 Microbenchmarks

3.6.2.1 *bstor*

Our goals in evaluating *bstor* are to quantify its raw performance and justify our design improvements relative to Venti. In our experiments, we configured *bstor*'s four SCSI disks each to use 4 GB of space for indexing. If one hopes to maintain good index performance (and not overflow buckets), then the index should remain less than half full. With our configuration (8 GB of usable index and 32-byte index entries), *bstor* can accommodate up to 2 TB of permanent data. For flow control and fairness, *bstor* allowed clients to make up to 40 outstanding RPCs. For the purposes of the microbenchmarks, we disabled *bstor*'s block cache but enabled an index cache of up to 100,000 entries. The circular temporary log was 720 MB and never filled up during our experiments.

We measured *bstor*'s performance while storing and fetching a batch of 20,000 unique 8 KB blocks. Figure 3.8 shows the averaged results from 20 runs of a 20,000 block experiment. In all cases, standard deviations were less than 5% of the average results. The first two results show that *bstor* can absorb bursts of 8 KB blocks at almost twice fast Ethernet rates, but that sustained throughput is limited by *bstor*'s ability to shuffle blocks from the temporary to the permanent logs, which it can do at 11.9 MB/s. The bottleneck in `store`ing blocks to the temporary log is currently CPU, and future versions of *bstor* might eliminate some unnecessary *memcpy*s to achieve better throughput. On the other hand, *bstor* can process the temporary log only as fast as it can read from its index disks, and there is little room for improvement here unless disks become faster or more index disks are used.

To compare with a Venti-like system, we implemented a Venti-like store mechanism. In `venti_store`, *bstor* first checks for a block's existence in the index and stores the block to the permanent log only if it is not found. That is, each `venti_store` entails an access

Operation	MB/s
`store` (burst)	18.4
`store` (sustained)	11.9
`venti_store`	5.1
`retrieve` (random + cold index cache)	1.2
`retrieve` (sequential + cold index cache)	9.1
`retrieve` (sequential + warm index cache)	25.5

Figure 3.8: *bstor* throughput measurements with the block cache disabled.

to the index disks. Our results show that `venti_store` can achieve only 27% of `store`'s burst throughput, and 43% of its sustained throughput.

Figure 3.8 also presents read measurements for *bstor*. If a client reads blocks in the same order they are written (*i.e.*, "sequential" reads), then *bstor* need not seek across the permanent log disk. Throughput in this case is limited by the per-block cost of locating hashes on the index disks and therefore increases to 25.5 MB/s with a warm index cache. Randomly-issued reads fare poorly, even with a warm index cache, because *bstor* must seek across the permanent log. In the context of SUNDR, slow random `retrieves` should not affect overall system performance if the client aggressively caches blocks and reads large files sequentially.

Finally, the latency of *bstor* RPCs is largely a function of seek times. `store` RPCs do not require seeks and therefore return in 1.6 ms. `venti_store` returns in 6.7 ms (after one seek across the index disk at a cost of about 4.4 ms). Sequential `retrieves` that hit and miss the index cache return in 1.9 and 6.3 ms, respectively. A seek across the log disk takes about 6.1 ms; therefore random `retrieves` that hit and miss the index cache return in 8.0 and 12.4 ms respectively.

3.6.2.2 Cryptographic overhead

SUNDR clients sign and verify version structures and update certificates using 2,048-bit ESIGN keys. Our implementation (based on the GNU Multiprecision library version 4.1.4) can complete signatures in approximately 150 μs and can verify them 100 μs. Precomputing a signature requires roughly 80 μs, while finalizing a precomputed signature is around 75 μs. We observed that these measurements can vary on the Pentium IV by as much as a factor of two, even in well-controlled micro-benchmarks. By comparison, an optimized version of the Rabin signature scheme with 1,280-bit keys, running on the same hardware, can compute signatures in 3.1 ms and can verify them in 27 μs.

3.6.3 End-to-end evaluation

In end-to-end experiments, we compare SUNDR to both NFS2 and NFS3 servers running on the same hardware. To show NFS in the best possible light, the NFS experiments run on the fast SCSI disks SUNDR uses for indexes, not the slower, larger EIDE log disks. We include NFS2 results because NFS2's write-through semantics are more like SUNDR's. Both NFS2 and SUNDR write all modified file data to disk before returning from a *close* system call, while NFS3 does not offer this guarantee.

Finally, we described in Section 3.4.3 that SUNDR clients must wait for the consistency server to write small pieces of data (VSLs and PVLs) to stable storage. The consistency server's storing of PVLs in particular is on the client's critical path. We present result sets for consistency servers running with and without flushes to secondary storage. We intend the mode with flushes disabled to simulate a consistency server with NVRAM.

All application results shown are the average of three runs. Relative standard deviations are less than 8% unless otherwise noted.

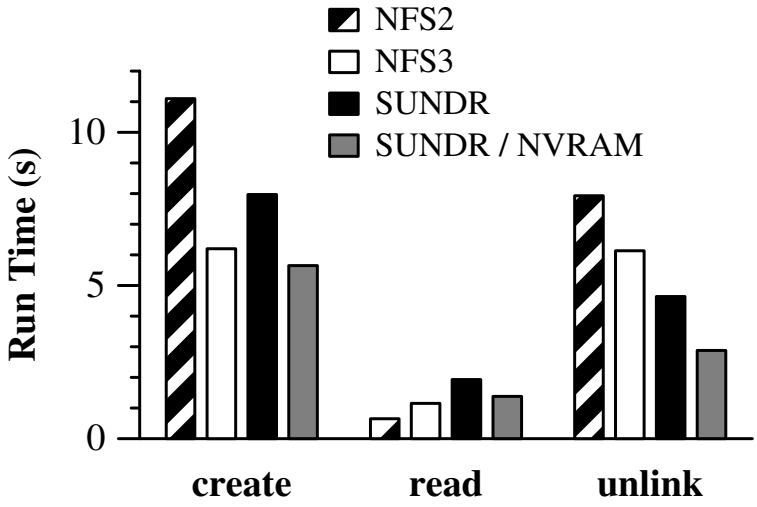


Figure 3.9: Single client LFS Small File Benchmark. 1000 operations on files with 1 KB of random content.

3.6.3.1 LFS small file benchmark

The LFS small file benchmark [73] tests SUNDR's performance on simple file system operations. This benchmark creates 1,000 1 KB files, reads them back, then deletes them. We have modified the benchmark slightly to write random data to the 1 KB files; writing the same file 1,000 times would give SUNDR's hash-based block store an unfair advantage.

Figure 3.9 details our results when only one client is accessing the file system. In the **create** phase of the benchmark, a single file creation entails system calls to *open*, *read* and *close*. On SUNDR/NVRAM, the *open* call involves two serialized rounds of the consistency protocol, each of which costs about 2 ms; the *write* call is a no-op, since file changes are buffered until close; and the *close* call involves one round of the protocol and one synchronous write of file data to the block server, which the client can overlap. Thus, the entire sequence takes about 6 ms. Without NVRAM, each round of the protocol takes approximately 1-2 ms longer, because the consistency server must wait for *bstor* to flush.

Unlike SUNDR, an NFS server must wait for at least one disk seek when creating a

new file because it synchronously writes metadata. A seek costs at least 4 ms on our fast SCSI drives, and thus NFS can do no better than 4 ms per file creation. In practice, NFS requires about 6 ms to service the three system calls in the **create** stage.

In the **read** phase of the benchmark, SUNDR performs one round of the consistency protocol in the *open* system call. The NFS3 client still accesses the server with an `access` RPC, but the server is unlikely to need any data not in its buffer cache at this point, and hence no seeking is required. NFS2 does not contact the server in this phase.

In the **unlink** stage of the benchmark, clients issue a single *unlink* system call per file. An *unlink* for SUNDR triggers one round of the consistency protocol and an asynchronous write to the block server to store updated i-table and directory blocks. SUNDR and SUNDR/NVRAM in particular can outperform NFS in this stage of the experiment because NFS servers again require at least one synchronous disk seek per file *unlink*ed.

We also performed experiments with multiple clients performing the LFS small file benchmark concurrently in different directories. Results for the **create** phase are reported in Figure 3.10 and the other phases of the benchmark show similar trends. A somewhat surprising result is that SUNDR actually scales better than NFS as client concurrency increases in our limited tests. NFS is seek-bound even in the single client case, and the number of seeks the NFS servers require scale linearly with the number of concurrent clients. For SUNDR, latencies induced by the consistency protocol limit individual client performance, but these latencies overlap when clients act concurrently. SUNDR's disk accesses are also scalable because they are sequential, sector-aligned writes to *bstor*'s temporary log.

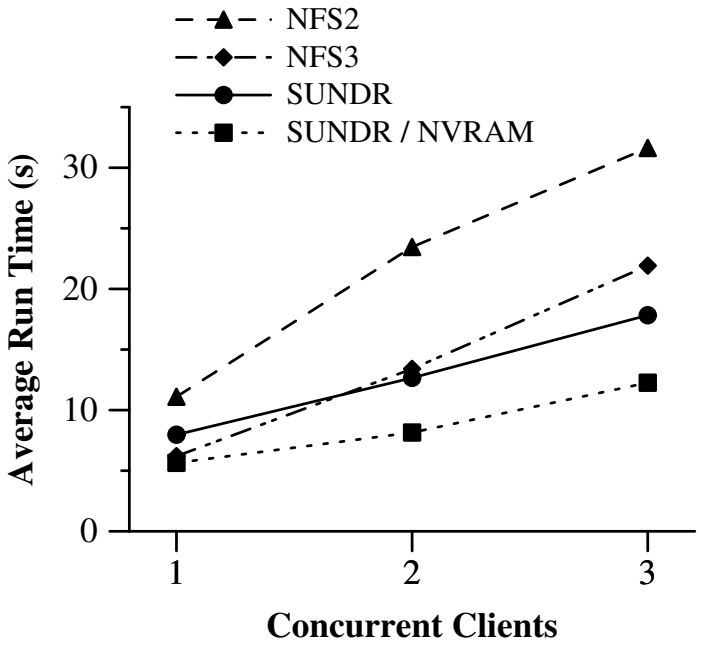


Figure 3.10: Concurrent LFS Small File Benchmark, **create** phase. 1000 creations of 1 KB files. (Relative standard deviation for SUNDR in 3 concurrent clients case is 13.7%)

3.6.3.2 Group contention

The group protocol incurs additional overhead when folding other users' changes into a group i-table or directory. We characterized the cost of this mechanism by measuring a workload with a high degree of contention for a group-owned directory. We ran a micro-benchmark that simultaneously created 300 new files in the same, group-writable directory on two clients. Each concurrent create required the client to re-map the group i-number in the group i-table and apply changes to the user's copy of the directory.

The clients took an average of 4.60 s and 4.26 s on SUNDR/NVRAM and NFS3 respectively. For comparison, we also ran the benchmark concurrently in two separate directories, which required an average of 2.94 s for SUNDR/NVRAM and 4.05 s for NFS3. The results suggests that while contention incurs a noticeable cost, SUNDR's performance even in this case is not too far out of line with NFS3.

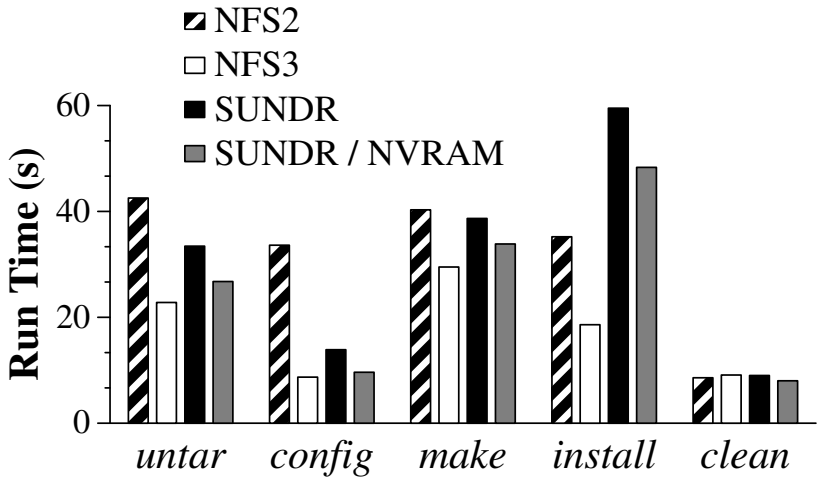


Figure 3.11: Installation procedure for `emacs_20.7`

3.6.3.3 Real workloads

Figure 3.11 shows SUNDR's performance in untaring, configuring, compiling, installing and cleaning an emacs 20.7 distribution. During the experiment, the SUNDR client sent a total of 42,550 blocks to the block server, which totaled 139.24 MB in size. Duplicate blocks, which *bstor* discards, account for 29.5% of all data sent. The client successfully `decrefed` 10,747 blocks, for a total space savings of 11.3%. In the end, 25,740 blocks which totaled 82.21 MB went out to permanent storage.

SUNDR is faster than NFS2 and competitive with NFS3 in most stages of the Emacs build process. We believe that SUNDR's sluggish performance in the *install* phase is an artifact of our implementation, which serializes concurrent **xfs** upcalls for simplicity (and not correctness). Concurrent **xfs** upcalls are prevalent in this phase of the experiment due to the *install* command's manipulation of file attributes.

Figure 3.12 details the performance of the *untar* phase of the Emacs build as client concurrency increases. We noted similar trends for the other phases of the build process. These experiments suggest that the scalability SUNDR exhibited in the LFS small file

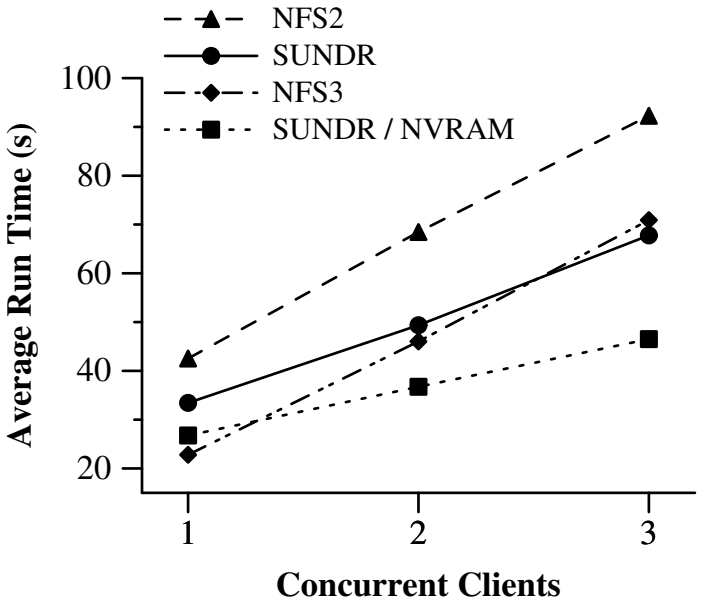


Figure 3.12: Concurrent *untar* of `emacs_20.7.tar`

benchmarks extends to real file system workloads.

3.6.3.4 CVS on SUNDR

We tested CVS over SUNDR to evaluate SUNDR's performance as a source code repository. Our experiment follows a typical progression. First, client A imports an arbitrary source tree—in this test `groff-1.17.2`, which has 717 files totaling 6.79 MB. Second, clients A and B check out a copy to their local disks. Third, A commits `groff-1.18`, which affects 549 files (6.06 MB). Lastly, B updates its local copy. Figure 3.1 shows the results.

SUNDR fares badly on the commit phase because CVS repeatedly opens, memory maps, unmaps, and closes each repository file several times in rapid succession. Every open requires an iteration of the consistency protocol in SUNDR, while FreeBSD's NFS3 apparently elides or asynchronously performs `access` RPCs after the first of several closely-

Phase	SUNDR	SUNDR NVRAM	NFS3	SSH
Import	13.0	10.0	4.9	7.0
Checkout	13.5	11.5	11.6	18.2
Commit	38.9	32.8	15.7	11.5
Update	19.1	15.9	13.3	11.5

Table 3.1: Run times for CVS experiments (in seconds).

spaced *open* calls. CVS could feasibly cache memory-mapped files at this point in the experiment, since a single CVS client holds a lock on the directory. This small change would significantly improve SUNDR's performance in the benchmark.

Chapter 4

BFT2F

We describe the design and implementation of SUNDR in the previous chapter. SUNDR is concerned with guaranteeing fork consistency without trusting any of its servers. However, because of this assumption, SUNDR also has a limitation: it can only guarantee ideal consistency if none of its servers fails in any way. When the servers fail benignly (i.e., in a fail-stop fashion), SUNDR loses liveness; When servers fail in a Byzantine way, SUNDR merely promises fork consistency in this worst case.

However, failure must be considered the common case in today's large scale storage systems, not the exception [41]. To deliver continuous service, one has to provide some sort of fault tolerance. Replication is one of the most common approaches. Further complicating the matter, masking some servers' malicious (Byzantine) behavior requires significantly more complicated replication and agreement protocols. The resulting replication systems are termed *Byzantine Fault Tolerant* systems (BFT) in the literature, and have received extensive study in the past 20+ years [51, 21, 55, 25, 22].

Unfortunately, all BFT systems require a "strong" assumption that at least some predetermined fraction of servers are honest. For example, the highest fraction of failures that an

asynchronous BFT system can survive without jeopardizing linearizability or liveness is f out of $3f + 1$ servers. The reason is that asynchronous communication makes it impossible to differentiate slow replicas from failed ones. To progress safely with f unresponsive replicas, a majority of the remaining $2f + 1$ responsive replicas must be honest.

In the smallest configuration, a BFT system requires four servers and remains secure as long as no two are compromised. These numbers limit BFT's effectiveness against software vulnerabilities, because it is hard to run four machines with entirely uncorrelated software flaws. For example, while Windows and Linux seldom have the same exploitable bugs, a number of flaws have affected both Linux and other Unix-like operating systems. BSD variants share a lot of code and are even more likely to experience correlated vulnerability.

Making matters worse, the security of today's best-known BFT algorithms fails completely given $f + 1$ compromised nodes. For example, an attacker who compromises two out of four replicas can return arbitrary results to any request by any client, including inventing past operations that were never requested by any user or rolling back history to undo operations that were already revealed to clients. In fact, it is not *a priori* harder to compromise two out of four different replicas than one secure server. Whether existing BFT algorithms exacerbate or mitigate the effects software bugs depends on the details of a specific deployment.

Fortunately, linearizability and total failure are not the only options. The goal of this chapter is to improve security when more than f out of $3f + 1$ servers in a replicated state system fail. Specifically, we explore the fork* consistency model introduced in section 2.2 to bound system's behavior when between $f + 1$ and $2f$ replicas have failed. When $2f + 1$ or more replicas fail, it is unfortunately not possible to make any guarantees without simultaneously sacrificing liveness for cases where fewer than f replicas fail.

Our proposed BFT2F protocol provides exactly the same linearizability guarantee as

PBFT when less than $1/3$ of replicas have failed. When at least $1/3$ but less than $2/3$ have failed, two outcomes are possible. (1) The system may cease to make progress—In other words, BFT2F does not guarantee liveness when $1/3$ of replicas or more are compromised. Fortunately, for most applications, denial of service is much less harmful than arbitrary behavior. (2) The system may continue to operate and offer fork* consistency, which again for many applications is considerably preferable to arbitrary behavior.

The rest of the chapter is organized as follows. We start by giving a brief survey of the background of BFT systems in section 4.1. Then we present BFT2F, our extension of Castro-Liskov's PBFT in section 4.2. We prove the correctness of the protocol in section 4.3, and finally evaluate the system in section 4.5.

4.1 Background

BFT systems are inspired by *The Byzantine General's problem* [51]. The Byzantine General's problem states that several generals encircle an enemy city, and need to formulate a common plan of attack. The generals communicate with each other by messenger. However, some of the generals are *traitorous*, and try to prevent *loyal* generals from reaching an agreement.

The problem is significantly complicated by the fact that traitorous generals could behave inconsistently. For instance, if three generals (G0, G1, G2) are voting, and G2 is serving as the commanding general sending an order to his lieutenants G0 and G1. The malicious general G2 sends G0 the same vote as his, while sending G1 the same vote as hers (not necessary the same as the G0's.). Therefore, G0 and G1 do not reach an agreement, if they vote differently. The problem must have algorithms that satisfy the following conditions.

1. All loyal lieutenants obey the same order.

2. If the commanding general is loyal, then every loyal lieutenant obeys his order.

It has been shown that there is no protocol that could achieve agreement with one third or more of the generals traitorous. Furthermore, two algorithms [51], one for oral messages and the other for signed messages, were presented given no more than one third of the generals are traitorous.

4.1.1 PBFT in a nutshell

Most BFT algorithms require expensive public-key cryptography to authenticate messages. Consequently, they perform at least an order of magnitude slower than those do not tolerate Byzantine failures. Castro and Liskov propose a first *practical* algorithm – PBFT [25] – that makes BFT system use feasible in real-world scenarios.

PBFT operates for $3f + 1$ replicated state machines in an asynchronous network, where f is the maximum number of faulty replicas the algorithm can tolerate. PBFT evolves over a series of *views* [65]. Within each view, one replica is designated the *primary*. A primary is responsible for assigning a specific order to operations in the system. A client sends a request message m to the primary. Then the algorithm passes through three phases: *pre-prepare*, *prepare*, *commit*. During the *pre-prepare* phase, the primary gives m a sequence number n, and sends m with assigned sequence number n to all other backup replicas. During the *prepare* phase, all replicas inform others about their receipt of m with n. When receiving $2f + 1$ such matching receipts, one replica can obtain the knowledge K [43] that there must be at least $f + 1$ non-faulty replicas agreeing m with n, therefore there won't be another $f + 1$ non-faulty replicas accepting different m' with n within the same view. In the *commit* phase, one sends K to all other peers. Consequently, when receiving $2f + 1$ such

Ks, they can be combined to form a proof of a common knowledge CK [43]: the replica knows that at least $f+1$ non-faulty replicas know K. This process ensures m with n would be passed to the new view, whenever the current primary becomes faulty. Once the *commit* phase finishes, replicas can safely execute m, and send the reply r back to the client. The client accepts r, after receiving $f+1$ matching replies.

When the primary is detected as faulty, a view-change is triggered to elect a new primary in a new view. The new primary is responsible for (re)generating a new `pre-prepare` message for each message m that has passed through the *prepare* phase in the old view, or a null message, if a sequence number is within the gap, but without the corresponding message.

PBFT uses a lot of optimization to make it fast [27]. First, to authenticate a message, PBFT appends a vector of MACs, one for each different recipient, avoiding the use of much more expensive public-key cryptography in the common case. Second, only one replica needs to send back the reply r, while others can just send *digest replies*. Third, a replica can *tentatively execute* a request after `prepare` phase, once all previous requests of lower sequence numbers have been committed. A client accepts the corresponding tentative reply on receiving $2f+1$ matching ones.

4.2 BFT2F Algorithm

In this section, we present BFT2F, an extension of the original PBFT protocol [25], which guarantees fork* consistency when more than f, but not more than $2f$ out of $3f+1$ servers fail in an asynchronous system.

Like PBFT, BFT2F uses a three-phase commit protocol [75]. The three phases are *pre-prepare*, *prepare*, and *commit*. In view number $view$, replica p acts as the primary if

$p = view \bmod 3f + 1$. The primary replica assigns a new sequence number to each client operation.

4.2.1 BFT2F Variables

Below, we describe major new variables and message fields introduced in BFT2F. We use superscripts to denote sequence number, e.g. msg^n refers to the message with sequence number n. We use subscripts to distinguish variables kept at different nodes.

Hash Chain Digest (HCD): A HCD encodes all the operations a replica has committed and the commit order. A replica updates its current HCD to be $HCD^n = D(D(msg^n) \circ HCD^{n-1})$ upon committing msg^n, where D is a cryptographic hash function and $\circ$ is a concatenation function. Replicas and clients use HCDs to verify if they have the same knowledge of the current system state.

Hash Chain Digest History: To check if a replica's current knowledge of the system state is strictly fresher than another replica's, but not forked. each replica keeps a history of its past HCDs. We denote the past HCD entry at replica p upon processing msg^n as $T_p[n]$.

Version Vector: Every node i represents its knowledge of the system state in a version vector V_i. The version vector consists of $3f + 1$ entries, one for each replica. Each entry has the form $\langle r, view, n, HCD^n \rangle_{K_r{}^{-1}}$, where r is the replica number, $view$ is the view number, n is the highest sequence number that node i knows that replica r has committed, and HCD^n is r's HCD after n operations. The entry is signed by node r's private key $K_r{}^{-1}$. We denote replica r's entry in V_i by $V_i[r]$, and the corresponding HCD field as $V_i[r].HCD$.

Let $V[p]$ and $V'[p]$ be two version entries for the same replica p. We say $V[p]$ *dominates* $V'[p]$ if the sequence number in $V[p]$ is greater than that in $V'[p]$ or both entries have the same view number, sequence number, and HCDs. We say V_j *supersedes* V_i, iff for all replicas p, $V_j[p]$ dominates $V_i[p]$. Upon receiving a different version vector, a node always updates its current version vector by replacing each entry with the dominating new entry.

4.2.2 BFT2F Node Behavior

In describing BFT2F, we borrow heavily from PBFT [25]. However, we point out two major differences between BFT2F and PBFT. First, unlike in PBFT, BFT2F replicas do not allow out of order commits. This requirement does not pose much overhead as replicas must execute client operations in increasing sequence numbers anyway. Second, BFT2F requires clients to wait for at least $2f + 1$ *matching* replies before considering an operation completed, as opposed to the $f + 1$ matching replies required in PBFT.

Client Request Behavior

A client a sends a request for operation $\langle \texttt{request}, op, ts, a, V_a \rangle_{{K_a}^{-1}}$ to the primary replica, where ts is the timestamp, V_a is client a's version vector and the message is signed by a's private key ${K_a}^{-1}$.

Server Behavior

Upon receiving a request message ($msg = \langle \texttt{request}, op, ts, a, V_a \rangle_{{K_a}^{-1}}$) from client a, the primary replica p authenticates the message and assigns it a sequence number n. It then multicasts a `pre-prepare` message $\langle \langle \texttt{pre-prepare}, p, view, n, D(msg^n) \rangle_{{K_p}^{-1}}, msg^n \rangle$ to all other replicas.

Upon receiving a `pre-prepare` message, replica q first checks that it has not accepted the same sequence number n for a different message msg'^n in the same view $view$. Replica

q also verifies that its knowledge of the system supersedes client a's by checking that all HCDs in a's version vector V_a have an existing matching entry in q's HCD history T_q. Replica q then multicasts a `prepare` message $\langle \texttt{prepare}, q, view, n, D(msg^n) \rangle_{K_q{}^{-1}}$ to all other replicas.

A replica u tries to collect $2f$ matching `prepare` messages (including one from itself) with the same sequence number n as that in the original `pre-prepare` message. When it succeeds, we say replica u has *prepared* the request message msg^n. Unlike PBFT, u does not commit out of order, i.e. it enters the *commit* phase only after having committed all requests with lower sequence numbers. To start committing, replica u updates its latest HCD to $HCD^n = D(msg^n \circ HCD^{n-1})$, adds (n, HCD^n) to its HCD history T_u, and multicasts a `commit` message $\langle \texttt{commit}, u, view, n, HCD^n \rangle_{K_u{}^{-1}}$ to all other replicas.

When replica w receives a `commit` message from replica u, it updates the entry for u in its current version vector,$V_w[u]$, to $\langle u, view, n, HCD^n \rangle_{K_u{}^{-1}}$. Replica w commits msg^n when it receives $2f + 1$ matching `commit` messages (including its own) for the same sequence number n and the same HCD (HCD^n).

Replica w executes the operation after it has committed the corresponding request message msg^n. It sends a reply message to the client containing the result of the computation as well as its current version vector V_w. Since w has collected $2f + 1$ matching `commit` messages, we know that these $2f + 1$ replicas are in the same fork set, up to sequence number n.

Behavior of Client Receiving Replies

A reply from replica w has the format, $\langle \texttt{reply}, view, ts, a, w, res, V_w \rangle_{K_w{}^{-1}}$, where $view$ is the current view number, ts is the original request's timestamp, res is the result of executing the requested operation and V_w is w's version vector. A client considers an operation completed after accepting at least $2f+1$ replies each of which contains the same ts,

res and a valid version vector V. A valid version vector contains $\geq 2f + 1$ entries with the same sequence number n and HCD^n. This check ensures the client only accepts a system state agreed upon by at least $2f + 1$ replicas. Therefore, if no more than $2f$ replicas fail, the accepted system state reflects that of at least one correct replica. Client a also updates the entry for replica w in its own version vector, $V_a[w]$, to $V_w[w]$ obtained from the $2f + 1$ replying replies.

To deal with unreliable communication, client a starts a timer after issuing a request and retransmits if it does not receive the required $2f + 1$ replies before the timer expires. Replicas discard any duplicate messages.

4.2.3 Garbage Collection

Each replica keeps all received messages in a *message log*. Replicas can use the same algorithm as in PBFT to truncate their message logs while preserving fork* consistency when no more than $2f$ replicas fail.

If replicas wish to be able to rewind and restore the system state to an ideally consistent state upon detecting a forking attack, they have to truncate message logs more conservatively than PBFT nodes. In particular, replica r can only safely discard all the messages whose sequence number is no more than the lowest sequence number, n_{low} in V_r. This ensures *all* replicas have executed the same sequence of operations prior to the head of the message log. Unfortunately, this implies that one faulty replica can cause message logs at correct replicas to grow indefinitely. To truncate a HCD history, replica r can delete all entry whose sequence number is less than n_{low}.

4.2.4 Server View Change

The view change algorithm differs in two aspects from PBFT [25]. The first lies in view-change message. A replica r triggers a timeout by sending a `view-change` message $\langle \texttt{view-change}, view+1, n, HCD^n, P\rangle_{K_r{}^{-1}}$ to all other replicas. Here (n, HCD^n) is r's highest committed sequence number - HCD pair. P is a set of sets P_m of `pre-prepare` messages, for each prepared message m with sequence number higher than n, and $2f$ corresponding matching `prepare` messages.

When the primary p in the new view $view+1$ receives $2f$ valid `view-change` messages, it multicasts a `new-view` message $\langle \texttt{new-view}, view+1, V, O\rangle_{K_p{}^{-1}}$. V is the set containing $2f+1$ valid `view-change` messages sent by backup replicas (including p's own). O is a set of `pre-prepare` messages constructed as below:

1. p determines *min-s* as the lowest sequence number in all `view-change` messages in V, and *max-s* as the highest sequence number in prepared messages in V.

2. For each sequence number n between *min-s* and *max-s*, p either (1) constructs a `pre-prepare` message in the new view, if a P_m in one of the view-change messages corresponding to the sequence number n, or (2) constructs a special *null* request $\langle\langle \texttt{pre-prepare}, p, view+1, n, D(null)\rangle_{K_p{}^{-1}}, null\rangle$ to fill in the sequence number gap.

A backup replica u in the new view accepts the `new-view` message if u validates the message. In particular, it needs to check whether the $(\textit{min-s}, HCD^{\textit{min-s}})$ pair matches that in its hash chain digest history T_u. If it does not exist in T_u, u sends a state transfer request to all other replicas, as will be discussed in the rest of the section. It also verifies O is properly constructed by the step above. If so, then it sends a `prepare` message for each message in O, and proceeds normally in the new view.

When there are no more than f faulty replicas, the above algorithm is essentially the same as PBFT. When more than f, but no more than $2f$ replicas fail, fork* consistency would not be violated across view changes: since *min-s* is picked as the lowest committed sequence number of $2f + 1$ replicas in V, this guarantees that these $2f + 1$ replicas were in the same fork set up to *min-s*. Subsequently, for each `pre-prepare` message m in O, it can not be used to join already forked replicas, because the HCD fields in the `commit` messages for m in the new view would disagree, preventing m from committing at all of them.

The second difference is how to transfer state S from others upon missing some messages. In PBFT, replica r just fetches the missing state from $2f+1$ other replicas. However, in BFT2F, when more than f replicas fail, malicious servers could join r's forked state with that of the non-faulty ones which have been forked along a different path (i.e., in a different fork set) in the replying replicas. Instead, BFT2F requires replying replicas to present all the messages from r's current committed sequence number to S's (highest) sequence number.[1] When getting $2f + 1$ such matching replies, r verifies whether there exists a valid state transition path from its current state to S by executing these messages tentatively. If such a path exists, r accepts S as its new state.

In the worst case, when $2f$ replicas fail maliciously, up to $f + 1$ view changes might succeed concurrently, leading to $f + 1$ fork sets. There are two cases to consider: (1) (Non-faulty) replica r has the state (represented as HCD) that is agreed upon by $2f + 1$ replicas in the new view, in which case r is assured to be in the same fork set as those replicas, whose `view-change` messages are listed in V, across the view change. (2) Or r misses S, since it can only update its state to S by applying all the missing messages in the same order. If it reaches the same S, then it means that r was, and is still in the same fork set with those $2f + 1$ replying replicas up till state S. In either case, fork* consistency is preserved across

[1] This requires nodes to take the conservative garbage collection approach.

view changes. Finally, a *prepared* message in the old view is not guaranteed to propagate to every new view, or maybe none of the new views. Still, this only affects liveness, not the safety guarantee of fork* consistency. BFT2F relies on clients to re-issue such messages in the new view to make progress.

4.2.5 An Example

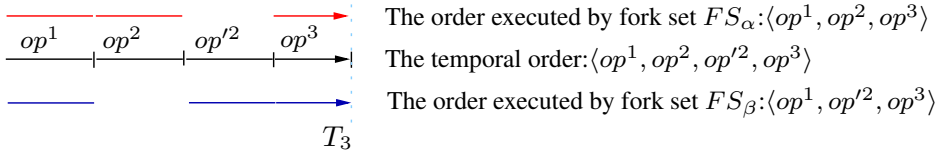
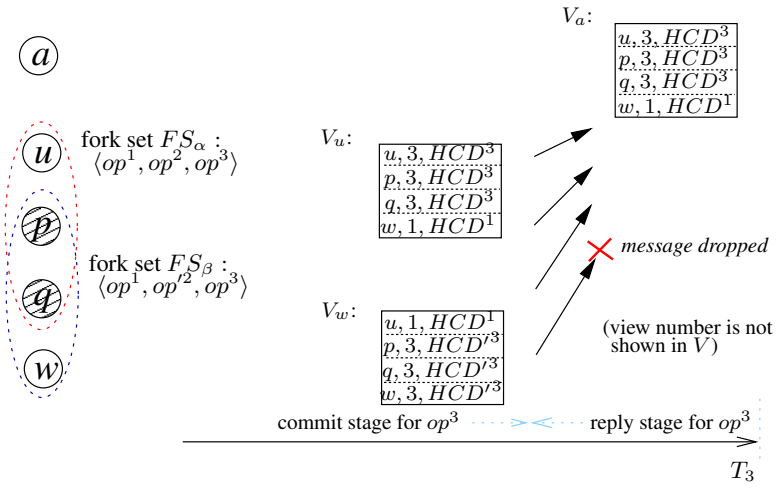


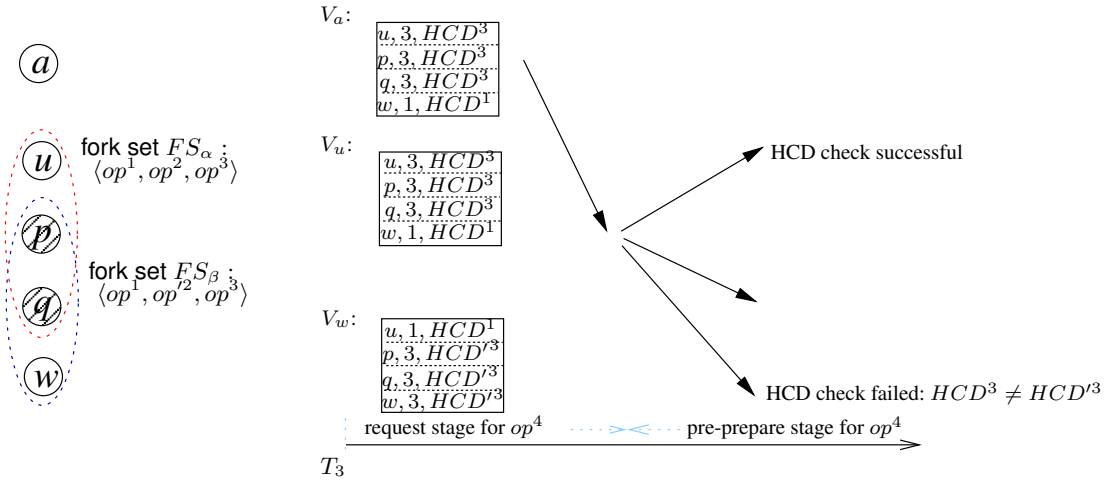
Figure 4.1: An example of two forked result lists. The timeline in the middle shows the result list that would have been executed by a non-faulty (never-forked) system. The timeline above it shows in a forked system, one fork set has executed a forked result list, which does not reflect operation op'^2. The timeline below it shows another forked result list that omits operation op^2.

We demonstrate the join-at-most-once property of BFT2F during normal case operations using a simple example. Similar to the example in section 2.2, the system consists of four replicas u, p, q, w with p being the primary in the current view and two replicas p, q being malicious.

Here we explain the intuition that join-at-most-once property can be achieved with a one-round protocol. Suppose if the system is forked into two fork sets just before client c issues two operations, op^1 and op^3, then op^1 might show up in two fork sets, because it is indistinguishable to the non-faulty replicas in both fork sets from the case in which the system is never forked. But op^3 will reflect the result of op^1, which commits c to compatibility with only one of the fork sets. Thus, op^3 will only appear in the fork set from which c gets the reply for op^1.



(a)



(b)

Figure 4.2: An example of join-at-most-once property. Suppose op^3 has been used to join two forked result lists as in Figure 4.1. Diagram (a) shows the commit and reply stages for operation op^3. Since the result lists of the two fork sets are already forked, the HCD field of sequence number 3 for non-faulty replicas in different fork sets is different: HCD^3 for u and HCD'^3 for w respectively. Client a accepts the reply from fork set FS_α, and updates its V_a accordingly. Notice a cannot receive the reply from (non-faulty) replicas (e.g., w) in FS_β simultaneously, without w being able to detect the system faulty. Diagram (b) shows the impossibility of any future operation by c appearing in both fork sets' result lists again. For example, if the next operation op^4 reaches non-faulty replicas in both fork sets, then the HCD check in pre-prepare stage could only succeed at one of them.

Now we consider a detailed example where client a issues the first (op^1) and fourth operation (op^3) and some other clients issue the second (op^2) and third (op'^2) operation. The result list $\langle op^1, op^2, op'^2, op^3 \rangle$ would have reflected the order assigned in an otherwise non-faulty system, as shown in Figure 4.1. Replica p shows all replicas the first operation from client a and assigns sequence number 1 to the operation (op^1). Subsequently, p only shows the second operation (op^2) to u and the third operation (op'^2) to w, but it assigns both operations with the same sequence number 2. As a result, two fork sets FS_α and FS_β are formed, where FS_α contains u which has seen $\langle op^1, op^2 \rangle$ and FS_β contains w which has seen $\langle op^1, op'^2 \rangle$. Replica p then manages to join two forked result lists for the first time with the same operation op^3; the two result lists become $\langle op^1, op^2, op^3 \rangle$ and $\langle op^1, op'^2, op^3 \rangle$, respectively. Suppose client a gets the required $2f + 1 = 3$ replies for op^3 from the fork set $FS_\alpha = u, p, q$. Consequently, client a's version vector contains $HCD^3 = D(op^3 \circ D(op^2 \circ D(op^1)))$, while replica w in FS_β has a different version vector V_w containing $HCD'^3 = D(op^3 \circ D(op'^2 \circ D(op^1)))$ (shown in Figure 4.2(a)). Thereafter, when malicious servers try to join the two forked result lists again with a's future operation, say, op^4, the HCD^3 included in a's request would conflict with that in w's HCD history, because sequence number 3 is associated with a different HCD: $HCD^3 \neq T_w[3]$ (shown in Figure 4.2(b)).

4.3 Proof Sketch

4.3.1 Normal case operations

Sublist operation Lemma: Every result list is a sublist of L^{all}, which contains all completed operations in a specific order.

Proof Sketch (Trivial): Suppose V is the version vector contained in one of the replies for

the completed operation op. According to the constraint test for a client to accept the reply of an operation: at least $2f + 1$ entries in V must have the same sequence number n, and HCD^n. Since at least one of the replicas in these entries is non-faulty, the honest replica's HCD^n should encode the result list it has executed. Without loss of generality, we assume it is $L = \langle op^{\alpha 1}, op^{\alpha 2}, ..., op^{\alpha i}, op \rangle$ ($HCD^n = D(op \circ ... \circ D(op^{\alpha 2} \circ D(op^{\alpha 1})))$). Therefore, op reflects the result list L. ∎

Self-consistency Lemma: Each honest user's operation appears in all his subsequent operations' result lists.

Proof Sketch: Suppose client c issues an operation op_c^i before op_c^j. First we prove that op_c^i appears in c's next operation op_c^{i+1}'s result list.

Since at least $2f + 1$ replicas' `reply` messages are required for client c to complete operation op_c^i, suppose these $2f + 1$ replying replicas are in the fork set FS, the sequence number is n and HCD HCD^n. Then the version vector $V_{c_i,post}$ in the request for c's next operation op_c^{i+1} must have $2f + 1$ matching entries with sequence number n and HCD HCD^n. In order for c to complete operation op_c^{i+1}, at least one of the replying replicas for op_c^{i+1} must be non-faulty, let it be w. When w checks the `pre-prepare` message for op_c^{i+1}, it must ensure that the (n, HCD^n) pair also appears in T_w. If it succeeds, this means w was also in FS at least until n. So the result list for operation op_c^{i+1} would contain op_c^i.

Then by induction, op_c^i would appear in op_c^j's result list, provided $j > i$. ∎

Join-at-most-once Lemma: If two result lists contain two operations op' and op from the same honest client, and op' precedes op, then the two result lists are identical up to op'.

Proof Sketch: We prove it by contradiction. Assume two forked result lists $L_\alpha = \langle op^{\alpha 1}, op^{\alpha 2}, ..., op^{\alpha i}, op_c^i, ..., op_c^j \rangle$, $L_\beta = \langle op^{\beta 1}, op^{\beta 2}, ..., op^{\beta j}, op_c^i ..., op_c^j \rangle$ are joined

twice by operation op_c^i and op_c^j from client c ($j > i$), and $L^{\alpha 1-\alpha i} \neq L^{\beta 1-\beta j}$. [2] Assume the fork set that generates L_α is FS_α, with u as the non-faulty replica in FS_α; and the fork set for L_β is FS_β, with w as the non-faulty replica.

Suppose client c gets the reply for operation op_c^i from fork set FS_α, and $V_{c_i,post}$ is c's version vector after receiving the reply for op_c^i (Notice c can not receive replies for operation op_c^i from both fork sets.). When c issues operation op_c^j (not necessary the next operation to op_c^i), suppose the version vector included in the request is $V_{c_j,pre}$. Then by the self-consistency lemma, the $2f+1$ matching entries in $V_{c_j,pre}$ encode a result list L', which has the prefix $\langle op^{\alpha 1}, op^{\alpha 2}, ..., op^{\alpha i}, op_c^i \rangle$. Assume these matching entries have the sequence number n_c, HCD HCD^{n_c} (HCD^{n_c} encodes the result list L'). Assume replica x is one of the non-faulty replicas that generate L'.

Suppose that w's current sequence number (before committing request op_c^j) is n_w, and also at least $2f+1$ entries (in fork set FS_β) in V_w have the sequence number n_w, since this is the prerequisite for w to commit w's last executed operation. When w receives the `pre-prepare` message for op_c^j from the primary, it checks that the sequence number and HCD pair in each entry in $V_{c_j,pre}$ matches those in T_w. If $n_c > n_w + 1$, then w would not commit the operation op_c^j as its next operation, which violates the assumption that w executes op_c^j. Then we must have $n_c \leq n_w + 1$, which means w has executed the operation for sequence number n_c, and has the corresponding HCD in its hash chain digest history T_w. Because x is the non-faulty replica from which c gets the reply for its previous operation to op^i, $V_{c_j,pre}[x].HCD = HCD^{n_c}$. This implies x evolves over result list L' whose prefix is $\langle op^{\alpha 1}, op^{\alpha 2}, ..., op^{\alpha i}, op_c^i \rangle$. But $T_w[n_c].HCD$ equals w's HCD at sequence number n_c, up to the point where w evolves over $\langle op^{\beta 1}, op^{\beta 2}, ..., op^{\beta j}, op_c^i, ... \rangle$. Because $L^{\alpha 1-\alpha i} \neq L^{\beta 1-\beta j}$, and cryptographic hash function D is collision-resistant, so $HCD^{n_c} \neq T_w[n_c].HCD$, and

[2] We use L^{i-j} to denote *consecutive* sublist from index i to j.

the check would fail. Thus, w could not commit and execute operation op_c^j, which contradicts the assumption.

∎

Theorem: BFT2F satisfies fork* consistency.

Proof Sketch: Immediately follows from three lemmas above. ∎

4.3.2 Servers view change

In the section above, we sketched the proof that BFT2F achieves fork* consistency during normal case operations. For the case of servers performing view change, since the prepared messages (O) issued by the new primary (in the `new-view` message) also go through the same three-phase protocol, the result sublist generated after the view change (as shown in Figure 4.3, SL_{v+1}) also has fork* consistency in the new view. [3] Now we only need to show that when result lists consist of operations completed in both the old view and the new view, fork* consistency is still preserved.

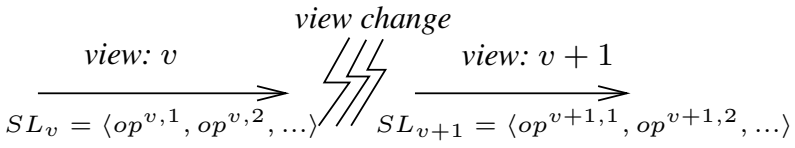


Figure 4.3: From section 4.3.1, either SL_v or SL_{v+1} delivers fork* consistency, respectively, within its own view. We only need to show, when considering completed operations from both sub result lists, it also preserves fork* consistency.

Sublist operation Lemma: Every result list is a sublist of L^{all}, which contains all com-

[3] Result sublist here means that it only contains operations completed in a specific view.

pleted operations in a specific order.

Proof Sketch (Trivial): It does not matter whether there has been a view change or not. At least one non-faulty replying replica for operation op ensures that the result list for op is some sublist of L^{all}. ∎

Self-consistency Lemma: Each honest user's operation appears in all his subsequent operations' result lists.

Proof Sketch: As in the corresponding proof for the normal case, here we need to consider the case where op_c^i is an operation completed in view v, and op_c^{i+1} in view $v+1$.

Regardless whether op_c^{i+1} is an operation passed in O from the old view, or is a brand new operation submitted in the new view, the request of op_c^{i+1} contains the version vector $V_{c_i,post}$ that client c updates from the reply of op_c^i in the old view. As usual, $V_{c_i,post}$ would contain at least $2f+1$ matching entries of the same sequence number and HCD. Without loss of generality, suppose it is (n, HCD^n). Then, in order for op_c^{i+1} to complete in the new view, at least one non-faulty replica w needs to check (n, HCD^n) is in its T_w, which guarantees that the result list for op_c^{i+1} would contain operation op_c^i. ∎

Join-at-most-once Lemma: If two result lists contain two operations op' and op from the same client, and op' precedes op, then the two result lists are identical up to op'.

Proof Sketch: Again, similiarily, we consider the case where op_c^i is an operation completed in view v, and op_c^j in view $v+1$. Then the proof follows as above. ∎

4.4 Discussion

Ideal consistency requires *quorum intersection*; any two quorums should intersect in at least one non-faulty replica. Fork* consistency requires only *quorum inclusion*; any quorum should include at least one non-faulty replica. Because there is at least one correct replica witness, quorum inclusion ensures that no legitimate operation is "lost" in the system. Like PBFT, BFT2F achieves ideal consistency when no more than f out of $3f + 1$ replicas fail, additionally, BFT2F provides fork* consistency when no more than $2f$ replicas fail. Unfortunately, BFT2F cannot guarantee *quorum availability*, i.e., liveness, with more than f faulty replicas.

One might wonder if there exist protocols that guarantee fork* consistency with even larger numbers of failures, e.g. up to $3f$ failures. The answer is yes, in fact, BFT2F can be easily modified to require $3f + 1$ matching replies instead of $2f + 1$ ones, achieving fork* consistency with up to $3f$ failures at the cost of sacrificing liveness when there is any failure in the system. Actually, we can conceptually view SUNDR as a system that is able to tolerate any number of failures in the system, yet does not guarantee liveness. System operators that care more about security than availability might tune the protocol to guarantee fork* consistency up to $3f$ failures.

Both SUNDR and BFT2F use version vectors to represent a node's knowledge of the current system state, with one major difference. SUNDR's version vector has one entry per client and BFT2F's version vector has one entry per server. This difference brings BFT2F two advantages. First, BFT2F's version vector is more efficient, as a distributed system typically has many more clients than servers. Second, in both protocols, a node does not see any updates for version entries of nodes in a different fork set. In SUNDR, a stagnant entry might also indicate that a client has been offline or has not performed any operations:

both are legitimate client actions. In contrast, as servers always remain online and process user operations, a stagnant server version entry in BFT2F is a good indication that the system might have forked.

4.5 Performance

We built a prototype implementation of the BFT2F algorithm on FreeBSD 4.11, based on our ported version of the BASE [72] library.

4.5.1 Implementation

BFT2F's additional guarantee over BFT [25] to allow detection of past consistency violations comes at the expense of much increased use of computationally expensive public key signatures instead of the symmetric session-key based Message Authentication Codes (MACs) used by PBFT. We use NTT's ESIGN with key length of 2048 bits in BFT2F. On a 3 GHz P4, it takes 150 μs to generate signatures, 100 μs to verify. For comparison, 1280 bits Rabin signatures take 3.1 ms to generate, 27 μs to verify.

All experiments run on four machines; three 3.0 GHz P4 machines and a 1.4 GHz Athlon machine. Clients run on a different set of 2.3 GHz Celeron machines. All machines are equipped with 1-3 GB memory and connected with each other via a 100 Mbps switch.

4.5.2 Micro benchmark

Our micro benchmark is the built-in *simple* program in BASE, which sends a *null* operation to servers and waits for the reply. Authenticating each message with public keys has higher cost over using MAC, resulting in BFT2F having higher latency (2.06 ms) than BASE

(1.10 ms) for each request.

4.5.3 Application-level benchmark

We modify NFS to run over BFT2F, and compare it to the native NFSv2, NFS-BASE running on 4 servers, and SUNDR running on 1 server. The evaluation takes five phases: (1) copy a software distribution package `nano-1.2.5.tar.gz` into the file system, (2) uncompress it in place, (3) untar the package, (4) compile the package, (5) clean the build objects.

	NFSv2 (no replication)	NFS-BASE	NFS-BFT2F	SUNDR
Phase 1	0.302	1.296	1.351	0.135
Phase 2	1.169	5.371	5.671	0.690
Phase 3	2.815	7.724	8.759	3.260
Phase 4	4.151	7.107	7.332	6.714
Phase 5	0.101	0.175	0.220	0.157
Total	8.538	21.673	23.333	10.956

Table 4.1: Performance comparison of different file system implementation (in seconds).

As Table 4.1 shows, the application-level performance slowdown in NFS-BFT2F relative to NFS-BASE is much less than that observed in our micro benchmark. This is because the high cost of public key operations is amortized over that of processing requests. Both BFT2F and NFS-BASE achieve much lower performance than NFSv2 and SUNDR, reflecting the cost of replication.

Chapter 5

Related Work

5.1 Cryptographic file systems

Protecting data with user-level tools like `crypt` can be a cumbersome burden for users. For example, one might forget to delete the cleartext after encrypting it; each access to the protected file requires one to type a password. Conversely, encryption can be also performed at a lower-level: on-disk encryption hardware encrypts data before it reaches physical media. Though transparent to end users, low-level disk encryption lacks flexible control: many users might have to use the same key, and sharing among users could also be difficult. CFS [20] is the first system that enforces security at file systems level, while enjoying the advantages of both.

CFS associates each protected directory with a key. Each file stored in the directory is encrypted on the way to the underlying block driver, and decrypted when read back. Cleartext is never left on the underlying disks, or remote file servers, which protects the file content in case the disk is stolen.

TCFS [28] addresses some practical drawback of CFS. It generates an independent

key for each different block, and stores it in the file inode. TCFS also improves users' experience by only requiring them to remember one passphrase for each protected file system.

Both CFS and TCFS are mostly suitable for the *single-writer and single-reader* scenario. The limitation arises in several aspects: (1) to enable multi-writer or multi-reader sharing, the keys to encrypt file data need to be shared. However, there is no easy way to distribute or revoke encryption keys scalably. (2) Most importantly, with untrusted servers, it is impossible to guarantee correct consistency semantics, which is critical for network file systems.

Plutus [47] and Chefs [38] address the first limitation aforementioned. Plutus groups related files into *filegroups*, and uses one *file lockbox* to manage all (symmetric) keys used to encrypt file data. By doing so, Plutus significantly reduces the number of keys that need to be exchanged when sharing files. Plutus/Chefs also introduce a novel concept, *key rotation*, to revoke access rights lazily: a file to which some users have been revoked access does not get (re)encrypted immediately until new modifications occur. Users with the latest key can derive all old versions of the key, but not vice versa.

SUNDR lacks some useful features of the above systems, while it mainly tries to address the second difficulty: how best a system can preserve its consistency guarantees with an untrusted server, which none of the previous systems have addressed.

SFS [59] eases key management by separating it from file systems security. A file server embeds its public key into a *self-certifying* file pathname. Each SFS pathname has the following format: `/sfs/@location,hostid/pathname`, where `location` is the hostname of the server, and `hostid` is the SHA-1 hash of the server's public key. Once a client has an SFS pathname, he/she can be assured to connect to the server securely, without having to commit to any particular key management policy ahead of time. However, SFS does rely on

honest servers, and handles corrupted servers by propagating self-authenticating revocation certificates in a best-effort way.

SFSRO [39] is a read-only version of SUNDR. Aiming at distributing content with untrusted hosts, SFSRO takes the same Merkle hash tree approach as in SUNDR. Content distributors sign the root of the tree with their private keys. Untrusted hosts cannot tamper with any of these blocks, since the root block is protected by the signature, and all lower-level blocks are protected by the cryptographic hashes. Duchamp [33], TDB [54], OceanStore [19], and Pond [71] all have made use of Merkle hash trees for comparing data or checking the integrity of part of a larger collection of data.

Staging data on untrusted surrogates has also been proposed in [36]. Read-only data is prefetched to a nearby surrogate to speed up mobile access. It provides both integrity and confidentiality protection to the data. However, writes still go directly to the trusted home server. Surrogates rely on CODA-style callbacks [50] to guarantee consistency.

SiRiUS [42] builds a secure file system layered upon existing insecure network file systems. SiRiUS shares some similarity with Plutus on achieving confidentiality and integrity. Yet it provides freshness (consistency) guarantees in a unique way. Every user's *root* directory contains a meta-data freshness file (*mdf-file*), which ties a timestamp to a Merkle hash tree of meta-data of all the files in the root directory. A freshness daemon updates *mdf-file* periodically. Readers who verify the timestamp can be assured they would not accept files older than that. However, it is not completely clear how feasible a 24/7 online daemon would be: attackers can subvert the system easily by launching DoS attacks on the daemon. Still, this idea can be adapted in SUNDR to build a trusted time-stamp box to detect forking attacks.

5.2 Byzantine Fault Tolerant systems

PBFT/BFT2F builds on replicated state machines. Replicated state machines generally deal with concurrency more efficiently than quorum systems, but scale poorly [14]. Many other BFT systems [69, 23] take this approach. Some wide area file systems [71, 15] run PBFT on their core servers. Instead of replicated state machines, another popular way to tolerate Byzantine faults is to use Byzantine Quorum Systems [55, 56, 57, 87]. Quorums have simpler construction and are generally more scalable [14]. However, quorums usually only provide low-level semantics, like read and write, which makes building arbitrary applications more challenging.

Some work has been done to reduce BFT's assumption barrier and to minimize the chance of potential violations of the assumption that no more than f replicas might fail. Proactive recovery [26] reduces the assumption of no more than f faulty replicas during the life time of the service to a window of vulnerability. It achieves this by periodically rebooting replicas to an uncompromised state. It shares the same problem with BFT when more than f replicas fail during this window of time. Furthermore, some problems such as software bugs persist across reboots. BASE [72] aims to reduce correlated failures. It abstracts well-specified state out of complex systems, and thus increases the chance of using different existing mature implementations. By separating execution replicas from agreement replicas [83], one can tolerate more failures within execution replicas or reduce replication cost. BAR [16] explicitly takes the behavior of selfish nodes into account, thus it is able to tolerate these faults which are not possible in traditional BFT systems. Dynamic Byzantine quorum systems [17] can adjust f on the fly, based on the observation of system behavior.

BFT2F's contribution lies in that it further limits the potential damage of not meeting

BFT's f threshold assumption. It allows the system to monitor any past consistency violations with more than f, but no more than $2f$ faulty replicas, yet still provides 100% compatibility with classical BFT system when no more than f replicas fail.

5.3 Encapsulation of hostile behavior

In this thesis, our approaches consider all servers (as in SUNDR), or a fraction of servers (as in BFT2F) *completely* untrusted, and rely on carefully-designed protocols to reveal malicious behavior to end users. Another direction is to enforce security policies within the server domain itself. Such systems usually need a small trusted computing base (TCB) underneath the untrusted components, either in software, hardware, or both, to host such a service.

By far, the most popular place to accommodate such a TCB is in a virtual machine monitors (VMM) [30]. A VMM is considerably simpler and smaller, because it only provides a narrow hardware-level interface rather than a full set of operating system abstractions. Thus, it is more easily verifiable, and less vulnerable to security holes. Service in a VMM would continue to run correctly even with guest operating systems subverted by attackers. VMMs also improve portability, since hardware interfaces evolve much more slowly than their software counterparts. Furthermore, by emulating hardware in software, VMMs gain additional advantages: First, it is easier to modify virtual machines to add additional functionalities. Second, virtual machine state can be manipulated more flexibly. The difficulty of the VMM approach is bridging the gap of hardware and guest operating system semantics. For example, one needs to traverse an emulated MMU to tie an application variable to a watched memory location. In the past, systems like secure logging/reply [34], intrusion backtracking [49], dynamic taint analysis [31], and predicate checking [46] have been

developed with this approach.

The VMM approach is still limited to the hardware emulated. Most current platforms lack any basic security measures. For instance, even *attesting* to the currently running software is hard now. Thus, trusted computing platforms, such as TCPA [7], have been proposed. With such "trusted" hardware support, essentially no other trust [53], or little trust [40] (in a secure VMM) is needed to let authors control their applications. However, as closed black-boxes that end users have no direct control of, whether such platforms can succeed is still unclear at the moment.

5.4 Anonymity systems

Finally, anonymity systems are of particular interest to us in the sense that some untrusted servers behave maliciously according to the identities of the clients. For example, if a bad server intends to hide some update from some clients, it would be much harder for him to do so if clients anonymized themselves when fetching updates, since the creator of the update could also anonymously fetch her own update to do a sanity check from time to time, and detect the attack with some probability.

In anonymity systems, clients do not want to reveal their identities to servers, but otherwise trust the servers to provide the advertised service honestly. This can be achieved by establishing anonymous channels between clients and the server. Thus, the server cannot trace back which client initiates the connection. The simplest solution is to route traffic through a centralized proxy [2]. However, this approach introduces a single point of failure: The failure of the proxy would discontinue the service entirely. Such systems are also vulnerable to an attacker that controls the proxy, since he can easily reveal the correlation between incoming and outgoing communication. Another classical approach is to use

mix-net, first introduced by Chaum in [29]. By encrypting and randomly shuffling all incoming and outgoing messages through a mix, a local eavesdropper cannot pinpoint the initiator. By chaining multiple mixes in cascade, one can also tolerate some fraction of hostile mixes. Examples of such systems include anonymous routing [76, 37], anonymous publishing [79, 78], and anonymous email [58].

Yet another alternative is based on the idea of *"blending into crowds"*. Crowds [70] aims to make web browsing anonymous. When a request is initiated, it can be submitted directly the server, or forwarded to another peer, depending on a local biased coin flip. Then the request goes on recursively through all peers en route, until it finally reaches the server. This process makes each peer in the crowd equal likely to be the originator, thus anonymizing the true originator.

Chapter 6

Conclusion

6.1 Future work

With the increasing popularity of P2P systems, such as Bittorrent [32], KaZaA [4], eMule [3], and gnutella [5], it is natural that peers trust each other even less than a client would typically trust a server. As a consequence, a system that takes the inherent distrustfulness of participants into consideration in its design would be more likely to survive such a hostile operating environment. Bittorrent assumes nodes are mainly selfish, and establish transient trust relation based on other nodes' current behavior. It explicitly takes the "Tit-for-Tat" strategy as a part of its protocol: nodes upload files in high bit rates only to those who also do the same reciprocally. Largely due to this fact, Bittorrent has overtaken others as the most popular P2P file sharing protocol.

Academic effort have also tried to solve the *whom to trust* problem in many other ways. Reputation systems [48, 80] allocate resources based on nodes' past behavior, to distinguish "bad" nodes from "good" ones. However, such systems have a drawback: first, the reputation system itself could be a target of attack, and therefore needs good a defense

mechanism. Second, malicious nodes can pretend to be good for a period of time, and then launch attacks in a well-coordinated fashion to inflict more damage. Social networks map people's real-world trust relationships into cyberspace. Using your friends' computers as peers in the network, rather than random ones as in the current systems, could result in significantly more reliable storage [52]. Forcing social links on routing structure [84] reduces the danger of Sybil attacks, since malicious users can easily forge many identities in the network, but not friend relationships in the real world. Nevertheless, the success of this approach still largely depends how much people would like to reveal their personal friendship information, and how wise they are in selecting "friends".

6.2 Conclusion

Placing unconditional trust on centralized storage systems, as in traditional approaches, is questionable. Not only is this unnecessary: for example, system administrators should not be able to access or modify your files; but also dangerous: your data might be tampered with by outside attackers without your ability to detect this fact promptly, and your retrieval might return stale data. In this thesis, we advocate building data storage systems on untrusted (or in other words, *possibly* compromised) hosts. Designing a system specifically for running over untrusted servers in the first place makes the consequences of a potential break-in clear, minimizes the resulting damages, and eases detection of any past attacks.

Through our experience with untrusted systems, several points are worth emphasizing:

1. **Graceful degradation**

"Something is better than nothing."

Graceful degradation specifies the behavior of a system when something goes wrong, if the system still supports audit, termination, and recovery. In a word, it should provide at least some protection in the worst-case scenario.

Most BFT systems are well known for "all-or-nothing": they either guarantee ideal consistency when the *f-threshold* assumption holds, or nothing at all. This may be one factor that has slowed adoption of BFT systems. As we show in chapter 4, it is desirable and feasible to provide some form of weak guarantees when more than $1/3$ of the nodes fail.

2. **Constrained behavior**

"Absolute power leads to absolute corruption."

Constrained behavior means that a compromised node, no matter how important it is, cannot change the system behavior in an arbitrary manner. This means responsibility and trust are more evenly distributed in the system. Furthermore, constrained behavior effectively limits the scope of damage that does occur.

For example, in SUNDR, a compromised server cannot inject fake updates, or tamper with users' data. In some DHTs [24], a malicious node is not allowed to arbitrarily pollute others' routing tables.

3. **Accountability**

"Freedom and responsibility without control?"

Accountability makes it possible to catch *any* past malicious behavior *eventually*, and provide a non-repudiable, tamper-resistant, and self-verifiable proof of the entity

that is responsible for the violation [67, 86]. Additionally, accountability could be a powerful deterrent to attacks.

Often, accountability is at odds with anonymity. [58] describes a case where anonymous email was abused. The failure of the anonymous email system in this case demonstrates the importance of accountability.

We propose two different solutions in this thesis. In the first approach, we design and implement SUNDR, which runs on *one* untrusted server. Even though there is no trusted party online, SUNDR still effectively minimizes the damage a malicious server can cause. A compromised server cannot arbitrarily tamper with users' data undetectably. The worst thing it can do is to conceal users' updates, and serve *stale*, but valid data. When the server does this, the SUNDR protocol forks mistreated users into different fork groups, where they can never see updates from users in other groups again. We call this fork consistency. Fork consistency significantly constrains the server's ability to prolong attacks, since any sort of out-of-band communication between users in different fork groups can reveal the attack.

Measurements of our implementation show performance that is usually close to and sometimes better than the popular NFS file system. Yet by reducing the amount of trust required in the server, SUNDR both increases people's options for managing data and significantly improves the security of their files.

The alternative solution is to use replicated state machines. Byzantine Fault Tolerant Systems (BFT) have been well known in the community to tolerate some fraction of malicious servers. However, to make BFT deliver ideal consistency, faulty replicas cannot exceed some threshold. (For example, in an asynchronous BFT system, no more than one third of the replicas can be faulty at the same time.) Little has been done to investigate the realm beyond this threshold assumption. We propose BFT2F, an extension to Castro-

Liskov's PBFT algorithm, to further bound malicious behavior when more than one third but no more than two thirds of the replicas fail, within which range BFT2F provides a weaker consistency guarantee – fork* consistency. Though weaker than fork consistency or ideal consistency, fork* consistency allows simpler and more efficient protocols. BFT2F does not guarantee liveness in the situation, but we believe the resulting system is still preferable to a system that exhibits arbitrary behavior when there are more than f failures. Evaluations of our prototype implementation show that BFT2F's additional guarantee comes with only a modest performance penalty.

Bibliography

[1] Amazon web services store: Amazon S3. http://aws.amazon.com/s3/.

[2] Anonymizer. http://www.anonymizer.com/.

[3] eMule. http://www.emule-project.net/.

[4] Gnutella. http://www.gnutella.com/.

[5] kaZaA. http://www.kazza.com/.

[6] Yahoo! mail. http://mail.yahoo.com/.

[7] The Trusted Computing Alliance. http://www.trustedpc.com/, 2000.

[8] Apache.org compromise report. http://www.apache.org/info/20010519-hack.html, May 2001.

[9] Business beware - the enemy lies within. http://www.suntimes.co.za/2002/08/04/business/surveys/survey22.asp, August 2002.

[10] Disgruntled UBS Painewebber employee charged with allegedly unleashing ”logic bomb” on company computers. http://www.cybercrime.gov/duronioIndict.htm, December 2002.

[11] Debian investigation report after server compromises. http://www.debian.org/News/2003/20031202, December 2003.

[12] Intrusion on www.gnome.org. http://mail.gnome.org/archives/gnome-announce-list/2004-March/msg00114.html, March 2004.

[13] Compromise of gluck.debian.org, lock down of other debian.org machines. http://lists.debian.org/debian-devel-announce/2006/07/msg00003.html, July 2006.

[14] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74, Brighton, United Kingdom, October 2005. ACM.

[15] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, December 2002.

[16] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 45–58, Brighton, United Kingdom, October 2005. ACM.

[17] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K. Reiter, and Rebecca N. Wright. Dynamic Byzantine quorum systems. In *Proceedings of the the International*

Conference on Dependable Systems and Networks (FTCS 30 and DCCA 8), pages 283–292, June 2000.

[18] Philip Bernstein, Vassco Hadzilacos, and Nathan Goodman. In *Concurrency control and recovery in database systems*, Boston, MA, 1987. Addison-Wesley.

[19] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. Oceanstore: An exteremely wide-area storage system. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, 2000.

[20] Matt Blaze. A cryptographic file system for Unix. In *1st ACM Conference on Communications and Computing Security*, pages 9–16, November 1993.

[21] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[22] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO'01*, pages 524–541, 2001.

[23] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 167–176, 2002.

[24] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[25] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.

[26] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 273–288, San Diego, CA, October 2000.

[27] Miguel Castro and Barbara Liskov. Byzantine fault tolerance can be fast. In *International Conference on Dependable Systems and Networks*, pages 513–518, Goteborg, Sweden, July 2001.

[28] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *USENIX Annual Technical Conference 2001, Freenix Track*, June 2001.

[29] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb 1981.

[30] Peter Chen and Brian Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, May 2001.

[31] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, pages 321–336, San Diego, CA, August 2004.

[32] Bram Cohen. Incentives build robustness in bittorrent. In *First Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[33] Dan Duchamp. A toolkit approach to partially disconnected operation. In *Proceedings of the 1997 USENIX*, pages 305–318. USENIX, January 1997.

[34] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay.

[35] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[36] Jason Flinn, Shafeeq Sinnamohideen, Niraj Tolia, and M. Satyanaryanan. Data staging on untrusted surrogates. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.

[37] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, D.C., 2002.

[38] Kevin Fu. *Integrity and Access Control in Untrusted Content Distribution Networks*. PhD thesis, Massachusetts Institute of Technology, September 2005.

[39] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[40] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206, Bolton Landing, NY, October 2003. ACM.

[41] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, Bolton Landing, NY, October 2003. ACM.

[42] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, pages 131–145. Internet Society (ISOC), February 2003.

[43] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the 3rd Symposium on Principle of Distributed Computing*, pages 50–61, Vancouver, BC, Canada, August 1984.

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

[45] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[46] Ashlesha Joshi, Samuel King, George Dunlap, and Peter Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 91–104, Brighton, United Kingdom, October 2005. ACM.

[47] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.

[48] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651, May 2003.

[49] Samuel King and Peter Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003. ACM.

[50] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[51] Leslie Lamport. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[52] Jinyang Li and Frank Dabek. F2F: Reliable storage in open networks. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS '06)*, Santa Barbara, CA, February 2006.

[53] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 178–192, Bolton Landing, NY, October 2003. ACM.

[54] Umesh Maheshwari and Radek Vingralek. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[55] Dahlia Malkhi and Michael Reiter. Byzantine quorum system. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 569–578, El Paso, TX, May 1997.

[56] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 7th IEEE Symposium on Reliable Distributed Systems*, pages 51–58, October 1998.

[57] Dahlia Malkhi, Michael K. Reiter, Daniela Tulone, and Elisha Ziskind. Persistent objects in the Fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, 2001.

[58] David Mazières and M. Frans Kaashoek. The design, implementation and operation of an email pseudonym server. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 27–36, 1998.

[59] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, December 1999. ACM.

[60] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the 21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[61] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. Technical Report TR2002–826, NYU Department of Computer Science, May 2002.

[62] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Berlin, 1987. Springer-Verlag.

[63] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 31–44, December 2002.

[64] Tatsuaki Okamoto and Jacques Stern. Almost uniform density of power residues and the provable security of ESIGN. In *Advances in Cryptology – ASIACRYPT*, pages 287–301, 2003.

[65] Brian Oki and Barbara Liskov. Viewstamped replication: a general primary copy. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, Ontario, Canada, August 1988.

[66] D. Stott Parker, Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[67] Bogdan C. Popescu, Bruno Crispo, and Andrew S. Tanenbaum. Secure data replication over untrusted hosts. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[68] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

[69] Michael K. Reiter. The Rampart toolkit for building highintegrity services. *Lecture Notes in Computer Science 938*, pages 99–110, 1994.

[70] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. Technical Report 97-15, DIMACS, April 1997.

[71] Sean Rhea, Patrick Eaton, and Dennis Geels. Pond: The OceanStore prototype. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.

[72] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 15–28, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[73] Mendel Rosenblum and John Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[74] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.

[75] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, Ann Arbor, MI, April 1981.

[76] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous connections and onion routing. In *Proceedings of the 18th annual Symposium on Security and Privacy*, pages 44–54, Oakland, CA, May 1997. IEEE.

[77] Jr. T. W. Page, R. G. Guy, J. S. Heidemann, D. H. Rather, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically repllicated, peer-to-peer filinga. *Software - Practice and Experience*, 28(2):155–180, February 1998.

[78] Marc Waldman and David Mazières. Tangler – a censorship-resistant publishing system based on document entanglements. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 126–135, November 2001.

[79] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, pages 59–72, Denver, CO, August 2000.

[80] Kevin Walsh and Emin Gun Sirer. Experience with an object reputation system for peer-to-peer filesharing. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, pages 1–14, San Jose, CA, May 2006. USENIX.

[81] Assar Westerlund and Johan Danielsson. Arla—a free AFS client. In *Proceedings of the 1998 USENIX, Freenix track*, New Orleans, LA, June 1998. USENIX.

[82] Charles P. Wright, Michael Martino, and Erez Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, June 2003.

[83] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services.

In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, Bolton Landing, NY, October 2003. ACM.

[84] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: Defending against sybil attacks via social networks. In *ACM SIGCOMM*, Pisa, Italy, September 2006.

[85] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4rd Symposium on Operating Systems Design and Implementation*, pages 305–318, October 2000.

[86] Aydan R. Yumerefendi and Jeffrey S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First Workshop on Hot Topics in System Dependability*, Yokohama, Japan, June 2005.

[87] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.