Secure Untrusted Data Repository (SUNDR)

Jinyuan Li, Maxwell Krohn,* David Mazières, and Dennis Shasha
NYU Department of Computer Science

Abstract

SUNDR is a network file system designed to store data securely on untrusted servers. SUNDR lets clients detect any attempts at unauthorized file modification by malicious server operators or users. SUNDR's protocol achieves a property called *fork consistency*, which guarantees that clients can detect any integrity or consistency failures as long as they see each other's file modifications. An implementation is described that performs comparably with NFS (sometimes better and sometimes worse), while offering significantly stronger security.

1 Introduction

SUNDR is a network file system that addresses a longstanding tension between data integrity and accessibility. Protecting data is often viewed as the problem of building a better fence around storage servers—limiting the number of people with access, disabling unnecessary software that might be remotely exploitable, and staying current with security patches. This approach has two drawbacks. First, experience shows that people frequently do not build high enough fences (or sometimes entrust fences to administrators who are not completely trustworthy). Second and more important, high fences are inconvenient; they restrict the ways in which people can access, update, and manage data.

This tension is particularly evident for free software source code repositories. Free software projects often involve geographically dispersed developers committing source changes from all around the Internet, making it impractical to fend off attackers with firewalls. Hosting code repositories also requires a palette of tools such as CVS [4] and SSH [35], many of which have had remotely exploitable bugs.

Worse yet, many projects rely on third-party hosting services that centralize responsibility for large numbers of otherwise independent code repositories. `sourceforge.net`, for example, hosts CVS repositories for over 20,000 different software packages. Many of these packages are bundled with various operating system distributions, often without a meaningful audit. By compromising sourceforge, an attacker can therefore introduce subtle vulnerabilities in software that may eventually run on thousands or even millions of machines.

Such concerns are no mere academic exercise. For example, the Debian GNU/Linux development cluster was compromised in 2003 [2]. 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. Similar attacks have also succeeded against Apache [1], Gnome [32], and other popular projects.

Rather than hope for invulnerable servers, we have developed SUNDR, a network file system that reduces the need to trust storage servers in the first place. SUNDR cryptographically protects all file system contents so that clients can detect any unauthorized attempts to change files. In contrast to previous Byzantine-fault-tolerant file systems [6, 27] that distribute trust but assume a threshold fraction of honest servers, SUNDR vests the authority to write files entirely in users' public keys. Even a malicious user who gains complete administrative control of a SUNDR server cannot convince clients to accept altered contents of files he lacks permission to write.

Because of its security properties, SUNDR also creates new options for managing data. By using SUNDR, organizations can outsource storage management without fear of server operators tampering with data. SUNDR also enables new options for data backup and recovery: after a disaster, a SUNDR server can recover file system data from untrusted clients' file caches. Since clients always cryptographically verify the file system's state, they are indifferent to whether data was recovered from untrusted clients or resided on the untrusted server all along.

This paper details the SUNDR file system's design and implementation. We first describe SUNDR's security protocol and then present a prototype implementation that gives performance generally comparable to the popular

*now at MIT CS & AI Lab

NFS file system under both an example software development workload and microbenchmarks. Our results show that applications like CVS can benefit from SUNDR's strong security guarantees while paying a digestible performance penalty.

2 Setting

SUNDR provides a file system interface to remote storage, like NFS [29] 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 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 explains the protocol, while Section 5 describes the server design.

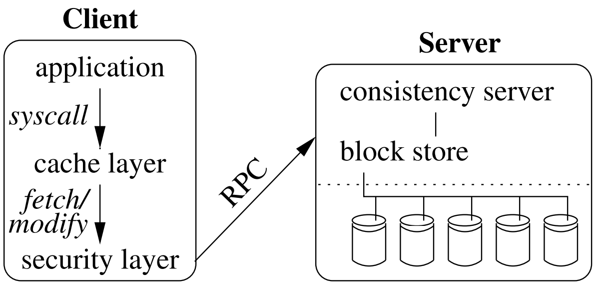

Figure 1: Basic SUNDR architecture.

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 an account, 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 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 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.[2] We call this property *fetch-modify consistency*.

If the server is dishonest, clients enforce a slightly

[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.

[2] Formally, *happens before* can be any irreflexive partial order that preserves the temporal order of non-concurrent operations (as in Linearizability [11]), orders any two operations by the same client, and orders a modification with respect to any other operation on the same file.

weaker property called *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 [16], and, assuming digital signatures and a collision-resistant hash function, proven SUNDR's protocol achieves it [17]. 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.

In order to discuss the implications of fork consistency and to describe SUNDR, we start with a simple straw-man file system that achieves fork consistency at the cost of great inefficiency (Section 3.1). We then propose an improved system with more reasonable bandwidth requirements called "Serialized SUNDR" (Section 3.3). We finally relax serialization requirements, to arrive at "concurrent SUNDR," the system we have built (Section 3.4).

3.1 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' 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 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 5, 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.

3.3 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 3.4 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* [18]. Second, each i-handle is tied to the latest version of every other i-handle using *version vectors* [23].

3.3.1 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 [9, 20], SUNDR names all on-disk data structures by cryptographic handles. The block store indexes most persistent data structures by their 20-byte SHA-1hashes, 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 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 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 $\langle \mathrm{principal}, \mathrm{i\text{-}number} \rangle$ 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 $\langle \mathrm{principal}, \mathrm{i\text{-}number} \rangle$ 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 multiple 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

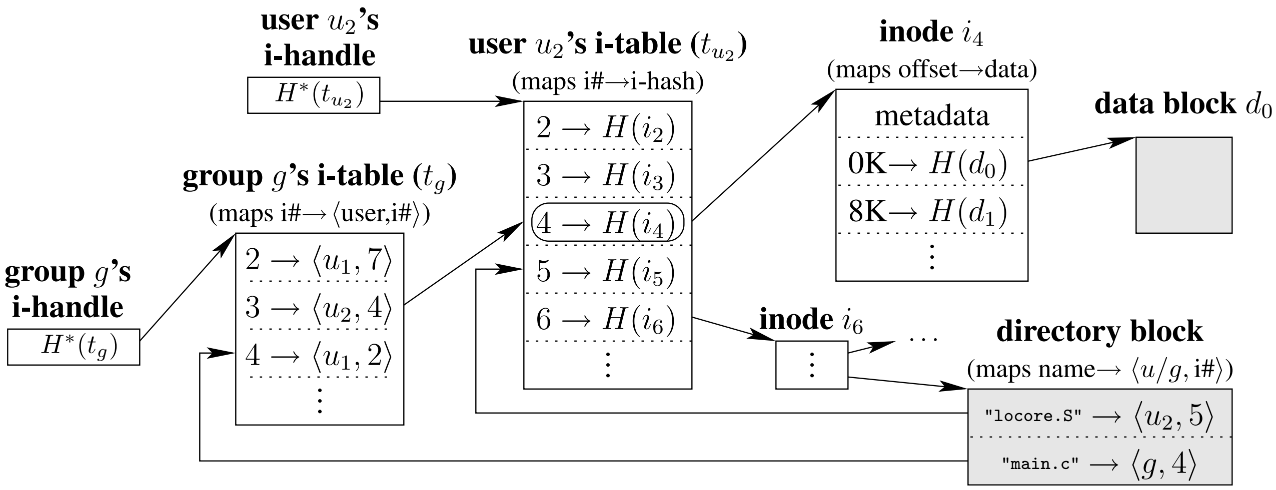

Figure 2: User and group i-handles. An *i-handle* is the root of a hash tree containing a user or group *i-table*. (H denotes SHA-1, while H^* denotes recursive application of SHA-1 to compute the root of a hash tree.) A *group i-table* maps group inode numbers to user inode numbers. A *user i-table* 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.

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.

3.3.2 Protocol

i-handles are stored in digitally-signed messages known as *version structures*, shown in Figure 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,

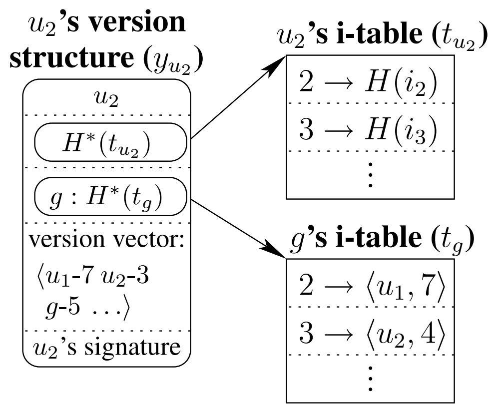

Figure 3: A version structure containing a group i-handle.

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

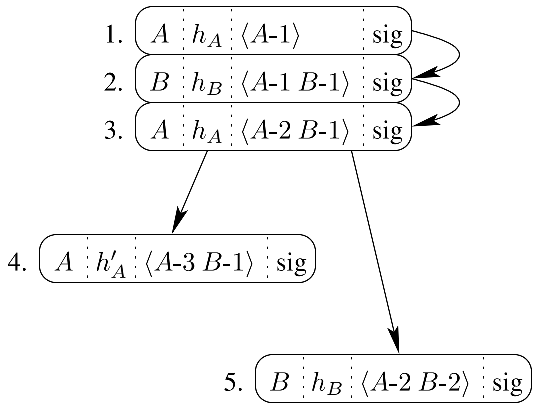

Figure 4: Signed version structures with a forking attack.

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 4 revisits the forking attack from the end of Section 3.1, 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 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 earlier work [16]).

One optimization worth mentioning is that SUNDR amortizes the cost of recomputing hash trees over several operations. As shown in Figure 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

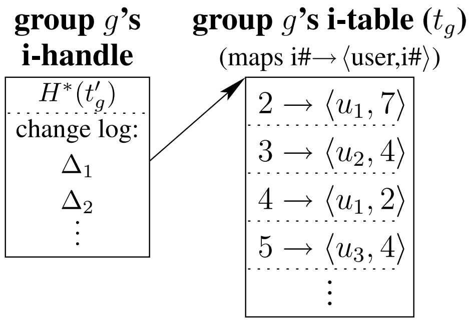

Figure 5: i-table for group g, showing the change log. t'_g is a recent i-table; applying the log to t'_g yields t_g.

when re-validating a cached file that has not changed since the hash tree root was last computed.

3.4 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.[3]

3.4.1 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 $\big(y_u[u] + 1$, unless u is pipelining multiple updates$\big)$,
- a hash of u's VSL entry $(H(y_u))$, and
- 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 $\langle \text{user/group}, \text{i\#} \rangle$.

[3]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, 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.

- 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 $\mathcal{O}_1$ is reflected in the VSL or PVL returned for $\mathcal{O}_2$'s UPDATE RPC, then we say $\mathcal{O}_1$ *happened before* $\mathcal{O}_2$. Conversely, if $\mathcal{O}_2$ is reflected in $\mathcal{O}_1$'s VSL or PVL, then $\mathcal{O}_2$ happened before $\mathcal{O}_1$. 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 $\langle$update certificate, unsigned version structure$\rangle$ 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.

3.4.2 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.)

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.

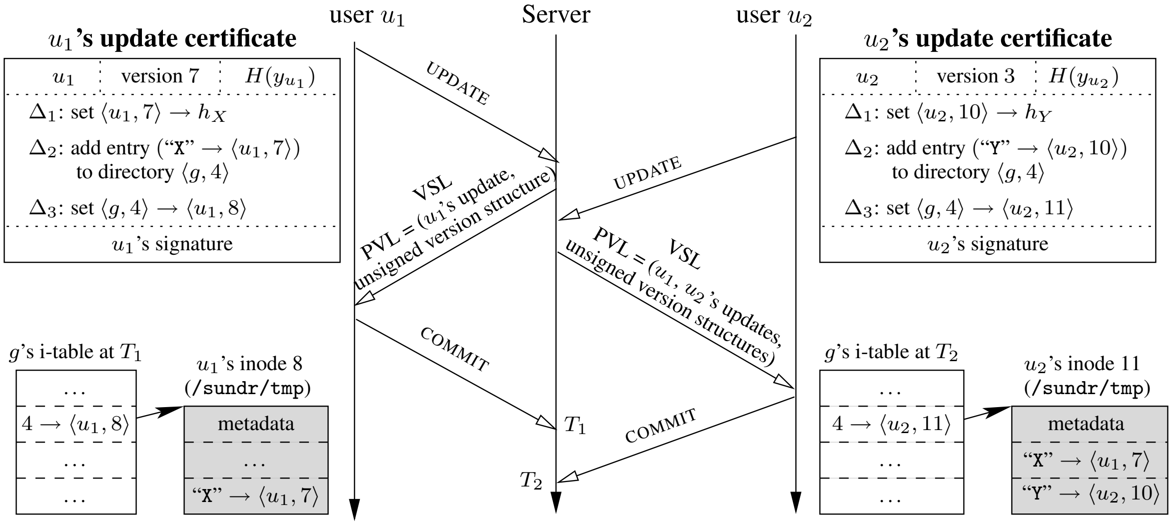

Figure 6: Concurrent updates to /sundr/tmp/ by different users.

3.4.3 Example

Figure 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 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 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.

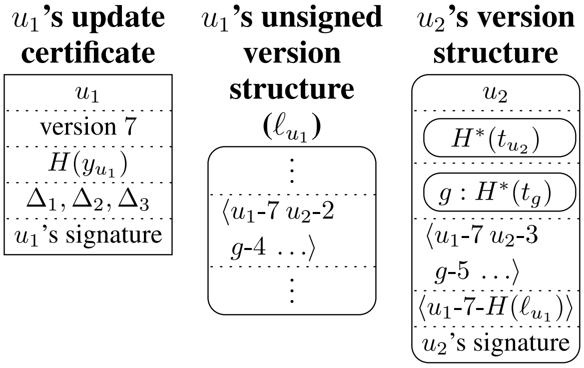

Figure 7: A pending update by user u_1, reflected in user u_2's version structure.

4 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 [13, 22], 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 [5, 10, 12, 34].

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.

5 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 [33] 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.

5.1 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.

5.2 Signature optimization

The cost of digital signatures on the critical path in SUNDR is significant. Our implementation therefore uses the ESIGN signature scheme,[4] 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.

5.3 Consistency server

The consistency server orders operations for SUNDR clients and maintains the VSL and PVL as described in Section 3. 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 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).

[4] Specifically, we use the version of ESIGN shown secure in the random oracle model by [21], with parameter $e = 8$.

6 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 [24] 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 7 we describe how SUNDR's performance would improve if *bstor* had a small amount of NVRAM to store update certificates.

6.1 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 STOREs then DECREFs 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.

6.2 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.

6.3 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.

7 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.

7.1 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.

7.2 Microbenchmarks

7.2.1 *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 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 STOREing 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*

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 8: *bstor* throughput measurements with the block cache disabled.

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 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 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.

7.2.2 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.

7.3 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 5.3 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.

7.3.1 LFS small file benchmark

The LFS small file benchmark [28] 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 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

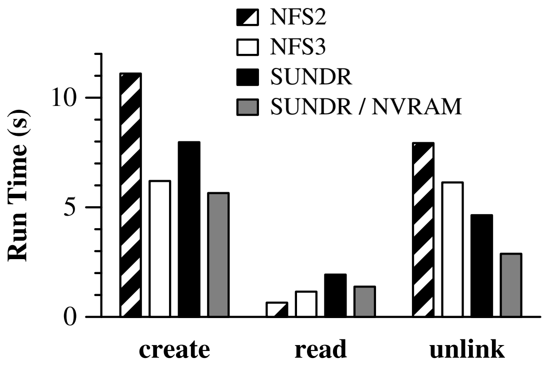

Figure 9: Single client LFS Small File Benchmark. 1000 operations on files with 1 KB of random content.

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 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.

7.3.2 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 measur-

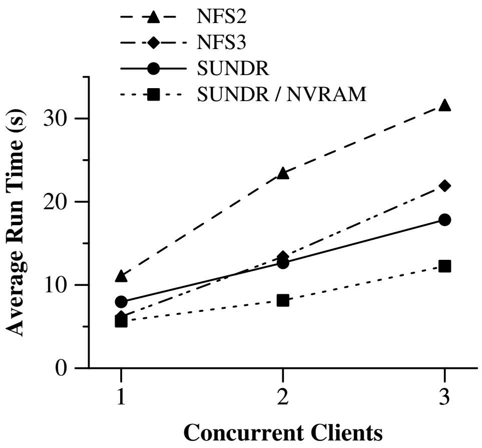

Figure 10: Concurrent LFS Small File Benchmark, **create** phase. 1000 creations of 1 KB files. (Relative standard deviation for SUNDR in 3 concurrent clients case is 13.7%)

ing 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.

7.3.3 Real workloads

Figure 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.

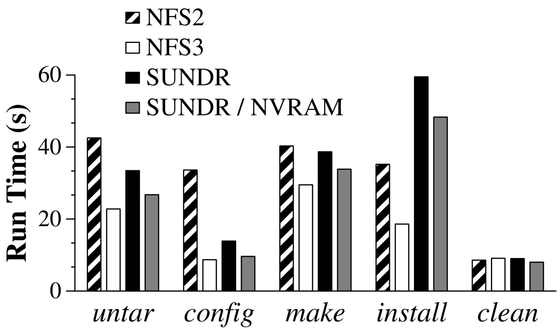

Figure 11: Installation procedure for `emacs_20.7`

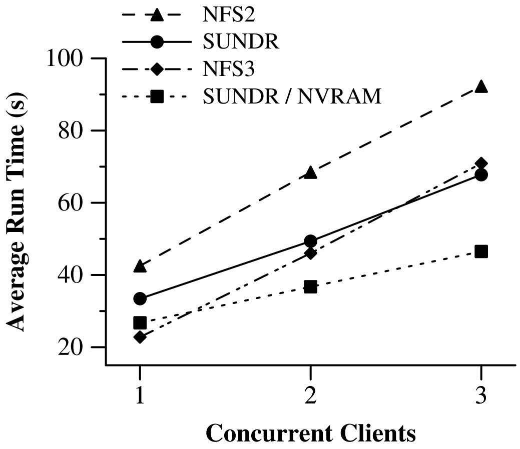

Figure 12: Concurrent *untar* of `emacs_20.7.tar`

Figure 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 benchmarks extends to real file system workloads.

7.3.4 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 13 shows the results.

SUNDR fares badly on the commit phase because CVS repeatedly opens, memory maps, unmaps, and closes each

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

Figure 13: Run times for CVS experiments (in seconds).

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-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.

8 Related work

A number of non-networked file systems have used cryptographic storage to keep data secret [5, 34] and check integrity [31]. Several network file systems provide varying degrees integrity checks but reduce integrity on read sharing [25] or are vulnerable to consistency attacks [10, 12, 19]. SUNDR is the first system to provide well-defined consistency semantics for an untrusted server. An unimplemented but previously published version of the SUNDR protocol [16] had no groups and thus did not address write-after-write conflicts.

The Byzantine fault-tolerant file system, BFS [6], 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. However, SUNDR provides weaker freshness guarantees than BFS, because of the possibility that a malicious SUNDR server can fork the file system state if users have no other evidence of each other's on-line activity.

Several projects have investigated storing file systems on peer-to-peer storage systems comprised of potentially untrusted nodes. Farsite [3] spreads such a file system across people's unreliable desktop machines. CFS [7] is a secure read-only file P2P system. Ivy [20], a read-write version of CFS, can be convinced to re-order operations clients have already seen. Pond [27] relies on a trusted "inner core" of machines for security, distributing trust in a BFS-like way.

SUNDR uses hash trees, introduced in [18], to verify a file block's integrity without touching the entire file system. Duchamp [8], BFS [6], SFSRO [9] and TDB [14] have all made use of hash trees for comparing data or checking the integrity of part of a larger collection of data.

SUNDR uses version vectors to detect consistency violations. Version vectors were used by Ficus [22] to detect update conflicts between file system replicas, and have also been used to secure partial orderings [26, 30]. Our straw-man file system somewhat resembles timeline entanglement [15], which reasons about the temporal ordering of system states using hash chains.

9 Conclusions

SUNDR is a general-purpose, multi-user network file system that never presents applications with incorrect file system state, even when the server has been compromised. SUNDR's protocol provably guarantees fork consistency, which essentially ensures that the server either behaves correctly or that its failure will be detected after communication among users. In any event, the consequences of an undetected server compromise are limited to concealing users' operations from each other after some forking point; the server cannot tamper with, inject, re-order, or suppress file writes in any other way.

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 placed in the server, SUNDR both increases people's options for managing data and significantly improves the security of their files.

Acknowledgments

Thanks to Michael Freedman, Kevin Fu, Daniel Giffin, Frans Kaashoek, Jinyang Li, Robert Morris, the anonymous reviewers, and our shepherd Jason Flinn.

This material is based upon work supported by the National Science Foundation (NSF) under grant CCR-0093361. Maxwell Krohn is partially supported by an NSF Graduate Fellowship, David Mazières by an Alfred P. Sloan research fellowship, and Dennis Shasha by NSF grants IIS-9988636, MCB-0209754, and MCB-0115586.

References

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

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

[3] 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.

[4] Brian Berliner. CVS II: Parellizing software development. In *Proceedings of the Winter 1990 USENIX*, Colorado Springs, CO, 1990. USENIX.

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

[6] 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.

[7] 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, October 2001. ACM.

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

[9] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.

[10] 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.

[11] 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.

[12] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. 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.

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

[14] 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.

[15] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] Michael Reiter and Li Gong. Securing causal relationships in distributed systems. *The Computer Journal*, 38(8):633–642, 1995.

[27] 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.

[28] M. Rosenblum and J. 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.

[29] 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.

[30] Sean W. Smith and J. D. Tygar. Security and privacy for partial order time. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 70–79, Las Vegas, NV, October 1994.

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

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

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

[34] 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.

[35] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.