Security and Decentralized Control in the SFS Global File System

by

David Mazières

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1997

[redacted]

Author ....
Department of Electrical Engineering and Computer Science
August 29, 1997

Certified by..
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Accepted by..
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Security and Decentralized Control in the SFS Global File System

by

David Mazières

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 1997, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

SFS (secure file system) is a global file system designed to be secure against all active and passive network attacks, provide a single namespace across all machines in the world, and avoid any form of centralized control. Using SFS, any unprivileged user can access any file server in the world securely and with no prior arrangement on the part of system administrators. New file servers are immediately accessible securely by all client machines. Users can name such new servers from any existing file system by specifying public keys in symbolic links. A prototype implementation of SFS installs easily, coexists with other file systems, and should port trivially to most UNIX platforms. Preliminary performance numbers on application benchmarks show that, despite its use of encryption and user-level servers, SFS delivers performance competitive with an in-kernel NFS implementation.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Acknowledgments

The SFS project is joint work with Emmett Witchel and M. Frans Kaashoek. Eddie Kohler helped write a paper that this thesis is based on.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents SFS, the first global file system to resist all active and passive network attacks while avoiding any form of centralized control. The explosion of the World-Wide Web over the past few years has clearly driven home the benefits of a decentralized system for sharing information over the wide area network. By comparison, the limited acceptance of secure HTTP has shown the degree to which centralized control can hinder the deployment of any technology. All other things equal, people should clearly prefer to run secure web servers over insecure ones. In practice, however, the cost and hassle of obtaining the necessary digital certificates from central certification authorites has kept most web servers from running secure HTTP. The SFS project proves that secure global network services can function with completely decentralized trust.

As a global file system, SFS offers a single file namespace on all machines. Users can reach any SFS file system from any SFS client in the world, simply by referencing that file system's pathname. The SFS protocol then cryptographically guarantees the integrity of all file data and the secrecy of any data not universe-readable. These guarantees hold over untrusted networks and across administrative realms; they require no shared secrets and apply even when people use servers they don't have accounts on. SFS never operates in a degraded or insecure mode. Unlike any previous file system, SFS achieves its security without granting control of the namespace to any naming authority. Clients can immediately see new file servers as they come on-line, with no need to reconfigure clients or register new servers in a centrally or hierarchically maintained database.

Many existing file systems encourage the formation of inconveniently large administrative realms, either to maintain security without restricting file sharing or to save users from unreasonable complexity. In contrast, we specifically built SFS for cross-administrative realm operation. SFS's global namespace already assures users access to any servers they need from any clients they use. In addition, user-controlled agent processes transparently authenticate users to any file servers they access. Thus, users can run a single authentication agent at login time and never need to worry about separately authenticating themselves to multiple servers.

SFS employs several novel techniques to avoid centralized control. First, it embeds cryptographic key hashes in pathnames to name file systems by their public keys. This simple, egalitarian naming scheme ensures that every client has at least one path to every SFS file system, and that clients can cryptographically protect any communications with file servers. Of course, anyone can assign additional, human-readable names to a file system through symbolic links. In this way, certification authorities consist of nothing more than ordinary SFS file servers that name file systems through symbolic links. Second, SFS allows read-only servers such as certification authorities to enjoy strong integrity guarantees and high performance through precomputed digital signatures. An off-line utility can create signature databases that prove the contents of file systems. Servers of read-only data can use these databases both to avoid the cost of any private key operations and to avoid keeping any on-line copies of a file system's private key. Finally, SFS combines many additional features often missing from network file systems, including tight consistency guarantees, automatic mapping of user and group identities between administrative realms, global, non-blocking automounting, and flexible, open-ended user authentication.

A reference implementation of SFS makes several contributions to user-level filesystems. SFS runs as a user-level NFS loopback file server—an ordinary, local process acting like a remote NFS server. It should consequently port trivially any Unix platform supporting NFS. Pervious NFS loopback file systems have exhibited poor performance. However, SFS makes use of a new asynchronous RPC library that allows arbitrarily many outstanding file operations. This non-blocking architecture overlaps the cost of encryption with network latency and lets servers achieve disk arm scheduling for multiple requests. SFS further uses aggressive on-disk client caching, with the result that our user-level implementation performs comparably to an in-kernel implementation of NFS, despite SFS additionally providing security and consistency.

We designed SFS to serve existing file systems and coexists with other network file systems, which will make it easy to install and gradually adopt. Our implementation uses a modular structure that allows multiple versions of the client and server to run on the same machine and share the same namespace. This structure should help innovation and prevent backwards compatibility from ever unreasonably burdening future revisions of the software.

The rest of this thesis discusses the design of SFS and the SFS protocol, a prototype imlementation of SFS, and performance measurements of the prototype implementation.

Chapter 2

Related Work

Many projects have explored distributed file systems in the context of caching, performance, security, or a global namespace [3, 7, 8, 11, 12, 13, 18, 21, 22, 23, 34, 39, 44, 45, 46, 48, 49, 53, 57]. While SFS's design goals have been met individually by previous projects, SFS is the first file system to provide security, a global namespace, and completely decentralized control at the same time.

One of the file systems in most widespread use today is Sun's NFS [44, 54]. NFS is purely a local-area file system; it doesn't perform well over wide area networks, and offers no security in the face of network eavesdropping. A later attempt to add security to NFS through secure RPC [55] had a very appealing design based on public key cryptography. Unfortunately, the protocol had several flaws. First, the public key lengths were far too short [30]. Second, the authentication information in RPC packets was not cryptographically tied to their contents, allowing attackers to tamper with request packets. Third, file data on the network was sent in cleartext, which may be acceptable for local area networks, but certainly will not work across the Internet.

AFS [24, 46] is probably the most successful global file system to date. It provides a clean separation between the local and global namespace by mounting all remote file systems under a single directory, `/afs`. AFS also makes extensive use of client caching to achieve good performance on remote file systems, and doesn't trust client machines beyond their authenticated users. Unlike SFS, however, AFS client machines contain a fixed list of available servers that only a privileged administrator can update. AFS uses Kerberos [52] shared secrets to protect network traffic, and so cannot guarantee the integrity of file systems on which users do not have accounts. Though AFS can be compiled to encrypt network communications to servers on which users have accounts, the commercial binary distributions in widespread use do not offer any secrecy. DFS [27] is a second generation file system based on AFS, in which a centrally maintained database determines all available file systems.

The Echo distributed file system [8, 9, 31, 32] achieves secure global file access without global trust of the authentication root. Each Echo client and server attaches to a particular point in the global namespace, which also forms a hierarchy of trust. Clients need not go through the root of this trust hierarchy to access servers with which they share a common prefix in the namespace. However, the root of the hierarchy is completely centralized. Users cannot reach new servers until someone has consented to attach those servers to some point in the namespace. Echo also does not allow for any local namespace, and changes the meaning of the root directory—interesting ideas, but ones that would prevent it from coexisting easily with other file systems.

The Truffles service [40] is an extension of the Ficus file system [23] to operate securely across the Internet. Truffles provides fine-grained access control with the interesting property that, policy permitting, a user can export files to any other user in the world, without the need to involve administrators. Unfortunately, the interface for such file sharing is somewhat clunky, and involves exchanging E-mail messages signed and encrypted with PEM. Truffles also relies on centralized, hierarchical certification authorities, naming users with X.500 distinguished names and requiring X.509 certificates for all users and servers.

Recently, Sun has introduced some extensions to NFS called WebNFS [14, 15] and aimed at replacing HTTP. WebNFS optimizes connection setup relative to NFS, and also allows anonymous file access through a "public" file handle, in effect providing a global file system. However, WebNFS isn't necessarily intended to be used as a file system, but rather as better protocol for user-level web browsers. Moreover, because most NFS server implementations are in the kernel, a few changes to an operating system's NFS code are all that is required to give WebNFS servers many of the performance benefits research systems like SPIN [6] get by downloading web servers into the kernel. Sun claims to have achieved an order of magnitude better performance from WebNFS than from an ordinary Web server. The small additions to NFS which constitute WebNFS greatly enhance its utility as a global file system, but do not address NFS's security problems.

Instead of speeding up the Web by replacing HTTP with a well-known network file system protocol, WebFS [56] implements a network file system on top of the HTTP protocol. Specifically, WebFS uses the HTTP protocol to transfer data between user-level HTTP servers and an in-kernel client file system implementation. This allows the contents of existing URLs to be accessed through the file system. WebFS attempts to provide authentication and security through a protocol layered over HTTP [4]; authentication requires a hierarchy of certification authorities.

WebFS lets normal file utilities manipulate the contents of remote web sites. It does not, however, address any of HTTP's serious shortcomings as a file sharing protocol. HTTP transfers the entire contents of a file before allowing clients to issue more requests. It does not allow reads at arbitrary offsets of a file. It requires clients to waste TCP connections if they want multiple outstanding requests. It often does not even allow clients to list the contents of directories. WebFS avoids some of these problems on special WebFS servers through a somewhat enhanced and incompatible version of HTTP. There really is no particular advantage to basing a new file system protocol on HTTP, however. True, some network firewall administrators allow HTTP to the outside world; WebFS can subvert their intentions by tunneling a file system over HTTP. The right thing to do, however, is to build a secure file system from the ground up, and to let people open their firewalls to it on its own merits.

Chapter 3

The SFS Namespace

One must ask two fundamental questions about any global file system: Who controls the namespace, and how do clients trust remote machines to serve parts of that namespace? A centrally controlled namespace hinders deployment; it prevents new file servers from coming on-line until they gain approval from the naming authority. On the other hand, clients need cryptographic guarantees on the integrity and secrecy of remote communications; these guarantees require a binding between encryption keys and points in the global namespace. To provide this binding without relying on naming authorities, SFS names file servers by their public keys, an approach similar in spirit to the SDSI public key infrastructure [42].

Each SFS file system is mounted on a directory named `/sfs/`*Location*:*HostID*. *Location* is a DNS host name; *HostID* is a cryptographic hash of the server's public key and hostname. *Location* tells SFS where to find a server for the file system, while *HostID* specifies a private key that the server must prove possession of. When a user references a non-existent directory of the proper format under `/sfs`, an SFS client attempts to contact the machine named by *Location*. If that machine exists, runs SFS, and can prove possession of a private key corresponding to *HostID*, then the client transparently creates the referenced directory in `/sfs/` and mounts the remote file system there. This automatic mounting file systems under pathames derived from public keys assures every SFS file system a place in the global SFS namespace.

Of course, no person will ever want to type a *HostID*—the hexadecimal representation of a 160-bit cryptographic hash. Instead, people can assign human-readable names to mount points through symbolic links. For instance, if Verisign acted as an SFS certification authority, client administrators would likely create a symbolic link from `/verisign` to the the mount point of that certification authority—a pathname like `/sfs/sfs.verisign.com:75b4e39a1b58c265f72dac35e7f940c6f093cb8-0`. This file system would in turn contain symbolic links to other SFS mount points, so that, for instance, `/verisign/mit` might point to `/sfs/sfs.mit.edu:0f69f4a0-59c62b35f2bdac05feef610af052c42c`.

There is nothing magic about `/verisign`, however; it is just a symbolic link. Thus, SFS supports the use of certification authorities, but neither relies on them for correct operation nor grants them any special privileges or status. In fact, organizations will probably install analogous symbolic links to their own servers in the root directories of their client machines. Individual users can create symbolic links for themselves in their home directories, perhaps after exchanging Host IDs in digitally signed email. Anyone with a universe-readable directory can create public links that others can make direct use of. For instance, `/verisign/mit/lcs/dm/sfs-dist` could be a path to the SFS distribution server. The chain of trust is always explicit in such pathnames. Despite its not being officially certified, people can still reach the `sfs-dist` server and understand who has given it that name.

Certification authorities, hierarchical delegation, informal key exchange, the "web of trust," and even hard-coded encryption keys all have their place in certain situations. SFS recognizes that no single name to key binding process can be right for all purposes. By exposing public keys, then, it provides security without placing any restrictions on key management.

Chapter 4

Design and protocol

This chapter describes the overall design of SFS and the SFS protocol. The next chapter discusses a reference implementation of SFS for Unix operating systems. We cover two dialects of the SFS protocol in turn, a general-purpose read-write protocol and a read-only protocol for heavily accessed public data with stringent security requirements. All clients must support both protocols, while servers run whichever one best suits their requirements.

4.1 Read-write protocol

The read-write SFS protocol provides the usual read and write file system operations, along with public key based user authentication. It guarantees the secrecy, integrity, and freshness of all data transmitted over the network, and requires file servers to perform on-line private key operations. This section describes the details of authentication and caching in the read-write protocol. Figures 4-4 and 4-5 at the end of the chapter list all remote procedure calls in the protocol.

4.1.1 Authentication protocol

Because users can't necessarily trust file servers with their passwords, all user authentication to remote file servers uses public keys in SFS. Every user on an SFS client machine registers an authentication agent process with the SFS client software. The authentication agent holds one or more of the user's private keys. When a user accesses a new file system, the client machine contacts the user's agent and gives the agent a chance to authenticate the user to the remote server through one of its private keys. If the authentication succeeds, the client machine receives an authentication number from the server and tags all file system requests from that user with that authentication number. After authentication, the client also obtains information about the user's exact credentials on the remote machine, so as to translate the user and group IDs of remote files to something that makes some amount of sense on the local machine. If the authentication fails, all file accesses made by the user are tagged with the authentication number 0, which permits access only to universe-readable files. When the authentication agent exits, the client machine flushes any cached data associated with that user and de-allocates any authentication numbers obtained through that agent. With this scheme, all user authentication after the initial agent registration takes place completely transparently.

The SFS authentication protocol was inspired by ssh [59], but with several fixes [1] and modifications. It takes place in two stages: first, authenticating the server to the client, and second, authenticating the user to the server, via the client. Three parties are involved: the SFS client software C, the SFS server software S, and the user's authentication agent A. The only current implementation of the agent software holds all private keys on the same machine as C. Each user can choose to run his own agent implementation, however. In the future, we envision dumb agents that forward authentication requests to smart cards or even to other agents through encrypted login connections, as is currently done with ssh.

The first stage, shown in Figure 4-1, authenticates the server to the client, with the client initiating the exchange. The goal is to agree on two client-chosen shared session keys, K_{sc} and K_{cs}, to encrypt and protect the integrity of future data communication between client and server. The server sends the client two public keys, PK_s and PK_t: the first, PK_s, is the long-lived key whose hash is in the pathname; the second, PK_t, is a temporary public key which changes every hour. Use of a temporary public host ID $= \{PK_s, \text{hostname}\}_{\text{SHA-1}}$

session ID $= \{K_{cs}, K_{sc}, PK_t, N_0\}_{\text{SHA-1}}$

1. $C \rightarrow S$: hostname, host ID
2. $S \rightarrow C$: PK_s, PK_t, N_0
3. $C \rightarrow S$: $\{\{K_{cs}, K_{sc}\}_{PK_t}\}_{PK_s}$, session ID, N_0

Figure 4-1: Key exchange and server authentication.

key provides forward secrecy: even if SK_s, the long-lived secret key, is compromised, old communication cannot be decrypted without factoring PK_t once SK_t has been destroyed.

The *host ID* in this protocol is the hash obtained from the pathname by the client. The client sends the host ID and hostname to the server in its first message so the same machine can serve multiple file systems without needing multiple IP addresses. The *session ID*, which is a hash of the two session keys, the server's temporary public key, and a server-chosen nonce N_0, is used in future communication to identify the session. The plaintext echoing of N_0 has no effect on authentication proper, but can be used by the server to filter out certain denial-of-service attacks that would otherwise require significant CPU time due to the expense of public-key decryption [25].

After this initial stage, all traffic from C to S is encrypted with K_{cs} and all traffic from S to C with K_{sc}. The client can be sure it is talking to the server as only the server could have decrypted these session keys. At this point, C can access S with anonymous permissions.

The second stage of the protocol authenticates the user to the server via the client, as shown in Figure 4-2. This stage is necessary to access protected files. Here, PK_u is the user's public key. The user's agent A has a copy of the user's private key, SK_u. N_u is a nonce, or challenge, chosen by the server. Note that all messages between client and server are encrypted with one of the private session keys K_{sc} and K_{cs}.

1. $C \rightarrow A$: hostname, host ID, session ID
2. $A \rightarrow C \rightarrow_{K_{cs}} S$: PK_u
3. $S \rightarrow_{K_{sc}} C \rightarrow A$: $\{N_u\}_{PK_u}$
4. $A \rightarrow C \rightarrow_{K_{cs}} S$: $\{N_u, \text{hostname}, \text{host ID}, \text{session ID}\}_{\text{SHA-1}}$
5. $S \rightarrow_{K_{sc}} C$: authentication number and remote credentials (UIDs, GIDs, etc.)

Figure 4-2: Authenticating users.

The authentication information in the last step of the user authentication is used to establish some reasonable correspondence between local and remote user and group IDs. In the first step, note that the agent has access to both host name and host ID, allowing it to certify the host ID if desired; however, the agent never sees either K_{cs} or K_{sc}, the session keys which are encrypting all client–server communication.

4.1.2 Caching

Reasonable performance for any global file system depends critically on the effectiveness of client caching. Caching in SFS is even more important, given the cost of software encryption for any data fetched over the network.

The only way to build a portable Unix reference implementation of SFS was to use NFS loopback mounts on client machines. Unfortunately, the restricted information the NFS interface provides somewhat limits cache design choices. Again for portability, the cache could not ask for version numbers or other nonstandard information which would need to be provided by the underlying file system. The fact that SFS client machines are untrusted produced a further restriction: we did not want to allow one misbehaving client to hang another client for unreasonable amounts of time. Our solution is a lease-based cache mechanism [21] where clients must hold a server-granted lease, with explicit expiration time, before accessing any file.

The SFS client cache stores three types of objects: file data, name lookups, and file attributes. Both successful and unsuccessful name lookups are cached. File data is cached in 64 Kbyte chunks; a large file or device for backing store is used to enable a huge cache, offsetting the cost of decryption.

Caching specifics

In order to return a file's attributes or data to an application, the SFS client must hold a current lease on the file. This is true even if the file is cached. The client obtains leases automatically, for the most part: if a remote file is not being written by any other client, the client's get-attributes request will also return a read lease for the requested file. If, however, there is an outstanding write lease on the file, the SFS server first retrieves the file's size and modification time from the writing client. After some sanity checks, it then returns the attributes to the requesting client with a zero-length lease.

If a client requests data (not attributes) from a file in the process of being written, the server sends an unsolicited message to the writing client requesting that the client give up its write lease. The server will not reply to the read request until the lease is returned or expires. To prevent a slow client from overly slowing the system, SFS servers maintain a maximum write lease time per client. A client slow to relinquish write leases is punished by cutting down its maximum write time.

SFS servers will also send unsolicited messages revoking read leases when a write occurs. However, these revocations are unconditional—the server does not even wait for a reply. Thus, no per-client maximum read lease time is required.

A file can remain in the cache after its read lease has expired. The client must renew the lease if the cached data is to be used, however; at renewal time, the client must compare modification and inode modification times with the server's current version. A conflict forces the client to reread the file. A similar scheme is used in the name lookup cache; here, entries are invalidated only if a directory's times have changed. As an optimization, the server sends deltas to clients holding read leases on a directory rather than invalidate the entire cache entry.

The server performs access checks at the time it grants leases. Since a server only deals with client daemons, of which there is one per machine, the client must take responsibility for handling access control when a file is shared among users on a single machine—these users share each other's leases. To facilitate this, each piece of data in the cache is marked with an authentication number obtained from some user's authentication agent, if any. When an authentication agent exits, which the client detects immediately, any data associated with that authentication number is flushed.

Callbacks and leases

The inclusion of server-generated unsolicited messages enables much higher performance. Specifically, because servers can directly ask write-lease holders for information, writing clients do not need to write through all attribute or data changes to the server. This enables a write-behind cache, with the associated improvement in write performance. These callbacks are also used to inform reading clients of changes, enabling a longer attribute cache time—readers don't have to check for changes as often—and better consistency guarantees than NFS provides.

While a callback-based invalidation protocol, without leases, could create a workable system, explicit leases are useful for two reasons. First, the expiration times on read leases limit the amount of state a server needs to keep. Second, some method is required to limit the length of time a problem client can hold a write lock, since SFS does not trust client machines; leases present an attractive and simple alternative.

4.2 Algorithms

We chose the Rabin-Williams public key system [38, 58] for SFS for several reasons. Like RSA, the Rabin algorithm depends on the difficulty of factoring. However, while there may exist an easier way of breaking RSA than by factoring the modulus, the Rabin cipher is provably as hard as factoring. Moreover, the Rabin algorithm has fast signature verification time (twice as fast as RSA with an exponent of 3), a property we make use of for read-only data as described in Section 4.3. We also hope to avoid US patent restrictions by using an algorithm other than RSA.

We chose SHA-1 [19], a collision-free hash function, to compute the *public-key-hash* used in pathnames of remote servers. The hash result is 160 bits long, which is currently expressed as 40 characters of hexadecimal digits in pathnames. We may switch to a more compact notation such as base-36 (0–9 and a–z) before releasing SFS. SHA-1 is also used to compute the session ID and the answer to the user authentication challenge. We also use SHA-1 in conjunction with the PRab redundancy function [5] for digital signatures.

The shared-key system used for data communication—that is, with K_{cs} and K_{sc}—is a combination of the Arcfour stream cipher [26] for secrecy and an SHA-1-based message authentication code (MAC) to detect any modification of network packets. We chose these algorithms for their strong security and high efficiency when implemented in software. The combination of Arcfour and our SHA-1-based MAC is almost twice as fast as DES alone, while providing strong secrecy and integrity and longer encryption keys.

SFS can function with arbitrary size public and shared keys. We currently use 1,024 bit keys for user and server public keys, 768 bit keys for temporary public keys, and 128 bits for Arcfour shared keys and for nonces. However, the key lengths can change without breaking compatibility.

4.3 Read-only protocol

Key exchange at connection setup is fairly expensive for an SFS server; for example, the two public-key decryptions cost roughly 100 milliseconds on a 200 MHz Pentium Pro. This enforces a hard limit of 10 connections accepted per second, potentially preventing an SFS server from serving more than a few thousand clients. One possible solution would be to require large groups of clients to access SFS servers through a single trusted proxy; however, several studies [11, 33] have shown that such organization-wide caches have extremely poor hit rates. Another solution would be to replicate widely used servers; however, this would require widely distributing these servers' private keys—unacceptable if high security is a requirement. In fact, for some purposes, even a single on-line copy of a file server's private key may pose an unacceptable security risk.

To address these issues, SFS servers of public data can operate in a read-only mode that requires no on-line private key operations. This read-only mode uses pre-computed digital signatures to prove the contents of a file system to client machines. While creating digital signatures can be costly (e.g. 60 milliseconds on a 200 MHz Pentium Pro for 1,024 bit Rabin-Williams keys), Rabin-Williams signature verification is extremely fast. A 200 MHz Pentium Pro can verify 1,024 bit Rabin-Williams signatures in only 250 microseconds—considerably less than the cost of a remote procedure call on anything but the fastest local-area networks. This makes it reasonable for clients to verify signatures on every RPC.

All signed SFS messages begin with a signature header that has several fields. An RPC version number specifies the format of marshaled data; it should remain fixed unless there is a change to the encoding of base types such as integers and arrays. Program and version numbers specify the SFS read-only protocol and its revision level. A direction integer specifies that the message is a reply (currently the only kind of signed message in SFS). An array of procedure numbers enumerates the procedures to which the message constitutes a valid reply; some messages can be given in response to multiple procedure calls—for instance a stale file handle message might be returned for both a read and get attributes request on a particular file. Finally, a start time and duration indicate the period of time during which the message should be considered valid.

Figure 4-6 lists the remote procedure calls in the read-only protocol. Unlike the read-write protocol, all replies in the SFS read-only protocol contain enough context to prevent them from being interpreted as replies to different questions. Get attributes replies, for instance, contain both a file's handle and its attributes. Failed name lookups return a directory file handle and two names that no files lie between alphabetically.

Replies to file reads are not signed. Clients must verify file contents separately. So as to obtain atomicity and non-repudiation, a single signature proves the entire

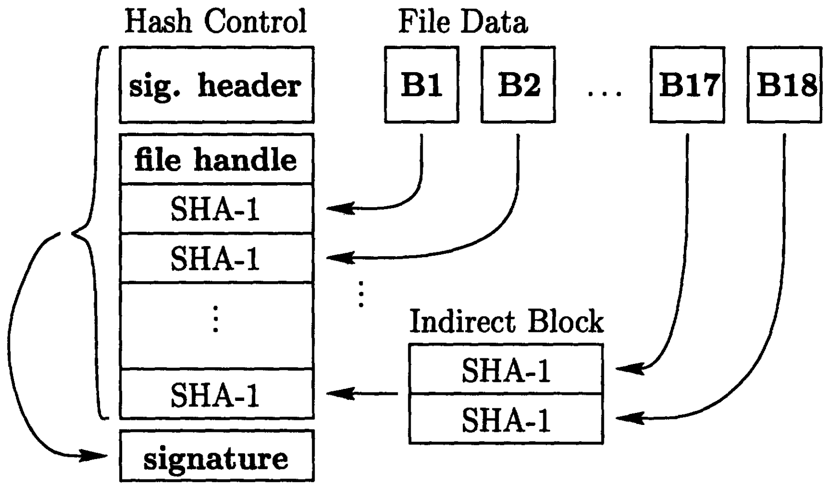

Figure 4-3: A hash control block

contents of a file. Simply signing the file's contents with its handle, however, would require clients to download an entire file before verifying any block. Instead, read-only servers sign a hash control block for each file. A hash control block, shown in Figure 4-3, consists of a file handle, cryptographic hashes of up to 16 8K data blocks at the start of the file, and, for larger files, the hashes of single, double, and tripple indirect hash blocks. A remote *readhash* procedure allows clients to retrieve these indirect blocks, each of which can contain up to 400 hashes of file data blocks, or of single or double indirect hash blocks. Thus, verifying the first block of a file requires only a small hash control message. A random read from a large file will additionally require three indirect blocks to be fetched in the worst case, but if multiple reads exhibit any spatial locality, the appropriate indirect blocks will very likely be cached at the client.

Note that while key exchange in the read-write protocol guarantees the integrity and secrecy of messages in both directions, the read-only protocol only guarantees integrity of messages from the server; it provides no secrecy whatsoever. Therefore, the read-only protocol is only appropriate for fully public, read-only data. Any data alteration or non-public data must use another path.

Where full key exchange guarantees the freshness of each message, signed messages remain valid for the entire duration of a predetermined time interval. Thus, the read-only protocol cannot provide the tight consistency guarantees that key exchange offers. Frequently changing data should not be made accessible through the read-only protocol, as the cost to recompute constantly expiring, per-file signatures will likely exceed the per-connection cost of initial key exchange.

Avoiding key exchange allows an SFS server to accept many more clients for stable read-only data. In addition, it allows read-only file systems to be replicated on untrusted machines by replicating signatures and data but not the private key. Although the signature solution only applies to read-only accesses, we believe that the number of people writing a particular file system will not reach critical levels.

connect	Specifies the file system a client intends to mount. Returns a public key that hashes to Host ID.
encrypt	Begin encrypting the connection (must follow a **connect**). Returns the root file handle.
login	Begins authenticating a user. Takes a public key and returns a cryptographic challenge.
chalres	Takes a response to a **login** challenge. If the response is correct, returns an authentication number.
logout	Deallocates an authentication number.
statfs	Returns the amount of used and free disk space available in a particular file's partition.
readdir	Returns an 8 KB block of file names in a directory.
lookup	Takes a directory and a file name. Returns a file handle, attributes, and a lease.
readlink	Returns the contents of a symbolic link.
getattr	Gets the attributes of a file and obtains a lease
getlease	Obtains or refreshes a lease.
read	Read file data.
write	Read file data.
create	Creates a file and obtains a lease.
link	Links a file into a directory.
remove	Unlinks a file from a directory.
mkdir	Creates a directory.
rmdir	Deletes a directory.
symlink	Creates a symbolic link.
setattr	Sets the attributes of a file.
rename	Renames a file.

Figure 4-4: Calls in the SFS read-write protocol

killlease	Requests that a client relinquish a write lease prematurely, or tells a client that a read lease will no longer be honored.
dirmod	Notifies a client of a created or deleted link in a directory for which the client holds a lease.
rename	Notifies a client of a rename operation affecting a directory for which the client holds a lease.

Figure 4-5: Callbacks in the SFS read-write protocol

connect	Specifies the file system a client intends to mount. Returns a signed message containing the root file handle and a public key that hashes to Host ID. (A tagged union makes this compatible with the read-write connect call.)
lookup	Takes a directory and a file name. On success, returns a signed message containing a directory file handle, a name, a file handle, and an attribute structure. On failure, returns a signed message with a directory and two names between which no files lie alphabetically. For recently deleted directories, can also return a signed stale file handle error.
getattr	Returns a signed message with a file handle and attributes. For recently deleted files, may also return a signed stale file handle error.
readlink	Returns a signed message with the file handle of a symbolic link and its contents. Can return a signed stale file handle error.
read	Returns an unsigned block from a file. The data must be verified through a hash control block.
readdir	Returns an unsigned block of file names in a directory. The data must be verified through a hash control block.
hashctl	Returns a signed hash control block for a file handle.
readhash	Returns a single, double, or tripple indirect hash block. The block is unsigned and must be verified through a hash control block (and possibly one or more indirect blocks).

Figure 4-6: Calls in the SFS read-only protocol

Chapter 5

Reference Implementation

We have built a reference implementation of SFS for Unix systems. The goals for this implementation are to provide portability and ease of installation. We were willing to sacrifice performance to meet these goals, so long as the overall performance of the reference implementation stayed competitive with NFS—a fairly-low-performance file system which nonetheless enjoys widespread use.

There are several ways of implementing a new file system in Unix; one of the most attractive is to put the file system in the kernel at the vnode layer [29]. NFS, AFS, WebFS and local file systems like FFS are all implemented at the vnode layer. Unfortunately, while writing a file system at the vnode layer gives the highest performance and the most control, the vnode interface differs enough between operating systems that portability becomes a serious problem. Also, people are generally more reluctant to modify their kernels, even through dynamically linked modules, than to install user-level software. (We do eventually hope to implement SFS as the vnode layer in a free operating system, but can't expect SFS to succeed on such a platform-specific implementation.)

New file systems can also be implemented by replacing system shared libraries or even intercepting all of a process's system calls, as the UFO system does [2]. Both methods are appealing because they can be implemented by a completely unprivileged user. Unfortunately, it is hard to implement complete file system semantics using these methods (for instance, you can't hand off a file descriptor using sendmsg).

Both methods also fail in some cases—shared libraries don't work with statically linked applications, and the UFO file system can't work across executions of set-user-id programs. Moreover, having different namespaces for different processes can cause confusion, at least on operating systems that don't normally support this.

We therefore decided to implement SFS using NFS loopback mounts. (We chose NFS version 2, as it is the most widely supported.) A user-level process on each SFS client machine binds a UDP socket and passes the address of that socket to the kernel in an NFS mount system call.

Several previous file systems have used NFS loopback mounts [10, 17, 20]. These file systems all have new functionality which complements existing file systems and is valuable even at the cost of some performance. SFS, however, aims to replace other network file systems rather than complement them; it must therefore offer competitive performance as well as new functionality.

5.1 Structure

SFS consists of three principal programs: *agent*, *sfscd*, and *sfssd*. In addition, some utility programs allow users to do such things as create public key pairs, and several auxiliary daemons get invoked by and communicate with *sfscd* and *sfssd*. Figure 5-1 shows the overall structure of SFS, which the next few subsections will explain in detail.

5.1.1 Programs

The *sfscd* program, which runs on all client machines, acts as an NFS server to the local operating system. *sfscd* mounts itself under the directory `/sfs` and responds to NFS requests from the kernel. When users reference non-existent directories of the proper format under `/sfs`, *sfscd* attempts to connect to correspondingly named file servers. If successful, it hands the connection off to an auxiliary client deamon for the appropriate protocol dialect and creates a mount point for the new file system.

Each user on a client machine runs an instance of the *agent* program. *agent*

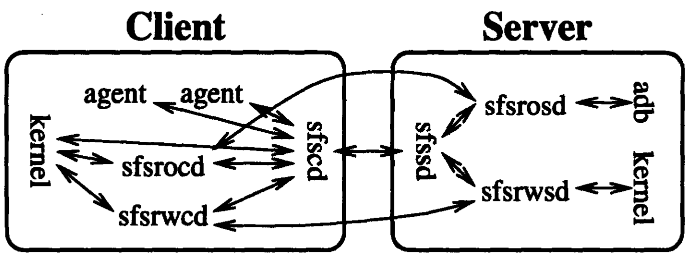

Figure 5-1: Structure of SFS.

contacts the *sfscd*[1] program using the SFS control protocol, *sfsctrl*, to declare its willingness to authenticate the particular user to remote file servers. This agent knows the user's private key, which can be stored locally or encrypted and universe-readable on a remote SFS file system. Each time the user access a remote SFS server for the first time, the agent automatically authenticates the user as described in Section 4.1.1. We expect that the agent will be started as part of the login process.

All server machines run either *sfsrwsd*, for read-write servers, or *sfsrosd* for read-only ones. Either one of these daemons can be run individually, or multiple instances of either or both can be started by an *sfssd* process. *sfsrwsd*, the read-write server, accesses exported local file systems through NFS. For performance, it needs to be able to keep the local disk queue full to achieve good disk-arm scheduling. It also needs to refer to files through a low-level representation (such as inode number) rather than through mutable pathnames, since pathnames might change between client references to the same file. NFS satisfies both these needs; by communicating to the local operating system through nonblocking sockets, *sfsrwsd* can issue parallel I/O requests in terms of low-level file handles. *sfsrosd*, the read-only server, serves information from a database rather than a local file system. It uses uses multiple processes to retrieve preformatted, digitally signed replies asynchronously from an on-disk B-tree.

[1]Currently *agent* actually communicates directly with the read-write daemon, *sfsrwcd*, because the read only auxiliary daemon does not need to authenticate users. This structural inelegance will be corrected in the near future.

5.1.2 Modular design

Network system software like SFS can easily get firmly entrenched and become very resistant to change. Moreover, concern for backwards compatibility can vastly complicate future versions of software, both hindering innovation and increasing the likelihood of security holes. We therefore chose a modular design for SFS to let multiple versions of the client and server exist on the same machine and share the same namespace.

On the server side, the main daemon, *sfssd*, is an extremely simple program (only 380 lines of C code). On startup, it scans a configuration file that maps file system names and protocol revision numbers to pathnames of an auxiliary daemon and arguments to pass those daemons. *sfssd* runs one instance of each auxiliary daemon listed in the configuration file. It then accepts all incoming network connections from SFS clients, determines which server should handle each connection, and hands connections off to the appropriate daemons through Unix's facilities for transferring file descriptors between processes. This organization allows multiple versions of an SFS server to run on the same machine using the same port number, and should consequently make upgrading a server relatively painless even when the new version of the software no longer supports an older protocol. True, under this multiple server scheme, two clients using different protocol versions may no longer enjoy the tight consistency guarantees offered individually by each version of the server. This seems like a small price to pay for vastly simplifying innovation, however.

The main client daemon, *sfscd*, is a bit more complicated than *sfssd*. It acts as an NFS server for the `/sfs` directory and creates new mount points for servers as they are referenced. *sfscd* begins by reading a configuration file that maps protocol dialects, RPC program numbers, and version numbers to the paths of auxiliary daemons and arguments to pass those daemons. It then spawns an instance of each auxiliarly daemon listed in the configuration file, giving each daemon a separate socket over which to receive NFS requests from the kernel. When someone references a non-existent server name in `/sfs`, *sfscd* contacts the server and decides based on the response to the initial connection request which auxiliary client daemon should handle the connection. It then transfers the connection to the appropriate daemon by passing it the file descriptor[2]. After receiving a connection, an auxiliary client daemon gives *sfscd* an NFS file handle for the root of the new file system, which *sfscd* then mounts at the appropriate place.

When a loopback NFS server crashes, it creates an unavailable NFS server, often leading to hung processes and requiring a machine to be rebooted. This can present a significant inconvenience to those developing loopback servers. However, *sfscd* keeps copies of the sockets it gives to auxiliary daemons for NFS. If an auxiliary daemon ever dies, *sfscd* takes over the socket and starts returning stale file handle errors to the kernel until it can cleanly unmount all of the crashed daemon's file systems.

The SFS client software's modular design allows old, new, and experimental versions of the SFS protocol to coexist on the same machine and share the same namespace. Furthermore, *sfscd* solves the greatest danger of experimental loopback servers, their tendency to lock up the machine if they crash. SFS should therefore continue to enjoy innovation even once people start relying on it.

5.2 Event-driven architecture

Because threads are nonportable on Unix, we used an event-driven architecture for both the client and server. Therefore, neither the client nor the server can ever make a blocking system call when it has any amount of work to do. In a few cases where *sfscd* needs to make system that might block (such as a mount or unmount call), it uses a separate process to make the call.

Supporting this event-driven architecture required building a new asynchronous RPC (ARPC) library. We chose to make this library compatible with Sun RPC [50] and the Sun XDR [51] marshaling routines generated by `rpcgen`. We also built

[2]On Linux, the one version of Unix that doesn't support file descriptor passing, the auxiliary client daemon can simply open a new connection to the server being mounted. While this wastes time and kernel protocol control blocks, we don't expect mounting of file systems to be on the critical path for most applications.

an encrypting packet stream transport over TCP, and added a new base type of multiprecision integer (to facilitate public key messages). This allowed the same RPC routines to be used for NFS, SFS, the authentication agent protocol, and all hash operations (which need to be performed on well-defined, endian-neutral data structures).

Though these rpcgen-generated XDR routines are less efficient than we would have hoped, using XDR has several benefits. First, it allows us to specify the exact SFS protocol with no ambiguity; as **rpcgen** is widely available, this specification can be easily and widely understood. Second, we could assure ourselves that the autogenerated marshaling code had no buffer overruns or other bugs that could be triggered by malformed remote data.

5.3 User-level NFS

Building a high performance multiuser file system over loopback NFS mountpoints posed several implementation challenges. First of all, many Unix kernels lock NFS mountpoints when a server fails to respond, so as to avoid flooding a server with retransmits. In order to prevent one slow server from blocking file system requests to another server we had to use a separate NFS loopback mountpoint for each remote file server. This vastly complicated the process of automatically mounting remote file systems, as a mount is only initiated once an NFS request on the **/sfs** directory is pending. Since we also wanted the **pwd** command to show full canonical pathnames, including a server's host ID, we could not simply use the solution employed by automounters of returning a symbolic link to an actual mount point in a different directory [16, 37]. Instead, *sfscd* uses a delaying tactic of sending the user through a chain of symbolic links which either return to the original mountpoint (now no longer a symbolic link), if the mount was successful, or end up with a "no such file or directory" error if the host does not exist or the public key does not match.

This solution is only possible because of the event-driven architecture of *sfscd*. While *sfscd* delays the user's request (after having redirected it, with a symbolic link, to a different mountpoint so as to avoid having the underlying operating system lock the `/sfs` directory), it continues to process other NFS requests. After an asynchronous hostname lookup, connection setup with the remote server, and public key verification, *sfscd* responds to the user's request, which has been delayed and redirected, by returning a symbolic link that points back into the `/sfs` directory where the remote file server has now been mounted.

Yet another challenge in implementing a user-level NFS server is that the underlying operating system hides information from it. In particular, an NFS server doesn't see file open, file close, and, worse yet, fsync (forced write through) operations. We deal with this problem in two ways. First, SFS does not perform write through on close operations; this was a conscious decision for performance based on the results of Ousterhout et al. [35], which show that most files stay open only for short periods of time, thereby undermining the usefulness of write-back caching. In the future, if our reference implementation can count on operating systems supporting NFS version 3, we could duplicate the NFS semantics on close, if desired.

Second, we implemented fsync by flushing a file before most attribute changes; this flush is required because an attribute change may revoke the permissions required to write back dirty blocks from the cache. Thus, a meaningless attribute change (e.g., changing the owner of a file to its current owner) can be used to guarantee that all dirty data has been written to disk. This solution is not ideal, since it requires modifications to shared libraries for applications to obtain the correct behavior from fsync. We are also considering performing an implicit fsync before all rename operations, since in practice the most critical uses of fsync seem to precede rename operations.

5.4 Client cache

To offset the cost of software encryption, *sfsrwcd* keeps a large on-disk client cache as described in Section 4.1.2. This on-disk cache can either be a regular file or, preferably, a reserved disk partition; the latter avoids any double buffering and does not disrupt efforts to cluster chunks of data. However, all experiments in this paper use a regular file for a cache.

To avoid blocking when accessing on-disk caches, SFS client daemons do so asynchronously through helper processes.

5.5 Implementation details

Maintaining consistency with write-back caches in the face of network failures is hard. An SFS client attempts to flush any outstanding writes in its cache after a network or server failure by periodically trying to reestablish a connection with the server. Once connection is reestablished, these writes will succeed as long as their users' agents are still running. If the blocks are written back, they may overwrite subsequent changes made by other users. If a user's agent no longer exists, however, pending writes will not be performed, as the user who performed them can no longer be authenticated. Note that if, after killing her *agent*, a user goes on to change the same file from another client machine, this may be the right behavior: the user's later changes will not be overwritten by her earlier ones. Using techniques developed for disconnected operation [28], one could do better; we may in the future adopt such techniques.

sfsrwcd needs to translate remote file permissions to something that makes sense on the local machine. This requires that it provide a per-user view of the file system. Consider, for instance, the case where two users in the same group on the local machine are in different groups on the remote machine. For files accessible to one of the remote groups, *sfsrwcd* must provide different group id mappings depending on which user performs an access. To accomplish this, SFS disables the kernel's NFS attribute cache for all SFS file systems, allowing it to give different local users different views of the file system. SFS builds ID translation tables similar to those of RFS [41]. However, in SFS these tables are maintained dynamically and automatically, and are specific to each user.

NFS version 2, has fixed-size, 32-byte file handles. *sfsrwsd* must be careful with these handles, because an attacker can gain complete access to a file system given only the NFS file handle of one directory. Moreover, SFS file handles are only 24 bytes, as client daemons need to tag them with server IDs before turning them into NFS file handles. *sfsrwsd* therefore compresses and encrypts NFS file handles to generate SFS ones. Because most operating systems use a stylized form for their supposedly opaque NFS file handles, *sfsrwsd* can reduce 24 of the 32 bytes of a file handle to a 4-byte index into a table, allowing squeezed NFS file handles to occupy only 12 of the 24 bytes in an SFS file handle. The remaining bytes are used for redundancy; they are set to zero before the file handle is encrypted with blowfish [47] in CFB mode.

5.6 Security issues

The SFS client daemons exchange NFS packets with the kernel over the loopback network interface. They do so by listening for UDP packets at a port with the localhost IP address. However, some operating systems will accept forged packets with localhost source and destination addressees if those packets arrive over a non-loopback interface. While routers cannot route such packets, attackers with direct access to a client's physical network can transmit them. SFS client daemons blowfish-encrypt their NFS file handles, making them hard to guess; this prevents attackers from modifying SFS file systems with forged loopback NFS requests. However, it may still be possible for attackers with network access to guess 32-bit RPC packet IDs and forge responses from SFS client daemons to the kernel. Operating systems that accept forged localhost packets should be fixed. Until they are, several freely available IP filter packages can be used to filter such forgeries. Nevertheless, people need to remain aware of this potential vulnerability.

In general, while SFS keeps file data private and prevents unauthorized modification in the face of network attacks, it does not solve all security problems. Obviously, SFS is only as good as the security of the local operating system. Someone who breaches the security of a client machine and becomes superuser can read SFS files accessible to any users on that machine. Moreover, the superuser can even attach to users' authentication agents with a debugger to read their private keys.

SFS servers must also be kept secure. While SFS encrypts file system data sent over the network, it does so with freshly negotiated session keys. Thus, servers need access to file data in unencrypted form. Though not strictly required by the protocol, the current server implementation also stores exported files in cleartext on disk, through the operating system's local file system. SFS allows file sharing across the network without sacrificing the security of servers or their local file systems. It does not provide higher security than a local file system, or otherwise replace encrypting file systems such as CFS [10] which encrypt all data written to disk.

Like most network services, SFS is vulnerable to denial of service attacks, such as SYN-bombing or large numbers of connections from the same client. More importantly, the very existence of a global file system raises new kinds of security concerns. For example, SFS makes it significantly easier for Trojan horses to copy all a user's private files to a remote location. Even with the level of security that SFS transparently provides, therefore, users must still be conscious of some security issues.

Chapter 6

Results

We will first discuss SFS design choices that affect performance, and then discuss the measured performance of our SFS reference implementation.

The results presented here are extremely promising, but SFS is not yet release quality. In particular, we intend to rewrite the client cache code entirely. When writing into a dirty chunk, the cache's current structure first requires the entire 64 KB chunk to be read into core if it is not there already. Fixing this will definitely improve performance. More seriously, however, the current cache can end up storing more dirty blocks than there is room for on the server. Correcting this will require limiting the amount of unallocated write behind when servers have little free disk space, and may consequently hurt performance.

6.1 Designing SFS for high performance

There are several factors which might differentiate SFS performance from NFS performance. First, SFS is more CPU intensive than NFS, because of the encryption and integrity checks performed on file data and metadata. However, SFS has two important features which allow a high performance implementation—a large data and meta-data cache, and an asynchronous RPC library.

6.1.1 SFS data and meta-data cache

SFS's large cache allows most recently-used data to be accessed locally. This saves both the communication cost with the server and the cost of decrypting and verifying data. However, retrieving file attributes requires the kernel to schedule the *sfscd* process. This is potentially expensive, and shows up as a small cost in some benchmarks.

Additionally, the cache allows SFS to buffer a large number of writes, much larger than the typical NFS implementation where the number of outstanding write requests is limited by the number of NFS I/O daemon processes.

6.1.2 Asynchronous RPC

By allowing multiple outstanding requests though ARPC, we can overlap their latencies, or use the time they are outstanding to do other work (like encryption—as measured in 6.2). SFS's aggressive write buffering can accumulate many file chunks that need to be written back to the server, and these write backs can proceed in parallel with each other and with additional requests. Read-ahead requests (which are not yet implemented) can also proceed in parallel. Renames, and most metadata changes are synchronously forwarded to the server and benefit only from the ARPC library's efficiency.

6.2 Measured performance of SFS

We ran several sets of experiments to validate the design of SFS. Most importantly, we wanted to ensure that the end-to-end performance of standard file system workloads, as exemplified by the modified Andrew Benchmark, was competitive with standard NFS. We also wanted to determine SFS's performance characteristics under specific types of loads. For this we used the Sprite LFS benchmarks that stress small file creation and deletion, and large file reads and writes. Finally, we look at the importance and interaction of two core implementation decisions—asynchronous RPC and data encryption.

Note that the mesurements in this section represent a version of SFS with a slightly older automounter. This should not matter as the cost of mounting file systems is not reflected in any of the benchmarks. Measurements of the newest version of SFS on a slightly different network configuration yielded results almost identical to those presented here.

6.2.1 Experimental setup

Client and servers are 200 MHz Pentium Pros running OpenBSD 2.0, with 64MB of memory, 4GB of disk, and 10Mb SMC Ultra Ethernet cards. The client and server were run on different, unloaded machines in multiuser mode. We feel this configuration mirrors what might be found in a modern lab or office. While no special effort was made to reduce ambient network traffic, all experiments were run until at least three runs generated very similar results. The network was periodically monitored to ensure that the experiments do not represent performance artifacts.

6.2.2 Modified Andrew benchmark

We ran the modified Andrew benchmark [36] on a local file system, NFS, and SFS with encryption. The timing results are shown in Figure 6-1. All experiments were run with a warm cache. SFS's data and meta-data cache and aggressive write-buffering make it faster than NFS on the MAB.

The first phase of the MAB just creates a few directories. The second stresses large-volume data movement as a number of small files are copied. The third phase collects the file attributes for a large set of files. The fourth phase searches the files for a string which does not appear and the final phase launches a compile.

Figure 6-1 also shows that the high security guarantees of SFS do not cost significant performance for this type of file system usage. The SFS caches are successful in allowing it to avoid frequent decryption. These numbers confirm our experience in using SFS for document preparation and for compiling (often the sources to SFS

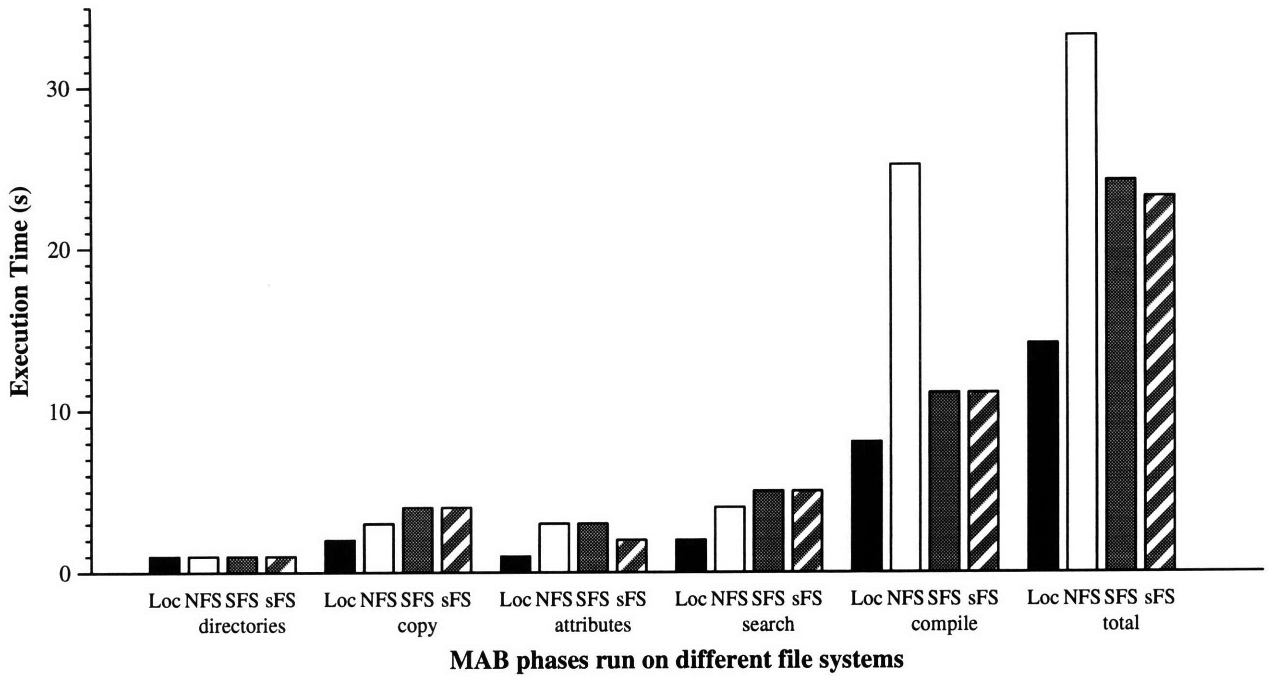

Figure 6-1: Wall clock execution time (in seconds) for the different phases of the modified Andrew benchmark, run on different file systems. sFS is SFS without data encryption, and Loc is OpenBSD's local FFS file system.

itself).

6.2.3 Sprite LFS microbenchmarks

The Sprite LFS microbenchmarks [43] do not represent any realistic file system workload, but they help isolate performance characteristics of the file system. The small file test creates 1,000 1KB files, reads them, and unlinks them all. The large file test writes a large file sequentially, reads from it sequentially, then writes it randomly, reads it randomly, then reads it sequentially. Data is synced to disk after each write.

SFS and NFS have comparable performance for create and unlink phase of the small file benchmark, as both need to synchronously contact the server. The read performance for small files is poor over SFS due to an inefficiency in the prototype implementation. The current SFS implementation manages its cache in 64KB chunks, regardless of the size of the file. When the chunk size is set to 8K, SFS create time increases to 29 seconds (equaling NFS), but the read time comes down from 25 seconds

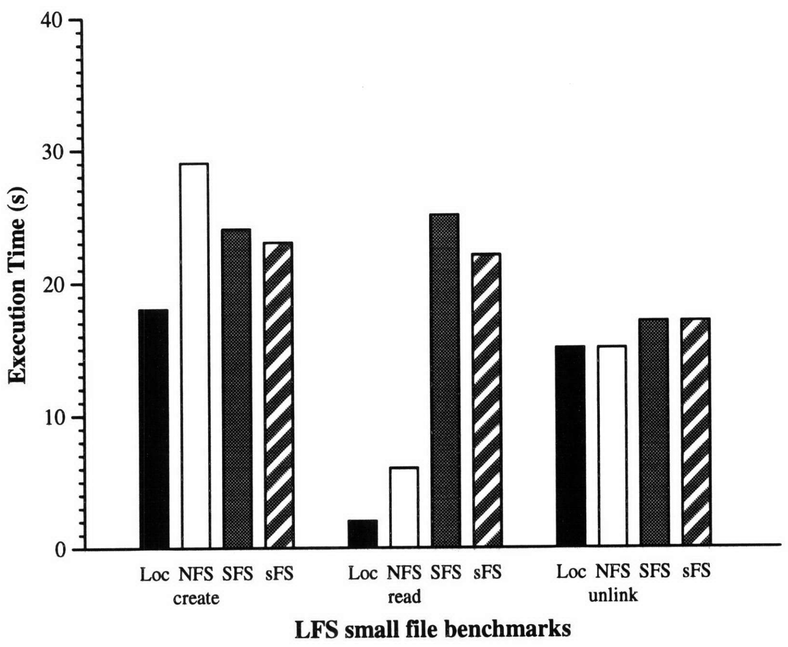

Figure 6-2: Wall clock execution time for the different phases of the Sprite LFS small file benchmark, run over different file systems. The benchmark creates, reads, and unlinks 1,000 1KB files. sFS is SFS without data encryption, and Loc is OpenBSD's local FFS file system.

to 4 seconds (faster than NFS), representing the true overhead of SFS relative to the local file system (due to the scheduling overhead for *sfscd*). It will not take much work to make SFS's cache chunks size sensitive to file and read request size.

For the large file benchmark (shown in Figure 6-3) SFS performs well on operations, like the large sequential write, where it can buffer the data, sending the writes to the server in parallel. It also performs well on operations that use its cache, like the large sequential reads.

The random write following the sequential write and read is tough for SFS because its cache has dirty data in it that must be read from disk and flushed to the server before space is created for the newly written data.

6.2.4 Asynchronous RPC microbenchmarks

The MAB and the LFS benchmarks seem to indicate that the encryption and integrity costs of SFS are small, and that the efficiency of and ability to overlap RPCs is an

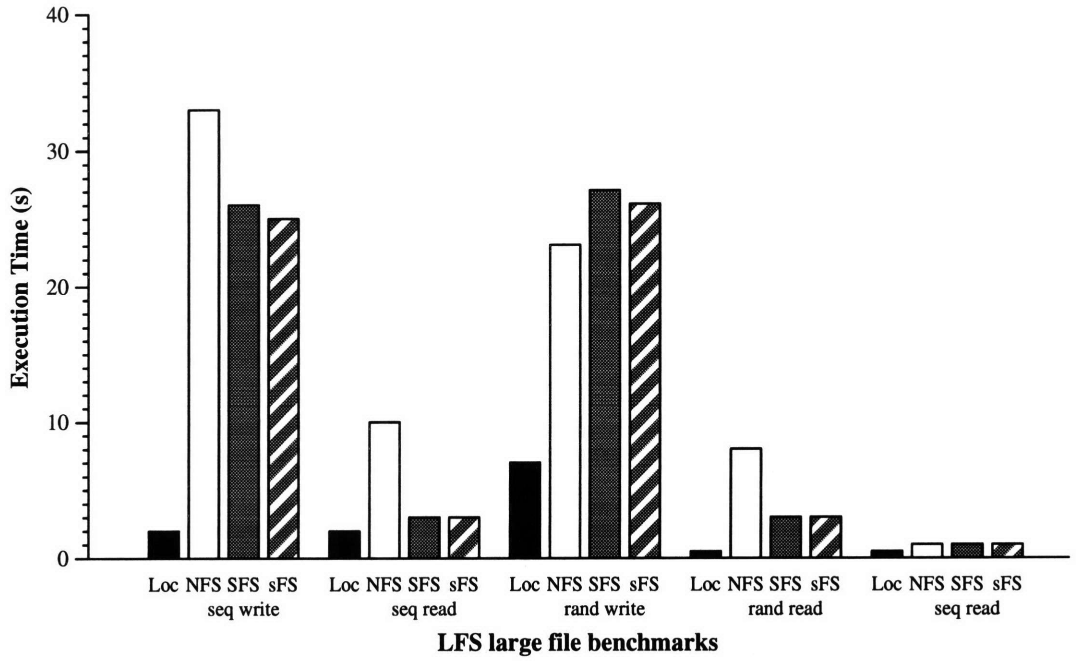

Figure 6-3: Wall clock execution time for the different phases of the Sprite LFS large file benchmarks, run over different file systems. The benchmark creates an 8MB file and reads and writes 8k chunks. sFS is SFS without data encryption, and Loc is OpenBSD's local FFS file system.

RPC Implementation	100B/1	100B/2	8KB/1	8KB/2
SUN RPC	0.22	—	20.0	—
ARPC	0.23	0.13 (1.7x)	1.7 (12x)	1.1 (18x)
XARPC	0.25	0.15 (1.5x)	2.0 (10x)	1.1 (18x)

Figure 6-4: The time (in milliseconds) taken for various RPC libraries to complete a call. Numbers in parenthesis are speed-up factors relative to SUN RPC. The first number in the column headings indicate the argument size (100 bytes or 8KB). The response is held fixed at 100 bytes. The second number indicates the number of outstanding requests (only the asynchronous RPC implementations can have multiple outstanding requests). Times are the average of 4000 calls for 100 byte argument and 700 calls for 8KB argument

important part of SFS's performance. We investigated this hypothesis, along with measuring the relative performance of SFS's asynchronous RPC library (ARPC), asynchronous RPC library with encryption (XARPC), and SUN RPC.

We measured RPC times for a 100 byte argument, as in the case of a set attribute, and with an 8KB argument, as in the case for a write. Responses are always 100 bytes, corresponding to a lease. The results are in Figure 6-4. All data is opaque, and so not marshalled by the XDR routines.

When only one request is outstanding, the base ARPC library is as efficient as SUN RPC for small (100 byte) arguments. Encryption adds only a 12% overhead. For large arguments (8KB), the base ARPC library is 12x faster than SUN RPC, due to heavy optimization for opaque data which avoids copies and mallocs. Encryption brings the performance down 17%, but that is still 10 times faster than SUN RPC.

In addition to the efficiency of the ARPC library, Figure 6-4 also shows how overlapping RPCs is a performance advantage for SFS. Small and large arguments, whether or not they use encryption, all show a 1.5–1.8x speed up from being able to overlap two RPCs.

Chapter 7

Future work

The primary goal of our continuing work on SFS is a stable release of the software, including free source code. SFS is designed to be used, and its benefits will not become fully clear until it is in use at many sites. Our research group will be SFS's first beta testers; we hope to eliminate NFS entirely from our networks and use SFS exclusively. This work will also include porting SFS to other Unix platforms, which, again, should be quite simple.

We may revise the authentication protocols. Currently, read-write servers must perform two private key operations at connection setup—one for authentication and one for forward secrecy. The latter could just as well be performed by the client, saving the server CPU cycles. Pushing this private key operation to clients could also allow access control in read-only servers (which for performance reasons do not perform private key operations); the user authentication protocol would guarantee the secrecy of access-controlled data through its use of the session ID.

Currently, all access control in SFS relies on user and group IDs, which are translated from one machine to another; users must have accounts on file servers to access any protected files. It would be useful to allow users to grant file access to the holders of particular public keys without having to get an administrator to create a local account for each such person, as was permitted by Truffles [40]. It would also be desirable to allow externally designated groups of users, so that, for instance, files can be exported to all employees of a company without the file server having to keep an explicit list of employee public keys. Utilities to list and manipulate remote files with their remote user- and group-names would also prove quite useful.

We may explore communication between clients. Insecure file systems such as xFS [3] have exploited client-to-client data transfers to improve the performance of shared files. Between mutually distrustful clients, this kind of cooperation opens up many interesting questions. Given that all SFS clients trust a server to control the contents of a particular file system, however, such client communication can be established without incurring any additional private key operations.

Finally, disconnected operation can be seriously improved. In fact, SFS may enable several new twists on previous schemes. For example, since all SFS users have private keys, clients can actually digitally sign file modifications after a network partition and write them back once a user has logged out.

Chapter 8

Conclusion

SFS demonstrates that a secure global file system with a uniform namespace does not need any centralized authority to manage that namespace. A portable, easy-to-install reference implementation shows that user-level NFS servers can offer reasonable performance and that the cost of software encryption can be tolerated through asynchronous RPC, large on-disk client caches, and encryption algorithms efficient in software. We hope to see many people use SFS.

Bibliography

[1] Martín Abadi. Explicit communication revisited: Two new examples. *IEEE Transactions on Software Engineering*, SE-23(3):185–186, March 1997.

[2] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proceedings of the 1997 USENIX*, pages 77–90. USENIX, January 1997.

[3] 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.

[4] Eshwar Belani, Alex Thornton, and Min Zhou. Authentication and security in webfs. from `http://now.cs.berkeley.edu/WebOS/security.ps`, January 1997.

[5] Mihir Bellare. The exact security of digital signatures—how to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology—Eurocrypt 1996*, volume 1070 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, 1995. ACM.

[7] A. D. Birrell and R. M. Needham. A universal file server. *IEEE Transactions on Software Engineering*, SE-6(5):450–453, September 1980.

[8] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Systems Research Center, September 1993.

[9] Andrew D. Birrell, Butler W. Lampson, Roger M. Needham, and Michael D. Schroeder. A global authentication service without global trust. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 223–230, 1986.

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

[11] Matthew Blaze and Rafael Alonso. Dynamic hierarchical caching in large-scale distributed file systems. In *Proceedings of the 12th International Distributed Computing Systems Conference*, pages 521–528, June 1992.

[12] Mark R. Brown, Karen N. Kolling, and Edward A. Taft. The Alpine file system. *ACM Transactions on Computer Systems*, 3(4):261–293, November 1985.

[13] Luis Felipe Cabrera and Jim Wyllie. Quicksilver distributed file services: An architecture for horizontal growth. In *Proceedings of the 2nd IEEE Conference on Workstations*, 1988.

[14] B. Callaghan. WebNFS client specification. RFC 2054, Sun Microsystems, Inc., October 1996.

[15] B. Callaghan. WebNFS server specification. RFC 2055, Sun Microsystems, Inc., October 1996.

[16] Brent Callaghan and Tom Lyon. The automounter. In *Proceedings of the Winter 1989 USENIX*, pages 43–51. USENIX, 1989.

[17] Vincent Cate. Alex—a global filesystem. In *Proceedings of the USENIX File System Workshop*, May 1992.

[18] Jeremy Dion. The Cambridge file server. *ACM SIGOPS Operating System Review*, 14(4):26–35, Oct 1980.

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

[20] David K. Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, Pacific Grove, CA, October 1991. ACM.

[21] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles.* ACM, 1989.

[22] Björn Grönvall, Ian Marsh, and Stephen Pink. A multicast-based distributed file system for the internet. In *Proceedings of the 7th ACM SIGOPS European Workshop*, pages 95–102, 1996.

[23] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

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

[25] Phil Karn and William Allen Simpson. The Photuris session key management protocol. Internet draft (draft-simpson-photuris-15), Network Working Group, July 1997. Work in progress.

[26] Kalle Kaukonen and Rodney Thayer. A stream cipher encryption algorithm "arcfour". Internet draft (draft-kaukonen-cipher-arcfour-01), Network Working Group, July 1997. Work in progress.

[27] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer 1990 USENIX*, pages 151–163. USENIX, 1990.

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

[29] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX*, pages 238–247. USENIX, 1986.

[30] Brian A. LaMacchia and Andrew M. Odlyzko. Computation of discrete logarithms in prime fields. In *Designs, Codes and Cryptography 1*, pages 47–62, 1991.

[31] Butler Lampson, Martín Abadi, Michael Burrows, and Edward P. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[32] Timothy Mann, Andrew D. Birrell, Andy Hisgen, Chuck Jerian, and Garret Swart. A coherent distrubuted file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, May 1994.

[33] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems, or your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX*, pages 305–312. USENIX, January 1992.

[34] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[35] J. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, WA, December 1985. ACM.

[36] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Summer USENIX '90*, pages 247–256, Anaheim, CA, June 1990.

[37] Jan-Simon Pendry. *Amd – an Automounter.* London, SW7 2BZ, UK. Manual comes with amd software distribution.

[38] Michael O. Rabin. Digitalized signatures and public key functions as intractable as factorization. Technical Report TR-212, MIT Laboratory for Computer Science, January 1979.

[39] 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.* North-Holland Publ., Amsterdam, 1981.

[40] Peter Reiher, Jr. Thomas Page, Gerald Popek, Jeff Cook, and Stephen Crocker. Truffles — a secure service for widespread file sharing. In *Proceedings of the PSRG Workshop on Network and Distributed System Security*, 1993.

[41] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, and Kang Yueh. RFS architectural overview. In *Proceedings of the Summer 1986 USENIX*, pages 248–259. USENIX, 1986.

[42] Ronald L. Rivest and Butler Lampson. SDSI—a simple distributed security infrastructure. Version 1.0 of working document from `http://theory.lcs.mit.edu/~rivest/publications.html`.

[43] 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.

[44] 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. USENIX, 1985.

[45] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[46] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[47] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.

[48] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25–34, Orcas Island, WA, December 1985. ACM.

[49] Alan B. Sheltzer and Gerald J. Popek. Internet Locus: Extending transparency to an Internet environment. *IEEE Transactions on Software Engineering*, SE-12(11):1067–1075, November 1986.

[50] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, August 1995.

[51] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.

[52] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*. USENIX, 1988.

[53] H. Sturgis, J. Mitchell, and J. Israel. Issues in the design and use of a distributed file system. *ACM Operating Systems Review*, 14(3):55–69, July 1980.

[54] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Working Group, March 1989.

[55] Bradley Taylor and David Goldberg. Secure networking in the Sun environment. In *Proceedings of the Summer 1986 USENIX*, pages 28–37. USENIX, 1986.

[56] Amin M. Vahdat, Paul C. Eastha, and Thomas E. Anderson. WebFS: A global cache coherent file system. from `http://www.cs.berkeley.edu/~vahdat/webfs/webfs.html`, December 1996.

[57] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70. ACM, 1983.

[58] Hugh. C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.

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