Building secure file systems out of Byzantine storage[*]

David Mazières and Dennis Shasha
NYU Department of Computer Science
{dm,shasha}@cs.nyu.edu

ABSTRACT

This paper shows how to implement a trusted network file system on an untrusted server. While cryptographic storage techniques exist that allow users to keep data secret from untrusted servers, this work concentrates on the detection of tampering attacks and stale data. Ideally, users of an untrusted storage server would immediately and unconditionally notice any misbehavior on the part of the server. This ideal is unfortunately not achievable. However, we define a notion of data integrity called *fork consistency* in which, if the server delays just one user from seeing even a single change by another, the two users will never again see one another's changes—a failure easily detectable with on-line communication. We give a practical protocol for a multi-user network file system called SUNDR, and prove that SUNDR offers fork consistency whether or not the server obeys the protocol.

1. INTRODUCTION

There are many reasons not to trust one's file server. An unscrupulous person with administrative access can easily tamper with files, making modifications that go undetected. Yet people often entrust servers to people who have no role in creating or using the data stored on them—for instance system administrative consultants or data warehouse employees. Furthermore, network intruders often penetrate servers to gain privileged access.

Incorrect server behavior can be categorized as either fail-stop or Byzantine. Fail-stop behavior encompasses failures in which the server is unable or unwilling to perform an expected task, such as returning a file it should be storing. People deal with such problems through disk and server redundancy and backups. Byzantine behavior, by contrast, includes faulty actions that may go undetected, such as subtle, malicious tampering by hackers or unscrupulous employees. This work is the first to show how to render Byzantine file server failures readily detectable.

[*]A full version of this paper is available as NYU computer science department technical report TR2002–826 [12].

This research was supported in part by National Science Foundation awards CCR–0093361 and IIS–9988345.

In an ideal network file system, if one user writes and closes a file, the next user to open the file should see the exact contents just written. In an untrusted setting, one can gain some assurance of data integrity with digital signatures, but a signature cannot guarantee that a user is reading the most recent version of a file. If the last signed file update user u_1 sees from user u_2 is 24 hours old, does this mean u_2 hasn't logged in for a day, or is the file system concealing u_2's updates from u_1?

We propose a slight weakening of traditional file system consistency semantics called *fork consistency*. A file system with fork consistency might conceal users' actions from each other, but if it does, users get divided into groups and the members of one group can no longer see any of another group's file system operations. Users can detect such partitioning through any available out-of-band communication—from conversation ("Did you see the new file?"), to client-to-client gossip protocols, to trusted on-line version-verification devices.

The remainder of this paper describes the data structures and protocol of a multi-user network file system called SUNDR (Secure Untrusted Data Repository). We prove that SUNDR guarantees fork consistency, whether or not the server obeys the protocol. As described here, SUNDR does not provide data secrecy. We intend to achieve secrecy by layering existing cryptographic storage techniques on top of SUNDR. However, since the consistency protocol is the more novel contribution, we will not address secrecy in this paper.

Each SUNDR user has a public key for digital signatures. A user's private key must remain secret from all other system elements, including the server and other users. Because the user signs its updates, neither the server nor any other user can forge updates from that user. SUNDR also requires every client to have a small amount of persistent storage so as to remember some version information about the last message it has signed.

SUNDR is based on the premise that digital signatures can be computed faster than network round trip times and verified highly efficiently. This assumption is increasingly valid as hardware speeds increase (and the speed of light doesn't). Today, an 800 MHz Pentium III can compute 1,024-bit Rabin signatures in 7 msec and verify them in 50 μsec. Some patented algorithms are even faster. Esign, for instance, achieves 350 μsec signatures and 200 μsec verifications with 2,048-bit keys. In SUNDR, opening a file or closing a modified file requires the client to compute two digital signatures and wait for one network round trip. A third, asynchronous message is required before another user can access freshly written contents. Concurrent operations can be piggybacked into a single pair of digitally signed messages, allowing heavily-loaded clients to reduce the total number of signatures they compute.

2. RELATED WORK

Recently, there has been growing interest in peer-to-peer storage systems comprised of potentially untrusted nodes. OceanStore [2] has envisaged data migrating all over the world to follow users, but has weak consistency and security guarantees. xFS [1] introduced the idea of serverless network file systems, while Farsite [4] investigated the possibility of spreading such a file system across people's unreliable desktop machines. Several new distributed hash tables such as Chord [18] and Pastry [16] show the potential to scale to millions of separately administered volunteer nodes, with CFS [6] layering a read-only file system on top of such a highly distributed architecture. These systems can all benefit from SUNDR's protocol, which is the first to implement anything resembling traditional file system semantics without trusting the storage infrastructure.

A number of file systems in the past have used cryptographic storage to keep data secret in the event of a server compromise. The swallow [15] distributed file system used client-side cryptography to enforce access control. Clients encrypted files before writing them to the server. Any client could read any file, but could only decrypt the file given the appropriate key. Unfortunately, one could not grant read-only access to a file. An attacker with read access could, by controlling the network or file server, substitute arbitrary data for any version of a file.

CFS [3] allows users to keep directories of files that get transparently encrypted before being written to disk. CFS does not allow sharing of files between users, nor does it guarantee freshness or integrity of data. It is intended for users to protect their most sensitive files from prying eyes, not as a general-purpose file system. Cepheus [8] adds integrity and file sharing to a CFS-like file system, but trusts the server for the integrity of read-shared data. SNAD [14] can use digital signatures for integrity, but does not guarantee freshness. PFS [17] is an elegant scheme for checking the integrity of a file system stored on an untrusted disk. With minor modifications, PFS could make strong freshness guarantees. However, PFS is really a local file system designed to reside on untrusted, potentially remote disks. Users on multiple clients cannot simultaneously access the same file system.

The Byzantine fault-tolerant file system, BFS [5], uses replication to ensure the integrity of a network file system. As long as more than $2/3$ of a server's replicas are uncompromised, any data read from the file system will have been written by a legitimate user. SUNDR, in contrast, does not require any replication or place any trust in machines other than a user's client. If data is replicated in SUNDR, only one replica need be honest for the file system to function properly. However, SUNDR and BFS provide different freshness guarantees.

SUNDR uses hash trees, introduced in [13], to verify a file block's integrity without touching the entire file system. Duchamp [7], BFS[5], SFSRO [9] and TDB [10] have all made use of hash trees for comparing data or checking the integrity of part of a larger collection of data.

3. ARCHITECTURE

SUNDR decomposes the problem of reading proper file data into two parts. First, starting with a bit string known as a user's i-handle and a file identifier called the i-number, a SUNDR client can retrive and verify the integrity of any block in the file. Second, there is a protocol for users to update their i-handles and for each user to ensure she has the latest i-handle of every other user. We describe the first procedure in this section, deferring the tougher problem problem of i-handle consistency to the next section.

We begin with a brief overview of SUNDR's client-server architecture and file system metadata structures. We simplify the description somewhat so as to leave room for the a detailed description and proof of the consistency protocol in the next section. The particular file system we describe provides a practical and concrete use of the consistency protocol, though our technique is potentially of more general applicability.

SUNDR is organized as a server and a set of clients. The server operates at a lower level than conventional file servers. It stores blocks of data for users, but neither understands nor interprets the blocks that it stores. The basic interface to the server consists of three RPCs, STORE, RETRIEVE, and UNREFERENCE. STORE stores a chunk of data on the server. RETRIEVE retrieves a block by its collision-resistant cryptographic hash. In case a block is stored multiple times, the server keeps a count for each user of how many times the user has stored the block. The UNREFERENCE RPC decrements a user's reference count (if not already zero), and reclaims the storage when a block's count goes to zero for all users.

SUNDR clients are responsible for interpreting the data chunks stored at servers as inodes, directory blocks, and file data. To ensure the integrity of the file system, SUNDR relies on a collision-resistant cryptographic hash function and a digital signature scheme. We assume the signatures are existentially unforgeable against a chosen message attack. Though of course there exists a finite but negligible chance of the cryptography failing, for the rest of this paper we simply assume that no collisions occur and no signatures are forged.

Every user of a SUNDR file system has a public key. For the purposes of this paper, we do not care how these keys are managed, so long as all parties agree on every user's public key. One possibility is to embed the superuser's public key in the file system's pathname, as in SFS [11], and for the keys of other users to reside in some file `/etc/sundr_users` owned by the superuser and which the server also knows how to retrieve. There is also a file `/etc/group` indicating which users are in which groups. The SUNDR server authenticates the users it communicates with, so as to prevent one user from unreferencing another's blocks. The server might also provide some mechanism to enforce disk quotas on users. Finally, the server itself has a public key known to users (for instance certified by the superuser's public key), so that clients can authenticate the server. We assume that all RPCs are authenticated, so that users can hold the server responsible for any incorrect replies.

In addition to storing blocks, the server stores a signed *version structure* for each user and group, and also some information about operations in progress. A version structure consists of version data (which we will describe in the next section) and an *i-handle*. The i-handle is a single cryptographic hash with which one can verify the integrity of any block of any file owned by a particular user. i-

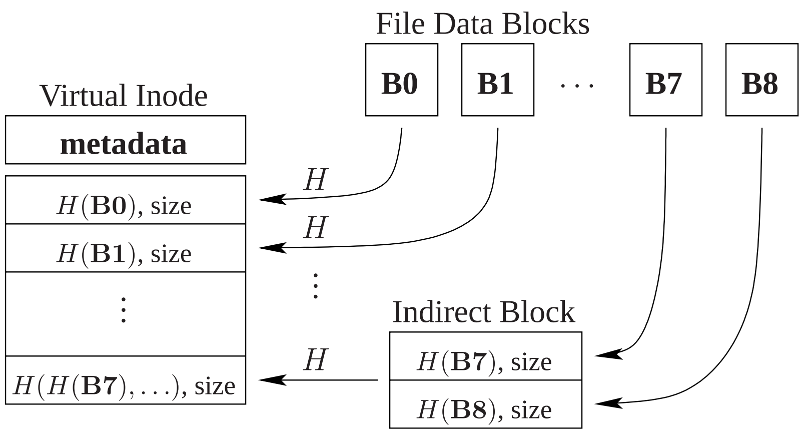

Figure 1: The SUNDR virtual inode structure

handles are the result of recursively hashing file system data structures in a manner similar to the SFSRO file system [9]. We describe these data structures from the ground up.

The lowest-level data structure in SUNDR is the *virtual inode*, shown in Figure 1, with which one can efficiently retrieve and verify any portion of a file or directory. The virtual inode contains a file's metadata and the size and cryptographic hashes of its blocks. For large files, the inode also contains the hash of an *indirect block*, which in turn contains hashes and sizes of file blocks. For larger files, an inode can point to double-, triple-, or even quadruple-indirect blocks. The hash of a virtual inode is known as a *file handle*. Given a file handle, one can retrieve any block of the file with RETRIEVE RPCs, first retrieving the inode by the file handle, then retrieving data and/or indirect blocks by the hashes in the inode.

Each user and group also has an ordered list, known as the *i-table*, mapping 64-bit per-user virtual inode numbers, or *i-numbers*, to file handles. An i-table lists every file belonging to a particular user or group. The i-table is broken into blocks, converted to a hash tree, and each node of the tree stored at the SUNDR server. The hash of the tree's root is the i-handle of the user or group. Given an i-handle, one can retrieve and verify the file handle of any i-number by retrieving the appropriate intermediary blocks from the server. Directories also use virtual inodes. The data blocks of a directory contain a list of $\langle \text{file name}, \text{user}, \text{i-number}\rangle$ triples, sorted by file name. By convention, inode number 2 in the superuser's i-table is the root directory of the file system.

To modify the file system, a client computes a new i-handle, stores new blocks at the SUNDR server, and computes and signs a new version structure. The client then updates its version structure on the server using two more RPCs, UPDATE and COMMIT, described in the next section. Finally, the client unreferences any blocks it no longer needs.

4. CONSISTENCY PROTOCOL

The goal of SUNDR's consistency protocol is to make it as easy as possible to detect whether the server has faithfully provided a consistent view of the file system to all clients. As we will show, if a SUNDR server fails to show one user another's updates, either user will detect the attack upon seeing any subsequent file system operation by the other, even if through a third user. We call this property *fork consistency*. This section begins with a formal definition of fork consistency. We then list and prove a set of criteria sufficient for any protocol to achieve fork consistency. Finally, we develop three successively more efficient and general protocols and show that they satisfy the criteria for fork consistency.

The file system literature uses the term *close-to-open consistency* to speak of the consistency of a wide variety of operations. In practice, some file system calls (such as truncate) synchronously modify a file without opening or closing it, and must also be immediately visible to other users. Thus, for the purposes of this paper, we will speak of *fetches* and *modifications* rather than opens and closes, and we will concern ourselves with fetch-modify consistency. A fetch is the process by which a client either validates its cached copy of a file or downloads new contents from the server. Modification is the process through which a client makes new file system content visible to other clients. (Modification can occur on file closes, but also on *fsync* calls and certain metadata operations).

Definition 1 *A* ***client*** *is an entity that produces a set of fetch and modify requests and sends them to the server. Each request has a wall-clock time associated with it called the* issue time.

Conceptually, the issue time corresponds to the time at which software invoked the file system's fetch or modify routine. If all goes well, the routine will return at some later time we call the completion time. Note we do not assume that clients have synchronized clocks. Thus, a client will not know the issue time of its own operations.

Definition 2 *A* ***principal*** *is an entity authorized to access the file system. Each principal has a public signature key, the private half of which we assume is unknown to the server.*

Examples of principals include a user, a user acting as a member of a group, and a client acting on behalf of a user. Note one private key may speak for multiple principals, as when a user is a member of several groups.

Definition 3 *A set of fetch and modify operations on a file system is* ***orderable*** *if each operation has a* completion time *later than its* *issue time and there exists a partial order,* happens before*, on the operations such that:*

1. *If the completion time of operation O_1 is earlier than the issue time of operation O_2, then O_1 happens before O_2.*
2. *Happens before orders any two operations by the same client (even if both operations are issued before either completes).*
3. *Happens before totally orders all modifications to any given file.*
4. *Happens before orders any fetch of a file with respect to all modifications to the same file.*

Orderability restricts but does not completely specify the order of operations. Once an operation completes, it happens before any subsequently issued operation. For concurrent operations, however, the file system is free to choose any order, so long as dependent operations have a definite order.

Definition 4 *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.*

Fetch-modify consistency relates the order of operations to the results of fetch operations, but leaves the precise semantics of a modification open. If modifications always overwrite the entire contents of a file (as in SUNDR), a fetch-modify consistent file system need only return the result of the last modification to a fetched file. File systems in which modifications change only part of a file must return the effect of composing all previous modifications in happens before order.

Definition 5 *Given a set of operations that have completed on a file system, a* ***forking tree*** *is a tree in which each node has an associated set of operations, called a* ***forking group****, and the forking groups have the following properties:*

1. *Every forking group is fetch-modify consistent.*
2. *For every client c, there is at least one forking group that contains every operation by c.*
3. *For any operation O, the set of nodes whose forking groups contain O includes a common ancestor, n, and all descendents of n.*
4. *If two nodes' forking groups both contain operations O_1 and O_2, and O_1 happens before O_2 in the first forking group, then O_1 also happens before O_2 in the second forking group.*
5. *Every operation in a node's forking group either occurred in the parent node's forking group or else happened after every operation in the parent node's forking group.*

Definition 6 *A file system is* ***fork consistent*** *iff it always guarantees the existence of a forking tree on the set of completed operations.*

Informally, each branch of a forking tree represents a failure of the server to deliver fetch-modify consistency. Initially, the operations of all clients are in the same (root) forking group, G_0. Then, if, say, c_1 makes a modification M to file f, M completes, and c_2 subsequently issues a fetch F that returns an older version of f, M and F must occur in two new forking groups—G_1 for M and subsequent operations by c_1, and G_2 for F and subsequent operations by c_2. At this point, c_1 can never see another operation by c_2 and vice versa (that's why we want a tree topology). Moreover, no later operations by other clients can go in G_0. Thus, the set of clients is partitioned into two groups that will never again see each others' operations. This situation is very likely to be noticed soon through out-of-band communication between clients or users.

4.1 Protocol correctness theorem

Protocol correctness theorem: A set of (completed) operations on a file system is fork consistent if there exists a partial order $<$ on operations with the following two properties:

1. Every two distinct operations created by a single client are ordered by $<$.
2. For any operation q, the set $\{o \mid o \leq q\}$ of all operations (by any client) less than or equal to q is totally ordered and fetch-modify consistent with $<$ as the happens-before relation.

Proof: Recall that a file system is **fork consistent** iff it always guarantees the existence of a forking tree on all completed operations of all clients.

Consider the set $\mathbf{X} = \{x_1, \ldots, x_k\}$ of maximal operations by $<$. (If the file system has been fetch-modify consistent and hasn't stopped completing operations, the set will have only one element.) Consider the collection $\mathbf{C} = \{C_1, \ldots, C_k\}$ of sets of operations such that $C_i = \{y \mid y \leq x_i\}$. By condition 2, the operations in each C_i are totally ordered. For that reason, we will reinterpret the C_is to be sequences of operations ordered by $<$. We call these sequences chains.

We now construct a forking tree. The nodes of the tree, the forking groups, consist of the chains and the greatest common prefix of every pair of chains. A node n's parent is simply the the longest strict prefix of n that is also a node. Thus, the chains constitute the tree's leaves, and the least common ancestor of any two nodes is their greatest common prefix.

For example, consider three maximal chains:

$$\begin{aligned} s_1 &= A, B, C, D, E, F \\ s_2 &= A, B, C, D'', E'' \\ s_3 &= A, B, C', D', E' \end{aligned}$$

Figure 2 shows the resulting tree.

Now let us verify the five required properties of the forking tree:

1. *Every forking group is fetch-modify consistent.*

   Since chains are totally ordered, and each forking group is a prefix of a chain, every forking group consists of a maximum element, q, and every operation $o \leq q$. Thus, it is fetch-modify consistent by condition 2.

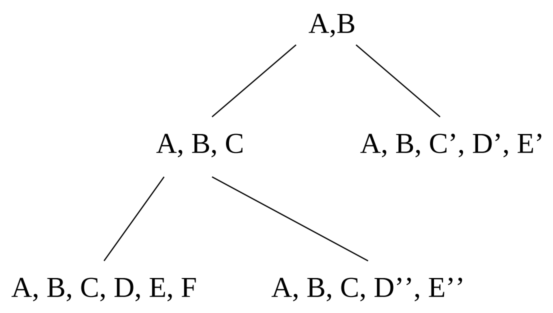

Figure 2: Tree resulting from the three maximal chains.

2. *For every client c, there is at least one forking group that contains every operation by c.*

   Since all operations formed by c are totally ordered by condition 1, they form part of a leaf forking group by construction.

3. *For any operation O, the set of nodes whose forking groups contain O includes a common ancestor, n, and all descendents of n.*

   By construction, the intersection of any set of forking groups is also a forking group. Let n be the intersection of all the forking groups containing O. n obviously contains O. n is also a prefix of any other node containing O, and thus must be a common ancestor. It also follows from our construction that any descendent of n is a superset of n, and hence contains O.

4. *If two nodes' forking groups both contain operations O_1 and O_2, and O_1 happens before O_2 in the first forking group, then O_1 also happens before O_2 in the second forking group.*

   Since all forking groups use the same ordering relation, they will order any common operations in the same way.

5. *Every operation in a node's forking group either occurred in the parent node's forking group or else happened after every operation in the parent node's forking group.*

   By construction, a child of node n contains an extension of the operations in n. Any member of the proper extension follows every operation in n by the ordering relation that corresponds to happens before.

■

4.2 Bare-bones protocol

We describe the SUNDR protocol in stages. We begin with a bare-bones protocol that provides fork consistency for an unrealistically simple usage scenario. We then extend this bare-bones protocol, using the same intuition, to achieve a practical, fork-consistent file system protocol.

In the bare-bones scenario, there is a single client per user producing all of that user's requests. Thus, we can employ the terms user and client somewhat interchangeably. We also assume that each file can be written by only one user; there are no group-writable files. Finally, we assume a low degree of concurrent file system access. Subsequent sections show how to handle concurrency efficiently and and how to deal with group-writable files. Finally, Section 4.5 describes how the mechanism for group-writable files can also allow one user to employ several clients. We also discuss several further optimizations of the protocol.

Recall that each user of the file system has a version structure, consisting of an i-handle and version data. The version data contains the name of the user whose version structure it is (the structure's *owner*, by whose private key the structure should be signed), and a list of user-version pairs.

In all the SUNDR protocols, every signed i-handle has a monotonically increasing version number. Moreover, users sign not just their own version numbers, but also their view of other users' version numbers. We will define an ordering on version structures such that with an honest server, all operations are totally ordered. However, any fetch-modify consistency failure results in two unordered version structures. Since any client would detect the attack if it saw the unordered structures, clients are split into forking groups that can no longer see each other's updates. We formalize the idea as follows.

Definition 7 *We use the following notation for version structures:*

$$\{\textsc{vrs}, h, u, u_1\text{-}n_1\ u_2\text{-}n_2 \ldots\}_{K_u^{-1}}$$

VRS is just a constant identifying the type of the signed data. h is the i-handle. u is the owner of the version structure. We use a hyphen to denote user-version pairs, so that u_i-n_i means that user u_i is at version number n_i. We use the subscript K_u^{-1} to denote that the structure has been signed by user u's private key.

Definition 8 *For any version structure x and user u, we let $x[u]$ designate either u's version number in x, or else 0 if u does not appear in x.*

Definition 9 *If x and y are two version structures, we say that $x \leq y$ iff for all users u, $x[u] \leq y[u]$. $x < y$ iff $x \leq y$ and there exists a user v such that $x[v] < y[v]$.*

The server maintains the latest version structure signed by each user. We call this collection of signed structures the *version structure list*, or VSL. The server is responsible for sending the latest VSL to anyone performing a fetch or modify operation. Each time a user fetches or modifies a file, it must update its entry in the VSL on the server with RPCs. User u updates its VSL entry as follows:

1. u obtains an exclusive lock on and downloads the VSL. (The lock is coordinated by the server, and thus is not trusted.) For each user v with a signed entry in the VSL, let y_v be that user's version structure.

2. u verifies that y_u is its current version structure, and verifies the signatures on other entries of the VSL.

3. u creates a new version structure, x, initializing it with y_u. u updates the i-handle in x if necessary.

4. For each user v in the VSL, u sets $x[v]$ to that user's signed version number, $x[v] \leftarrow y_v[v]$.

5. u increments its own version number $x[u] \leftarrow x[u] + 1$.

6. u verifies that all entries in the previous VSL (including its old entry y_u) are totally ordered, and that x is greater than all VSL entries. (Note that this verification will fail if for some some users v and w, $y_v[w] > y_w[w]$, because then x will not exceed y_v.)
7. u signs x and sends it to the server, releasing the lock.
8. The server checks that x is totally ordered with respect to the other version structures in the VSL. (This is protection against malicious clients.)

The issue time of an operation is the moment the client begins step 1 of the protocol. The completion time is when the last step finishes.

We illustrate the protocol with an example. Consider two file system users, u and v, both initially at version number 1. The VSL will contain the following entries:

$$\begin{aligned} y_u &= \{\text{VRS}, h_u, u, u\text{-}1 \ \ldots\}_{K_u^{-1}} \\ y_v &= \{\text{VRS}, h_v, v, u\text{-}1\ v\text{-}1 \ \ldots\}_{K_v^{-1}} \end{aligned}$$

If u follows the protocol to update its i-handle to h'_u, its version number will also increase in the VSL:

$$\begin{aligned} y_u &= \left\{\text{VRS}, h'_u, u, u\text{-}2\ v\text{-}1 \ \ldots\right\}_{K_u^{-1}} \\ y_v &= \left\{\text{VRS}, h_v, v, u\text{-}1\ v\text{-}1 \ \ldots\right\}_{K_v^{-1}} \end{aligned}$$

At this point $y_u \geq y_v$. If v now updates its version structure (for instance by fetching a file), v's new version structure will reflect u's new version number:

$$\begin{aligned} y_u &= \left\{\text{VRS}, h'_u, u, u\text{-}2\ v\text{-}1 \ \ldots\right\}_{K_u^{-1}} \\ y_v &= \left\{\text{VRS}, h_v, v, u\text{-}2\ v\text{-}2 \ \ldots\right\}_{K_v^{-1}} \end{aligned}$$

At this point, $y_v \geq y_u$. If, however, the server failed to provide v with u's latest version structure, v would not reflect u's new version, and the VSL would contain:

$$\begin{aligned} y_u &= \left\{\text{VRS}, h'_u, u, u\text{-}2\ v\text{-}1 \ \ldots\right\}_{K_u^{-1}} \\ y_v &= \left\{\text{VRS}, h_v, v, u\text{-}1\ v\text{-}2 \ \ldots\right\}_{K_v^{-1}} \end{aligned}$$

Now the VSL is unordered ($y_u \not\leq y_v$ and $y_v \not\leq y_u$).

Proposition one client: All version structures created by a single client that obeys the bare-bones protocol are totally ordered.

Proof: Immediate by steps 2-6 of the protocol. ∎

Fetch-modify lemma: Suppose all clients follow the bare-bones protocol. Let q be a version structure. Let $\mathbf{O}$ be the set of all completed operations by all clients satisfying $\mathbf{O} = \{o \mid \text{vs}(o) \leq q\}$, where $\text{vs}(o)$ designates the version structure of o. Whether or not the server obeys the protocol, if $\mathbf{O}$ is totally ordered by $<$, then $\mathbf{O}$ is fetch-modify consistent with $<$ as the happens-before relation.

Proof: We first must show that $\mathbf{O}$ is orderable using $<$ as the happens before relation. The execution of the protocol gives each operation a completion time after its issue time. Moreover, since $<$ totally orders $\mathbf{O}$, it satisfies requirements 2–4 of orderability. For the first requirement of orderability, suppose two version structures x and y are ordered and x's operation completed before y's was issued. Let u be the user that signed x, and v be the user that signed y. If $u = v$, then the protocol ensures $x < y$. Otherwise, let y' be v's version structure in the VSL u received while creating x. y' must have completed before x (it had already been signed when u signed x). By assumption, x completed before y issued. Thus, y' completed before y issued. Since y and y' both come from the same client, it follows that $y' < y$ and $y'[v] < y[v]$. By step 4 of the protocol, $x[v] = y'[v]$, implying $x[v] < y[v]$. Thus, since x and y are ordered, it must be that $x < y$.

Now, we show that the file semantics are correct with respect to $<$. Let y, signed by v, be the version structure corresponding to a fetch in $\mathbf{O}$ of file f. By assumption, $\mathbf{O}$ must contain all operations with version structures less than y. Thus, any VSL entries that could have passed step 6 of the protocol when v signed y must be in $\mathbf{O}$. Let x, signed by u, be the greatest version structure less than y (and therefore in $\mathbf{O}$) associated with a modification of f. It follows that for any x' signed by u, if $x \leq x' < y$, then x' designates the same contents for f as x. In particular, let x' be u's entry in the VSL upon which y is based. Since $x'[u] = y[u]$ and users sign at most one version structure for each of their own version numbers, x' must be the greatest version structure signed by u less than y. Thus, $x \leq x' < y$ and v's fetch must have returned the same contents for f as designated by x's i-handle. ∎

No join lemma: Suppose there are two version structures x and y such that $x \not\leq y$ and $y \not\leq x$. If clients follow the protocol, no client will sign any version structure greater than both x and y.

Proof: First note that for any client c and number n, c will sign at most one version structure t with $t[c] = n$. Moreover, any two version structures t and t' signed by c are ordered, and $t < t'$ iff $t[c] < t'[c]$.

Assume that there exists a version structure w such that $x < w$ and $y < w$. There must be at least one minimal version structure $z \leq w$ such that $x \leq z$ and $y \leq z$. In fact, since x and y are unordered, z cannot be either of them, and we must have $x < z$ and $y < z$. Let L be the VSL that was sent to the client that signed z and from which this client calculated z.

Let u be the user that signed x, and let x' be u's entry in L. It must be the case that $x \leq x'$. We show this by contradiction. Assume $x' < x < z$.

- It cannot be the case that z was signed by u, because then it would follow that $x'[u] < x[u] < z[u]$ and hence $z[u] \geq x'[u] + 2$, which is impossible since the protocol sets $z[u] = x'[u] + 1$.

- On the other hand, if z were signed by a different user from u, then the protocol would set $z[u] = x'[u]$. Since $x < z$, $x[u] \leq z[u] = x'[u]$, implying $x \leq x'$.

By a similar argument, the structure y' in L signed by the same user as y must satisfy $y \leq y'$. Since all version structures in L must be ordered, we also have that x' and y' are ordered. Assume without loss of generality that $x' < y'$. We then have $x \leq x' < y' < z$ and $y \leq y'$, but then y' contradicts the assumption that z is minimal. Hence no such z exists. ∎

Bare-bones theorem: When clients follow the bare-bones protocol, they achieve fork consistency whether or not the server obeys the protocol.

Proof: Using the $<$ relation on version structures to order their corresponding operations, the two conditions of the Protocol correctness theorem hold. Condition 1 holds by Proposition one client. condition 2 holds for the following reason. For any version structure q, the set $\{o \mid o \leq q\}$ of all version structures less than q is totally ordered by the No-join lemma. Therefore, the associated operations are fetch-modify consistent with $<$ as the happens-before relation by the Fetch-modify lemma. ∎

4.3 Increasing concurrency

The bare-bones protocol serializes all version structure updates with a global lock on the VSL—an unacceptable restriction for a real distributed file system. The full protocol therefore uses an additional mechanism to support concurrent version structure updates. The basic approach is for users to declare pending updates to their version structures (resulting from either file fetches or modifications) with signed *update certificates*. Other users can then concurrently perform non-conflicting operations on the file system.

An update certificate issued by user u has the form $\{\text{UPD}, u, n, H(y_u), \text{inode-list}\}_{K_u^{-1}}$. UPD is just a constant (the type of the signed message). n is u's new version number in the forthcoming version structure. $H(y_u)$ is a collision-resistant hash of u's current entry in the VSL. Note that $n = y_u[u] + 1$, except when pipelining several updates. Finally, the update certificate also contains a list of i-table entries of the form $\langle \text{i-number}, \text{file handle} \rangle$ for any file inodes modified by the update, and $\langle \text{i-number}, \text{delta} \rangle$ for directory inodes. In the case of a fetch, inode-list is empty.

We now extend the version structure so that, in addition to client-version pairs, a version structure contains a (possibly empty) list of client-version-hash triples that reflect concurrent updates by other clients. A version structure must still contain one user-version pair for each user, but may also contain user-version-hash triples for consecutive version numbers up to and including the version number in any user-version pair.

The hash values are either a reserved value, $\perp$, or else the output of a function V whose domain is version structures. Informally, V puts the elements of a version structure into canonical form, removes the i-handle, and computes a collision resistant hash of the result. Specifically, given a version structure

$$x = \{\text{VRS}, u, h, u_1\text{-}n_1\ u_2\text{-}n_2\ \ldots, u_k\text{-}n_k\text{-}h_k\ \ldots\}$$

we compute $V(x)$ as follows: First remove the i-handle, h. Then sort the user-version pairs by user, and the user-version-hash triples by user and version. Finally output a collision-resistant hash H of the remaining, sorted fields of x.

To simplify the proof, we shall from now on assume that every version structure x owned by u contains the triple u-$x[u]$-$\perp$.

Definition 10 *We define the $\leq$ relation on extended version structures as follows. Given two version structures, x and y, we say $x \leq y$ iff the following two conditions hold:*

1. *For all users u, $x[u] \leq y[u]$ (i.e., $x \leq y$ by the old definition).*
2. *For each user-version-hash triple u-n-h in y, one of the following conditions must hold:*
   (a) *$x[u] < n$ (x happened before the pending operation that u-n-h represents), or*
   (b) *x also contains u-n-h (x happened after the pending operation and reflects the fact the operation was pending), or*
   (c) *x contains u-n-$\perp$ and $h = V(x)$ (x was the pending operation).*

We say $x = y$ if x and y have identical contents except possibly for the i-handle. We say $x < y$ iff $x \leq y$ and $x \neq y$.

Proposition: The $\leq$ relation on version structures is transitive.

Proof: Let $x \leq y$ and $y \leq z$. If $x = y$ or $y = z$, then the proposition is trivially true. Assume $x < y$ and $y < z$. Condition 1 follows from the fact that (using numerical $\leq$) $x[u] \leq y[u] \leq z[u]$ for all u. For each user-version-hash u-n-h in z, if $y[u] < n$, then we have $x[u] \leq y[u] < n$. On the other hand, if y also contains u-n-h, then either $x[u] < n$, or x contains u-n-h, or x contains u-n-$\perp$ and $V(x) = h$. ∎

The server now maintains a *pending version list*, or PVL, in addition to the VSL. The PVL consists of a set of update-certificate, unsigned-version-structure pairs, $\langle\{\text{UPD}, u, n, H(y_u), \text{inode-list}\}, \ell\rangle$. An update certificate declares an upcoming version structure that will become part of the VSL, at which point the update certificate will be removed from the PVL. The unsigned version structure, ℓ, has the same contents as u's upcoming version structure, except that ℓ has the value $\perp$ instead of an i-handle.

User u now performs the following steps to update its entry in the VSL, depicted graphically in Figure 3:

1. u sends the server an update certificate including its new version number and any modified inodes.
2. The server sends the VSL and PVL to u, along with any old version structures no longer in the VSL but still referenced by update certificates in the PVL. (These old version structures are needed only when pipelining updates.)
3. u sanity-checks all data received from the server in the previous step. All digital signatures must verify. All hashes in update certificates must match version structures signed by the same user. All version structures (signed and unsigned) must be totally ordered by the new $<$ relation. All old version structures must be less than the same user's entry in the VSL. All version numbers in a particular client's update certificates must be consecutive and start at one greater than the version structure in the VSL. The PVL must include the update certificate u signed in step 1 (and none signed by u with greater version numbers—such certificates might exist if there are pipelined updates).

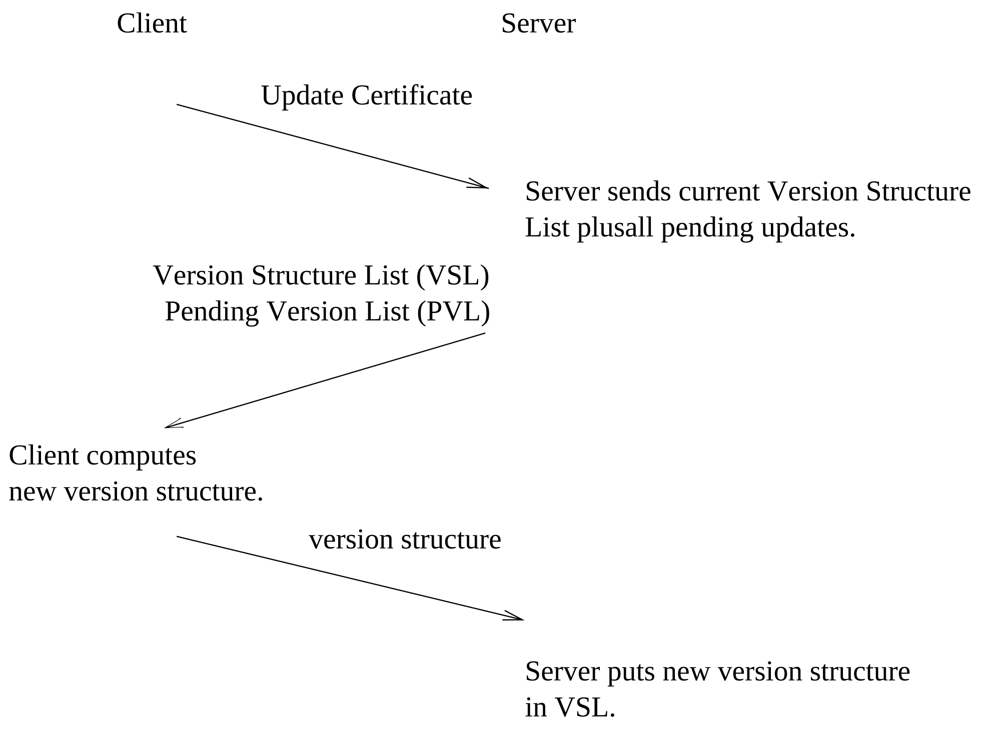
Client
Server
Update Certificate
Server sends current Version Structure List plusall pending updates.
Version Structure List (VSL)
Pending Version List (PVL)
Client computes new version structure.
version structure
Server puts new version structure in VSL.

Figure 3: The concurrent bare-bones protocol.

4. u initializes a new version structure x, by processing the VSL as in the bare-bones protocol. Then, for each client with an update certificate, u increases the corresponding user's version number in x to match the version number n in the update certificate. If one client has multiple update certificates, u takes the one with the highest version number. Since u's own update certificate is in the PVL, $x[u]$ will contain the version number from step 1.

5. For every entry $\langle\{\text{UPD}, v, n, H(y_v), \text{inode-list}\}, \ell\rangle$ in the PVL except the one signed in step 1, u adds the triple v-n-$V(\ell)$ to x. For the update certificate in step 1, u adds u-$x[u]$-$\perp$ to x. Intuitively, this encodes the history of operations in the PVL into x.

6. For every version structure y in the VSL, u checks that $y < x$. For all unsigned version structures ℓ in the PVL, u checks that either $\ell < x$ or $\ell = x$ and corresponds to the update certificate from step 1.

7. u signs x and sends it to the server. The server checks that x is totally ordered with respect to the other version structures in the VSL and PVL.

8. u checks for a modify-fetch conflict. If u is fetching a file and the file is listed in one of the PVL's update certificates, there must be a pending modification to the file. In this case, u does not return from the fetch call immediately, but instead requests and waits for the server to send it the version structure corresponding to the latest version of the file. u checks that this version structure matches the unsigned structure in the PVL.

Proposition concurrent one client: All version structures created by a single client that obeys the concurrent bare-bones protocol are totally ordered.

Proof: Immediate by step 4 of the protocol. ■

Concurrent fetch-modify lemma: Suppose all clients follow the concurrent bare-bones protocol. Let q be a version structure. Let **O** be the set of all completed operations by all clients satisfying $\mathbf{O} = \{o \mid \text{vs}(o) \leq q\}$, where $\text{vs}(o)$ designates the version structure of o. Whether or not the server obeys the protocol, if **O** is totally ordered by $<$, then **O** is fetch-modify consistent with $<$ as the happens-before relation.

Proof: We first must show that **O** is orderable using $<$ as the happens before relation. The execution of the protocol gives each operation a completion time after its issue time. Moreover, since $<$ totally orders **O**, it satisfies requirements 2–4 of a happens before relation. For the first requirement of happens before, suppose two version structures x and y are ordered and x's operation completed before y's was issued. Let u be the user that signed x, and v be the user that signed y. If $u = v$, then the protocol ensures $x < y$. Otherwise, since y was issued after x completed, u had already signed x at the time v signed the update certificate for the operation associated with y. No update certificate or version structure of y could have a version $\geq y[v]$ before v signed its update certificate. So, if x followed the protocol, then $x[v] < y[v]$. Since x and y are ordered, it must be that $x < y$.

Let y, signed by v, be the version structure corresponding to a fetch $F \in$ **O** of file f. By assumption, **O** must contain all operations with version structures less than y. Thus, any VSL entries that could have passed step 6 of the protocol when v signed y must be in $\mathbf{O}$. Let x, signed by u, be the greatest version structure less than y (and therefore in $\mathbf{O}$) associated with a modification of f. It follows that for any x' signed by u, if $x \leq x' < y$, then x' designates the same contents for f as x. Let x' be u's entry in the VSL upon which y is based. If $x \leq x'$, then F must have returned the same contents for f as designated by x's i-handle. If, on the other hand, $x' < x < y$, y must have seen an update certificate for version $x[u]$ of user u. But then by step 8 of the protocol, the fetch would have waited for x before returning, and thus would base the file contents returned on x's i-handle. ∎

Concurrent no join lemma: Suppose there are two version structures x and y such that $x \not\leq y$ and $y \not\leq x$. If clients follow the protocol, no client will sign any version structure greater than both x and y.

Proof: As before, for any client c and number n, c will sign at most one version structure t with $t[c] = n$. Moreover, any two version structures signed by c are ordered.

Assume that there exists a version structure w such that $x < w$ and $y < w$. There must be at least one minimal version structure $z \leq w$ such that $x \leq z$ and $y \leq z$. Let L be the VSL and P be the PVL that were sent to the client that signed z and from which this client calculated z.

We first note that there must exist some x' in L or P such that $x \leq x' < z$. To see this, let u be the user that signed x, and let x'' be u's entry in L. If $x \leq x''$, then we just let $x' = x''$. On the other hand, if $x'' < x$, then $x''[u] < x[u]$, and hence z must contain some triple u-$x[u]$-$V(x)$, which the signer of z would have included only if some $\ell_x = x$ appeared in P. In this case we set $x' = \ell_x$.

By a similar argument, there exists some y' in L or P such that $y \leq y' < z$. All version structures in L and P must be ordered by the sanity check step of the protocol, so in particular x' and y' are ordered. Assume without loss of generality that $x' < y'$. We then have $x \leq x' < y' < z$ and $y \leq y'$, but then y' contradicts the assumption that z is minimal, and hence no such z exists. ∎

Concurrent Bare-Bones theorem: When clients follow the concurrent bare-bones protocol, they achieve fork consistency whether or not the server obeys the protocol.

Proof: Using the $<$ relation on version structures to order their corresponding operations, the two conditions of the Protocol correctness theorem hold. Condition 1 holds by Proposition concurrent one client. Condition 2 holds for the following reason. For any version structure q, the set $\{o \mid o \leq q\}$ of all version structures less than q is totally ordered by the Concurrent no-join lemma. Therefore, the associated operations are fetch-modify consistent with $<$ as the happens-before relation by the Concurrent fetch-modify lemma. ∎

4.4 Generalizing to Groups

Until now, we have assumed that each user modifies files only in her own i-table. In practice, systems often contain group-writable files and directories that might be modified by several users. Groups in SUNDR are treated similarly to users. Every group has its own associated i-table, and each version structure additionally contains a version number for every group. This change requires substantial extensions to the protocol and proof, for which we refer the reader to the full version of this paper [12].

4.5 Pragmatic considerations

Client failures. In the concurrent protocols, a fetch waits for a conflicting modification. If the modification never completes, then the fetch waits forever. However, this is not necessary. The fetch could time out and return an error code, and the user could then sign the same version structure it would have signed had it gotten the version structure it was waiting for. With this change, we could still construct a forking tree on all completed operations except the fetches that returned error codes.

When a client failure causes an incomplete modify operation, the user can repair the situation by logging into a working client and reissuing the modification. All information necessary to reissue the modification is included in the inode-list of the update certificate.

Malicious clients can write spurious data to files that they own, but signatures prevent them from writing data to any other files. They can send the server version structures that destroy the total ordering of the VSL, but a well-behaved server will refuse such updates. Even when bad clients collude with a bad server, the set of completed operations by good clients on files that no bad client has permission to write will still have fork consistency. In particular, If c_1 and c_2 follow the protocol, c_1 misses an update of file f by c_2, and no bad client can write f, then as before, c_1 and c_2 can never again see each other's updates.

Users logged into multiple clients. We have assumed so far that each user runs on a single client. This is not realistic. However, the relationship between clients and users is analogous to the one between users and groups. So, one approach would be to create a version structure entry for each client a user is logged into. There is an optimization in which, while each user's version structure must still contain a version number for every client that user is logged into, for users other than the signer, the version structure need only contain a version number for one client—the one to make the most recent modification on behalf of that user. There is another optimization that allows clients eventually to stop appearing in version structures after a user has logged out. A full description of these optimizations is beyond the scope of this paper.

Bandwidth optimizations. Note finally that the server does not need to send the full VSL in response to each update certificate, but can instead send only new version structures since the last operation by the same client.

5. SUMMARY

We have described SUNDR, a network file system whose protocol makes even Byzantine file server failures readily detectable. Through digital signatures and a novel consistency protocol, SUNDR automatically detects almost any incorrect or malicious behavior on the part of the server. The only attack not immediately detectable is effectively to create an exact replica of a file system and partition users so that one group of users sees each replica and the two groups' operations are entirely concealed from each other.

Even this attack is detectable however, if users have any ability to communicate out-of-band. A simple pinging protocol, a trusted version-verification server, and even informal human communication are sufficient to reveal such a partitioning attack.

6. ACKNOWLEDGMENTS

We'd like to thank Allan Gottlieb, M. Frans Kaashoek, David Molnar, Robert Morris, and Marc Waldman for their helpful comments.

7. REFERENCES

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roseli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996. Also appears in Proceedings of the of the 15th Symposium on Operating System Principles.

[2] 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.

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

[4] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS*, pages 34–43, 2000.

[5] 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.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Chateau Lake Louise, Banff, Canada, 2001. ACM.

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

[8] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, May 1999.

[9] 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.

[10] 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, October 2000.

[11] 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, 1999. ACM.

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

[13] 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.

[14] Ethan Miller, Darrell Long, William Freeman, and Benjamin Reed. Strong security for distributed file systems. In *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, pages 34–40, Phoenix, AZ, April 2001.

[15] David Reed and Liba Svobodova. Swallow: A distributed data storage system for a local network. In A. West and P. Janson, editors, *Local Networks for Computer Communications*, pages 355–373. North-Holland Publ., Amsterdam, 1981.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[17] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *Proceedings of the 2001 USENIX*. USENIX, June 2001.

[18] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applicatio ns. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.