A toolkit for user-level file systems

David Mazières
Department of Computer Science, NYU
dm@cs.nyu.edu

Abstract

This paper describes a C++ toolkit for easily extending the Unix file system. The toolkit exposes the NFS interface, allowing new file systems to be implemented portably at user level. A number of programs have implemented portable, user-level file systems. However, they have been plagued by low-performance, deadlock, restrictions on file system structure, and the need to reboot after software errors. The toolkit makes it easy to avoid the vast majority of these problems. Moreover, the toolkit also supports user-level access to existing file systems through the NFS interface—a heretofore rarely employed technique. NFS gives software an asynchronous, low-level interface to the file system that can greatly benefit the performance, security, and scalability of certain applications. The toolkit uses a new asynchronous I/O library that makes it tractable to build large, event-driven programs that never block.

1 Introduction

Many applications could reap a number of benefits from a richer, more portable file system interface than that of Unix. This paper describes a toolkit for portably extending the Unix file system—both facilitating the creation of new file systems and granting access to existing ones through a more powerful interface. The toolkit exploits both the client and server sides of the ubiquitous Sun Network File System [15]. It lets the file system developer build a new file system by emulating an NFS server. It also lets application writers replace file system calls with networking calls, permitting lower-level manipulation of files and working around such limitations as the maximum number of open files and the synchrony of many operations.

We used the toolkit to build the SFS distributed file system [13], and thus refer to it as the SFS file system development toolkit. SFS is relied upon for daily use by several people, and thus shows by example that one can build production-quality NFS loopback servers. In addition, other users have picked up the toolkit and built functioning Unix file systems in a matter of a week. We have even used the toolkit for class projects, allowing students to build real, functioning Unix file systems.

Developing new Unix file systems has long been a difficult task. The internal kernel API for file systems varies significantly between versions of the operating system, making portability nearly impossible. The locking discipline on file system data structures is hair-raising for the non-expert. Moreover, developing in-kernel file systems has all the complications of writing kernel code. Bugs can trigger a lengthy crash and reboot cycle, while kernel debugging facilities are generally less powerful than those for ordinary user code.

At the same time, many applications could benefit from an interface to existing file systems other than POSIX. For example, non-blocking network I/O permits highly efficient software in many situations, but any synchronous disk I/O blocks such software, reducing its throughput. Some operating systems offer asynchronous file I/O through the POSIX *aio* routines, but *aio* is only for reading and writing files—it doesn't allow files to be opened and created asynchronously, or directories to be read.

Another shortcoming of the Unix file system interface is that it foments a class of security holes known as time of check to time of use, or TOCTTOU, bugs [2]. Many conceptually simple tasks are actually quite difficult to implement correctly in privileged software—for instance, removing a file without traversing a symbolic link, or opening a file on condition that it be accessible to a less privileged user. As a result, programmers often leave race conditions that attackers can exploit to gain greater privilege.

The next section summarizes related work. Section 3 describes the issues involved in building an NFS loopback server. Section 4 explains how the SFS toolkit facilitates the construction of loopback servers. Section 5 discusses loopback clients. Section 6 describes applications of the toolkit and discusses performance. Finally, Section 7 concludes.

2 Related work

A number of file system projects have been implemented as NFS loopback servers. Perhaps the first example is the Sun *automount* daemon [5]—a daemon that mounts remote NFS file systems on-demand when their pathnames are referenced. Neither *automount* nor a later, more advanced automounter, *amd* [14], were able to mount file systems in place to turn a pathname referenced by a user into a mount point on-the-fly. Instead, they took the approach of creating mount points outside of the directory served by the loopback server, and redirecting file accesses using symbolic links. Thus, for example, *amd* might be a loopback server for directory `/home`. When it sees an access to the path `/home/am2`, it will mount the corresponding file system somewhere else, say on `/a/amsterdam/u2`, then produce a symbolic link, `/home/am2` → `/a/amsterdam/u2`. This symbolic link scheme complicates life for users. For this and other reasons, Solaris and Linux pushed part of the automounter back into the kernel. The SFS toolkit shows they needn't have done so for mounting in place, one can in fact implement a proper automounter as a loopback server.

Another problem with previous loopback automounters is that one unavailable server can impede access to other, functioning servers. In the example from the previous paragraph, suppose the user accesses `/home/am2` but the corresponding server is unavailable. It may take *amd* tens of seconds to realize the server is unavailable. During this time, *amd* delays responding to an NFS request for file `am2` in `/home`. While the the lookup is pending, the kernel's NFS client will lock the `/home` directory, preventing access to all other names in the directory as well.

Loopback servers have been used for purposes other than automounting. CFS [3] is a cryptographic file system implemented as an NFS loopback server. Unfortunately, CFS suffers from deadlock. It predicates the completion of loopback NFS write calls on writes through the file system interface, which, as discussed later, leads to deadlock. The Alex ftp file system [7] is implemented using NFS. However Alex is read-only, which avoids any deadlock problems. Numerous other file systems are constructed as NFS loopback servers, including the semantic file system [9] and the Byzantine fault-tolerant file system [6]. The SFS toolkit makes it considerably easier to build such loopback servers than before. It also helps avoid many of the problems previous loopback servers have had. Finally, it supports NFS loopback clients, which have advantages discussed later on.

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 [1]. 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 neither approach works with setuid utilities such as *lpr*. Moreover, having different namespaces for different processes can cause confusion, at least on operating systems that don't normally support this.

FiST [19] is a language for generating stackable file systems, in the spirit of Ficus [11]. FiST can output code for three operating systems—Solaris, Linux, and FreeBSD—giving the user some amount of portability. FiST outputs kernel code, giving it the advantages and disadvantages of being in the operating system. FiST's biggest contributions are really the programming language and the stackability, which allow simple and elegant code to do powerful things. That is somewhat orthogonal to the SFS toolkit's goals of allowing file systems at user level (though FiST is somewhat tied to the VFS layer—it couldn't unfortunately be ported to the SFS toolkit very easily). Aside from its elegant language, the big trade-off between FiST and the SFS toolkit is performance vs. portability and ease of debugging. Loopback servers will run on virtually any operating system, while FiST file systems will likely offer better performance.

Finally, several kernel device drivers allow user-level programs to implement file systems using an interface other than NFS. The now defunct UserFS [8] exports an interface similar to the kernel's VFS layer to user-level programs. UserFS was very general, but only ran on Linux. Arla [17], an AFS client implementation, contains a device, xfs, that lets user-level programs implement a file system by sending messages through `/dev/xfs0`. Arla's protocol is well-suited to network file systems that perform whole file caching, but not as general-purpose as UserFS. Arla runs on six operating systems, making xfs-based file systems portable. However, users must first install xfs. Similarly, the Coda file system [12] uses a device driver `/dev/cfs0`.

3 NFS loopback server issues

NFS loopback servers allow one to implement a new file system portably, at user-level, through the NFS protocol rather than some operating-system-specific kernel-internal API (e.g., the VFS layer). Figure 1 shows the architecture of an NFS loopback server. An application accesses files using system calls. The operating system's NFS client implements the calls by sending NFS requests

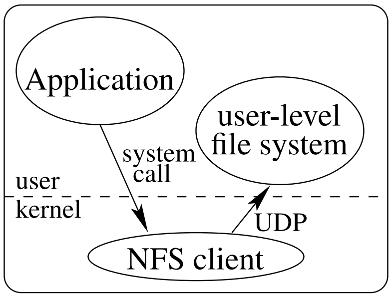

Figure 1: A user-level NFS loopback server

to the user-level server. The server, though treated by the kernel's NFS code as if it were on a separate machine, actually runs on the same machine as the applications. It responds to NFS requests and implements a file system using only standard, portable networking calls.

3.1 Complications of NFS loopback servers

Making an NFS loopback server perform well poses a few challenges. First, because it operates at user-level, a loopback server inevitably imposes additional context switches on applications. There is no direct remedy for the situation. Instead, the loopback file system implementer must compensate by designing the rest of the system for high performance.

Fortunately for loopback servers, people are willing to use file systems that do not perform optimally (NFS itself being one example). Thus, a file system offering new functionality can be useful as long as its performance is not unacceptably slow. Moreover, loopback servers can exploit ideas from the file system literature. SFS, for instance, manages to maintain performance competitive with NFS by using leases [10] for more aggressive attribute and permission caching. An in-kernel implementation could have delivered far better performance, but the current SFS is a useful system because of its enhanced security.

Another performance challenge is that loopback servers must handle multiple requests in parallel. Otherwise, if, for instance, a server waits for a request of its own over the network or waits for a disk read, multiple requests will not overlap their latencies and the overall throughput of the system will suffer.

Worse yet, any blocking operation performed by an NFS loopback server has the potential for deadlock. This is because of typical kernel buffer allocation strategy. On many BSD-derived Unixes, when the kernel runs out of buffers, the buffer allocation function can pick some dirty buffer to recycle and block until that particular buffer has been cleaned. If cleaning that buffer requires calling into the loopback server and the loopback server is waiting for the blocked kernel thread, then deadlock will ensue.

To avoid deadlock, an NFS loopback server must never block under any circumstances. Any file I/O within a loopback server is obviously strictly prohibited. However, the server must avoid page faults, too. Even on operating systems that rigidly partition file cache and program memory, a page fault needs a `struct buf` to pass to the disk driver. Allocating the structure may in turn require that some file buffer be cleaned. In the end, a mere debugging *printf* can deadlock a system; it may fill the queue of a pseudo-terminal handled by a remote login daemon that has suffered a page fault (an occurrence observed by the author). A large piece of software that never blocks requires fundamentally different abstractions from most other software. Simply using an in-kernel threads package to handle concurrent NFS requests at user level isn't good enough, as the thread that blocks may be the one cleaning the buffer everyone is waiting for.

NFS loopback servers are further complicated by the kernel NFS client's internal locking. When an NFS request takes too long to complete, the client retransmits it. After some number of retransmissions, the client concludes that the server or network has gone down. To avoid flooding the server with retransmissions, the client locks the mount point, blocking any further requests, and periodically retransmitting only the original, slow request. This means that a single "slow" file on an NFS loopback server can block access to other files from the same server.

Another issue faced by loopback servers is that a lot of software (e.g., Unix implementations of the ANSI C *getcwd()* function) requires every file on a system to have a unique (`st_dev`, `st_ino`) pair. `st_dev` and `st_ino` are fields returned by the POSIX *stat()* function. Historically, `st_dev` was a number designating a device or disk partition, while `st_ino` corresponded to a file within that disk partition. Even though the NFS protocol has a field equivalent to `st_dev`, that field is ignored by Unix NFS clients. Instead, all files under a given NFS mount point are assigned a single `st_dev` value, made up by the kernel. Thus, when stitching together files from various sources, a loopback server must ensure that all `st_ino` fields are unique for a given mount point.

A loopback server can avoid some of the problems of slow files and `st_ino` uniqueness by using multiple mount points—effectively emulating several NFS servers. One often would like to create these mount points on-the-fly—for instance to "automount" remote servers as the user references them. Doing so is nontrivial because of vnode locking on file name lookups.

While the NFS client is looking up a file name, one cannot in parallel access the same name to create a new mount point. This drove previous NFS loopback automounters to create mount points outside of the loopback file system and serve only symbolic links through the loopback mount.

As user-level software, NFS loopback servers are easier to debug than kernel software. However, a buggy loopback server can still hang a machine and require a reboot. When a loopback server crashes, any reference to the loopback file system will block. Hung processes pile up, keeping the file system in use and on many operating systems preventing unmounting. Even the *unmount* command itself sometimes does things that require an NFS RPC, making it impossible to clean up the mess without a reboot. If a loopback file system uses multiple mount points, the situation is even worse, as there is no way to traverse higher level directories to unmount the lower-level mount points.

In summary, while NFS loopback servers offer a promising approach to portable file system development, a number of obstacles must be overcome to build them successfully. The goal of the SFS file system development toolkit is to tackle these problems and make it easy for people to develop new file systems.

4 NFS loopback server toolkit

This section describes how the SFS toolkit supports building robust user-level loopback servers. The toolkit has several components, illustrated in Figure 2. *nfsmounter* is a daemon that creates and deletes mount points. It is the only part of the SFS client that needs to run as root, and the only part of the system that must function properly to prevent a machine from getting wedged. The SFS automounter daemon creates mount points dynamically as users access them. Finally, a collection of libraries and a novel RPC compiler simplify the task of implementing entirely non-blocking NFS loopback servers.

4.1 Basic API

The basic API of the toolkit is effectively the NFS 3 protocol [4]. The server allocates an `nfsserv` object, which might, for example, be bound to a UDP socket. The server hands this object a dispatch function. The object then calls the dispatch function with NFS 3 RPCs. The dispatch function is asynchronous. It receives an argument of type pointer to `nfscall`, and it returns nothing. To reply to an NFS RPC, the server calls the `reply` method of the `nfscall` object. This needn't happen before the dispatch routine returns, however. The `nfscall` can be stored away until some other asynchronous event completes.

4.2 The *nfsmounter* daemon

The purpose of *nfsmounter* is to clean up the mess when other parts of the system fail. This saves the loopback file system developer from having to reboot the machine, even if something goes horribly wrong with his loopback server. *nfsmounter* runs as root and calls the *mount* and *unmount* (or *umount*) system calls at the request of other processes. However, it aggressively distrusts these processes. Its interface is carefully crafted to ensure that *nfsmounter* can take over and assume control of a loopback mount whenever necessary.

nfsmounter communicates with other daemons through Unix domain sockets. To create a new NFS mount point, a daemon first creates a UDP socket over which to speak the NFS protocol. The daemon then passes this socket and the desired pathname for the mount point to *nfsmounter* (using Unix domain socket facilities for passing file descriptors across processes). *nfsmounter*, acting as an NFS client to existing loopback mounts, then probes the structure of any loopback file systems traversed down to the requested mount point. Finally, *nfsmounter* performs the actual mount system call and returns the result to the invoking daemon.

After performing a mount, *nfsmounter* holds onto the UDP socket of the NFS loopback server. It also remembers enough structure of traversed file systems to recreate any directories used as mount points. If a loopback server crashes, *nfsmounter* immediately detects this by receiving an end-of-file on the Unix domain socket connected to the server. *nfsmounter* then takes over any UDP sockets used by the crashed server, and begins serving the skeletal portions of the file system required to clean up underlying mount points. Requests to other parts of the file system return stale file handle errors, helping ensure most programs accessing the crashed file system exit quickly with an error, rather than hanging on a file access and therefore preventing the file system from being unmounted.

nfsmounter was built early in the development of SFS. After that point, we were able to continue development of SFS without any dedicated "crash boxes." No matter what bugs cropped up in the rest of SFS, we rarely needed a reboot. This mirrors the experience of students, who have used the toolkit for class projects without ever knowing the pain that loopback server development used to cause.

On occasion, of course, we have turned up bugs in kernel NFS implementations. We have suffered many kernel panics trying to understand these problems, but, strictly

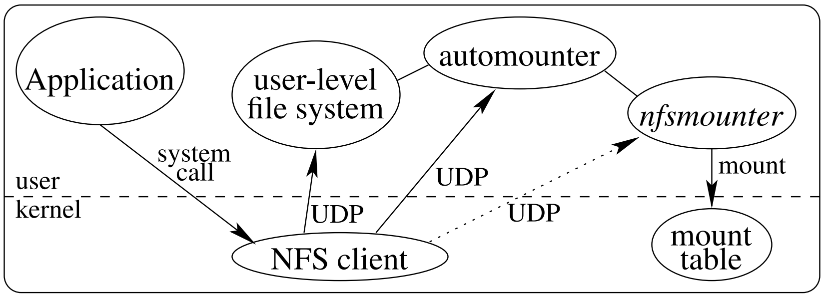

Figure 2: Architecture of the user-level file system toolkit

speaking, that part of the work qualifies as kernel development, not user-level server development.

4.3 Automounting in place

The SFS automounter shows that loopback automounters can mount file systems in place, even though no previous loopback automounter has managed to do so. SFS consists of a top level directory, `/sfs`, served by an automounter process, and a number of subdirectories of `/sfs` served by separate loopback servers. Subdirectories of `/sfs` are created on-demand when users access the directory names. Since subdirectories of `/sfs` are handled by separate loopback servers, they must be separate mount points.

The kernel's vnode locking strategy complicates the task of creating mount points on-demand. More specifically, when a user references the name of an as-yet-unknown mount point in `/sfs`, the kernel generates an NFS `LOOKUP` RPC. The automounter cannot immediately reply to this RPC, because it must first create a mount point. On the other hand, creating a mount point requires a *mount* system call during which the kernel again looks up the same pathname. The client NFS implementation will already have locked the `/sfs` directory during the first `LOOKUP` RPC. Thus the lookup within the *mount* call will hang.

Worse yet, the SFS automounter cannot always immediately create a requested mount point. It must validate the name of the directory, which involves a DNS lookup and various other network I/O. Validating a directory name can take a long time, particularly if a DNS server is down. The time can be sufficient to drive the NFS client into retransmission and have it lock the mount point, blocking all requests to `/sfs`. Thus, the automounter cannot sit on any `LOOKUP` request for a name in `/sfs`. It must reply immediately.

The SFS automounter employs two tricks to achieve what previous loopback automounters could not. First, it tags *nfsmounter*, the process that actually makes the mount system calls, with a reserved group ID (an idea first introduced by HLFSD [18]). By examining the credentials on NFS RPCs, then, the automounter can differentiate NFS calls made on behalf of *nfsmounter* from those issued for other processes. Second, the automounter creates a number of special "`.mnt`" mount points on directories with names of the form `/sfs/.mnt/0/`, `/sfs/.mnt/1/`, ... The automounter never delays a response to a `LOOKUP` RPC in the `/sfs` directory. Instead, it returns a symbolic link redirecting the user to another symbolic link in one of the `.mnt` mount points. There it delays the result of a `READLINK` RPC. Because the delayed readlink takes place under a dedicated mount point, however, no other file accesses are affected.

Meanwhile, as the user's process awaits a `READLINK` reply under `/sfs/.mnt/`n, the automounter actually mounts the remote file system under `/sfs`. Because *nfsmounter*'s NFS RPCs are tagged with a reserved group ID, the automounter responds differently to them—giving *nfsmounter* a different view of the file system from the user's. While users referencing the pathname in `/sfs` see a symbolic link to `/sfs/.mnt/`..., *nfsmounter* sees an ordinary directory on which it can mount the remote file system. Once the mount succeeds, the automounter lets the user see the directory, and responds to the pending `READLINK` RPC redirecting the user to the original pathname in `/sfs` which has now become a directory.

A final problem faced by automounters is that the commonly used *getcwd()* library function performs an *lstat* system call on every entry of a directory containing mount points, such as `/sfs`. Thus, if any of the loopback servers mounted on immediate subdirectories of `/sfs` become unresponsive, *getcwd()* might hang, even when run from within a working file system. Since loopback servers may depend on networked resources that become transiently unavailable, a loopback server may well need to become unavailable. When this happens, the loopback server notifies the automounter, and the automounter returns temporary errors to any process attempting to access the problematic mount point (or rather, to any process except *nfsmounter*, so that unavailable file systems can still be unmounted).

4.4 Asynchronous I/O library

Traditional I/O abstractions and interfaces are ill-suited to completely non-blocking programming of the sort required for NFS loopback servers. Thus, the SFS file system development toolkit contains a new C++ non-blocking I/O library, *libasync*, to help write programs that avoid any potentially blocking operations. When a function cannot complete immediately, it registers a callback with *libasync*, to be invoked when a particular asynchronous event occurs. At its core, *libasync* supports callbacks when file descriptors become ready for I/O, when child processes exit, when a process receives signals, and when the clock passes a particular time. A central dispatch loop polls for such events to occur through the system call *select*—the only blocking system call a loopback server ever makes.

Two complications arise from this style of event-driven programming in a language like C or C++. First, in languages that do not support closures, it can be inconvenient to bundle up the necessary state one must preserve to finish an operation in a callback. Second, when an asynchronous library function takes a callback and buffer as input and allocates memory for its results, the function's type signature does not make clear which code is responsible for freeing what memory when. Both complications easily lead to programming errors, as we learned bitterly in the first implementation of SFS which we entirely scrapped.

libasync makes asynchronous library interfaces less error-prone through aggressive use of C++ templates. A heavily overloaded template function, *wrap*, produces callback objects through a technique much like function currying: *wrap* bundles up a function pointer and some initial arguments to pass the function, and it returns a function object taking the function's remaining arguments. In other words, given a function:

```
res_t function (a1_t, a2_t, a3_t);
```

a call to `wrap (function, a1, a2)` produces a function object with type signature:

```
res_t callback (a3_t);
```

This *wrap* mechanism permits convenient bundling of code and data into callback objects in a type-safe way. Though the example shows the wrapping of a simple function, *wrap* can also bundle an object and method pointer with arguments. *wrap* handles functions and arguments of any type, with no need to declare the combination of types ahead of time. The maximum number of arguments is determined by a parameter in a perl script that actually generates the code for wrap.

```
class foo : public bar {
  /* ... */
};

void
function ()
{
  ref<foo> f = new refcounted<foo>
     (/* constructor arguments */);
  ptr<bar> b = f;
  f = new refcounted<foo>
     (/* constructor arguments */);
  b = NULL;
}
```

Figure 3: Example usage of reference-counted pointers

To avoid the programming burden of tracking which of a caller and callee is responsible for freeing dynamically allocated memory, *libasync* also supports reference-counted garbage collection. Two template types offer reference-counted pointers to objects of type `T`—`ptr<T>` and `ref<T>`. `ptr` and `ref` behave identically and can be assigned to each other, except that a `ref` cannot be NULL. One can allocate a reference-counted version of any type with the template type `refcounted<T>`, which takes the same constructor arguments as type `T`. Figure 3 shows an example use of reference-counted garbage collection. Because reference-counted garbage collection deletes objects as soon as they are no longer needed, one can also rely on destructors of reference-counted objects to release resources more precious than memory, such as open file descriptors.

libasync contains a number of support routines built on top of the core callbacks. It has asynchronous file handles for input and formatted output, an asynchronous DNS resolver, and asynchronous TCP connection establishment. All were implemented from scratch to use *libasync*'s event dispatcher, callbacks, and reference counting. *libasync* also supplies helpful building blocks for objects that accumulate data and must deal with short writes (when no buffer space is available in the kernel). Finally, it supports asynchronous logging of messages to the terminal or system log.

4.5 Asynchronous RPC library and compiler

The SFS toolkit also supplies an asynchronous RPC library, *libarpc*, built on top of *libasync*, and a new RPC compiler, *rpcc*. *rpcc* compiles Sun XDR data structures into C++ data structures. Rather than directly output code for serializing the structures, however, *rpcc* uses templates and function overloading to produce a generic way of traversing data structures at compile time. This allows one to write concise code that actually compiles to a number of functions, one for each NFS data type. Serialization of data in RPC calls is but one application of this traversal mechanism. The ability to traverse NFS data structures automatically turned out to be useful in a number of other situations.

As an example, one of the loopback servers constituting the client side of SFS uses a protocol very similar to NFS for communicating with remote SFS servers. The only differences is that the SFS protocol has more aggressive file attribute caching and lets the server call back to the client to invalidate attributes. Rather than manually extract attribute information from the return structures of 21 different NFS RPCs, the SFS client uses the RPC library to traverse the data structures and extract attributes automatically. While the compiled output consists of numerous functions, most of these are C++ template instantiations automatically generated by the compiler. The source needs only a few functions to overload the traversal's behavior on attribute structures. Moreover, any bug in the source will likely break all 21 NFS functions.

4.6 Stackable NFS manipulators

An SFS support library, *libsfsmisc*, provides stackable NFS manipulators. Manipulators take one `nfsserv` object and produce a different one, manipulating any calls from and replies to the original `nfsserv` object. A loopback NFS server starts with an initial `nfsserv` object, generally `nfsserv_udp` which accepts NFS calls from a UDP socket. The server can then push a bunch of manipulators onto this `nfsserv`. For example, over the course of developing SFS we stumbled across a number of bugs that caused panics in NFS client implementations. We developed an NFS manipulator, `nfsserv_fixup`, that works around these bugs. SFS's loopback servers push `nfsserv_fixup` onto their NFS manipulator stack, and then don't worry about the specifics of any kernel bugs. If we discover another bug to work around, we need only put the workaround in a single place to fix all loopback servers.

Another NFS manipulator is a demultiplexer that breaks a single stream of NFS requests into multiple streams. This allows a single UDP socket to be used as the server side for multiple NFS mount points. The demultiplexer works by tagging all NFS file handles with the number of the mount point they belong to. Though file handles are scattered throughout the NFS call and return types, the tagging was simple to implement using the traversal feature of the RPC compiler.

4.7 Miscellaneous features

The SFS toolkit has several other features. It supplies a small, user-level module, *mallock.o*, that loopback servers must link against to avoid paging. On systems supporting the *mlockall()* system call, this is easily accomplished. On other systems, *mallock.o* manually pins the text and data segments and replaces the *malloc()* library function with a routine that always returns pinned memory.

Finally, the SFS toolkit contains a number of debugging features, including aggressive memory checking, type checking for accesses to RPC union structures, and easily toggleable tracing and pretty-printing of RPC traffic. Pretty-printing of RPC traffic in particular has proven an almost unbeatable debugging tool. Each NFS RPC typically involves a limited amount of computation. Moreover, separate RPC calls are relatively independent of each other, making most problems easily reproducible. When a bug occurs, we turn on RPC tracing, locate the RPC on which the server is returning a problematic reply, and set a conditional breakpoint to trigger under the same conditions. Once in the debugger, it is generally just a matter of stepping through a few functions to understand how we arrive from a valid request to an invalid reply.

Despite the unorthodox structure of non-blocking daemons, the SFS libraries have made SFS's 60,000+ lines of code (including the toolkit and all daemons) quite manageable. The trickiest bugs we have hit were in NFS implementations. At least the SFS toolkit's tracing facilities let us quickly verify that SFS was behaving correctly and pin the blame on the kernel.

4.8 Limitations of NFS loopback servers

Despite the benefits of NFS loopback servers and their tractability given the SFS toolkit, there are two serious drawbacks that must be mentioned. First, the NFS 2 and 3 protocols do not convey file closes to the server. There are many reasons why a file system implementor might wish to know when files are closed. We have implemented a close simulator as an NFS manipulator, but it cannot be 100% accurate and is thus not suitable for all needs. The NFS 4 protocol [16] does have file closes, which will solve this problem if NFS 4 is deployed.

The other limitation is that, because of the potential risk of deadlock in loopback servers, one can never predicate the completion of an NFS write on that of a write issued to local disk. Loopback servers can access the local disk, provided they do so asynchronously. *libasync* offers support for doing so using helper processes, or one can do so as an NFS loopback client. Thus, one can build a loopback server that talks to a remote server and keeps a cache on the local disk. However, the loopback server must return from an NFS write once the corresponding remote operation has gone through; it cannot wait for the write to go through in the local cache. Thus, techniques that rely on stable, crash-recoverable writes to local disk, such as those for disconnected operation in CODA [12], cannot easily be implemented in loopback servers; one would need to use raw disk partitions.

5 NFS loopback clients

In addition to implementing loopback servers, *libarpc* allows applications to behave as NFS clients, making them loopback clients. An NFS loopback client accesses the local hard disk by talking to an in-kernel NFS server, rather than using the standard POSIX open/close/read/write system call interface. Loopback clients have none of the disadvantages of loopback servers. In fact, a loopback client can still access the local file system through system calls. NFS simply offers a lower-level, asynchronous alternative from which some aggressive applications can benefit.

The SFS server software is actually implemented as an NFS loopback client using *libarpc*. It reaps a number of benefits from this architecture. The first is performance. Using asynchronous socket I/O, the loopback client can have many parallel disk operations outstanding simultaneously. This in turn allows the operating system to achieve better disk arm scheduling and get higher throughput from the disk. Though POSIX does offer optional *aio* system calls for asynchronous file I/O, the *aio* routines only operate on open files. Thus, without the NFS loopback client, directory lookups, directory reads, and file creation would all still need to be performed synchronously.

The second benefit of the SFS server as a loopback client is security. The SFS server is of course trusted, as it may be called upon to serve or modify any file. The server must therefore be careful not to perform any operation not permitted to the requesting users. Had the server been implemented on top of the normal file system interface, it would also have needed to perform access control—deciding on its own, for instance, whether or not to honor a request to delete a file. Making such decisions correctly without race conditions is actually quite tricky to do given only the Unix file system interface.[1] As an NFS client, however, it is trivial to do. Each NFS request explicitly specifies the credentials with which to execute the request (which will generally be less privileged than the loopback client itself). Thus, the SFS server simply tags NFS requests with the appropriate user credentials, and the kernel's NFS server makes the access control decision and performs the operation (if approved) atomically.

The final benefit of having used a loopback client is in avoiding limits on the number of open files. The total number of open files on SFS clients connected to an SFS server may exceed the maximum number of open files allowed on the server. As an NFS loopback client, the SFS server can access a file without needing a dedicated file descriptor.

A user of the SFS toolkit actually prototyped an SFS server that used the POSIX interface rather than act as a loopback client. Even without implementing leases on attributes, user authentication, or unique `st_ino` fields, the code was almost as large as the production SFS server and considerably more bug-prone. The POSIX server had to jump through a number of hoops to deal with such issues as the maximum number of open files.

5.1 Limitations of NFS loopback clients

The only major limitation on NFS loopback clients is that they must run as root. Unprivileged programs cannot access the file system with the NFS interface. A related concern is that the value of NFS file handles must be carefully guarded. If even a single file handle of a directory is disclosed to an untrusted user, the user can access any part of the file system as any user. Fortunately, the SFS RPC compiler provides a solution to this problem. One can easily traverse an arbitrary NFS data structure and encrypt or decrypt all file handles encountered. The SFS toolkit contains a support routine for doing so.

A final annoyance of loopback clients is that the file systems they access must be exported via NFS. The actual mechanics of exporting a file system vary significantly between versions of Unix. The toolkit does not yet have a way of exporting file systems automatically. Thus, users must manually edit system configuration files before the loopback client will run.

[1] In the example of deleting a file, the server would need to change its working directory to that of the file. Otherwise, between the server's access check and its *unlink* system call, a bad user could replace the directory with a symbolic link, thus tricking the server into deleting a file in a different (unchecked) directory. Cross-directory renames are even worse—they simply cannot be implemented both atomically and securely. An alternative approach might be for the server to drop privileges before each file system operation, but then unprivileged users could send signals to the server and kill it.

# Lines	Function
19	includes and global variables
22	command-line argument parsing
8	locate and spawn *nfsmounter*
5	get server's IP address
11	ask server's portmap for NFS port
14	ask server for NFS file handle
20	call *nfsmounter* for mount
20	relay NFS calls
119	**Total**

Figure 4: Lines of code in *dumbfs*

6 Applications of the toolkit

The SFS toolkit has been used to build a number of systems. SFS itself is a distributed file system consisting of two distinct protocols, a read-write and a read-only protocol. On the client side, each protocol is implemented by a separate loopback server. On the server side, the read-write protocol is implemented by a loopback client. (The read-only server uses the POSIX interface.) A number of non-SFS file systems have been built, too, including a file system interface to CVS, a file system to FTP/HTTP gateway, and file system interfaces to several databases.

The toolkit also lets one develop distributed file systems and fit them into the SFS framework. The SFS read-write protocol is very similar to NFS 3. With few modifications, therefore, one can transform a loopback server into a network server accessible from any SFS client. SFS's libraries automatically handle key management, encryption and integrity checking of session traffic, user authentication, and mapping of user credentials between local and remote machines. Thus, a distributed file system built with the toolkit can without much effort provide a high level of security against network attacks.

Finally, the asynchronous I/O library from the toolkit has been used to implement more than just file systems. People have used it to implement TCP proxies, caching web proxies, and TCP to UDP proxies for networks with high loss. Asynchronous I/O is an extremely efficient tool for implementing network servers. A previous version of the SFS toolkit was used to build a high performance asynchronous SMTP mail server that survived a distributed mail-bomb attack. (The "hybris worm" infected thousands of machines and made them all send mail to our server.)

6.1 *dumbfs* – A simple loopback server

To give a sense for the complexity of using the SFS toolkit, we built *dumbfs*, the simplest possible loopback file system. *dumbfs* takes as arguments a server name, a pathname, and a mount point. It creates a loopback NFS mount on the mount point, and relays NFS calls to a remote NFS server. Though this functionality may sound worthless, such a utility does actually have a use. Because the SFS RPC libraries will trace and pretty-print RPC traffic, *dumbfs* can be used to analyze exactly how an NFS client and server are behaving.

The implementation of *dumbfs* required 119 lines of code (including blank lines for readability), as shown in Figure 4. Notably missing from the breakdown is any clean-up code. *dumbfs* does not need to catch any signals. If it dies for any reason, *nfsmounter* will clean up the mount point.

Figure 5 shows the implementation of the NFS call relaying code. Function `dispatch` is called for each NFS RPC. The first two lines manipulate the "auth unix parameters" of the NFS call—RPC terminology for user and group IDs. To reuse the same credentials in an outgoing RPC, they must be converted to an `AUTH *` type. The `AUTH *` type is defined by the RPC library in the operating system's C library, but the `authopaque` routines are part of the SFS toolkit.

The third line of `dispatch` makes an outgoing NFS RPC. `nfsc` is an RPC handle for the remote NFS server. `nc->getvoidarg` returns a pointer to the RPC argument structure, cast to `void *`. `nc->getvoidres` similarly returns a pointer to the appropriate result type, also cast to `void *`. Because the RPC library is asynchronous, `nfsc->call` will return before the RPC completes. `dispatch` must therefore create a callback, using `wrap` to bundle together the function `reply` with the argument `nc`.

When the RPC finishes, the library makes the callback, passing an additional argument of type `clnt_stat` to indicate any RPC-level errors (such as a timeouts). If such an error occurs, it is logged and propagated back as the generic RPC failure code `SYSTEM_ERR`. `warn` is the toolkit's asynchronous logging facility. The syntax is similar to C++'s `cout`, but `warn` additionally converts RPC and NFS `enum` error codes to descriptive strings. If there is no error, `reply` simply returns the result data structure just filled in by the outgoing NFS RPC.

6.1.1 *dumbfs* performance

To analyze the inherent overhead of a loopback server, we measured the performance of *dumbfs*. We ran experiments between two 800 MHz Pentium III machines, each with 256 MBytes of PC133 RAM and a Seagate ST318451LW disk. The two machines were connected with 100 Mbit/sec switched Ethernet. The client ran FreeBSD 4.2, and the server OpenBSD 2.8.

```
void
dispatch (nfscall *nc)
{
  static AUTH *ao = authopaque_create ();
  authopaque_set (ao, nc->getaup ());
  nfsc->call (nc->proc (), nc->getvoidarg (), nc->getvoidres (),
              wrap (reply, nc), ao);
}

static void
reply (nfscall *nc, enum clnt_stat stat)
{
  if (stat) {
    warn << "NFS server: " << stat << "\n";
    nc->reject (SYSTEM_ERR);
  }
  else
    nc->reply (nc->getvoidres ());
}
```

Figure 5: *dumbfs* dispatch routine

To isolate the loopback server's worst characteristic, namely its latency, we measured the time to perform an operation that requires almost no work from the server—an unauthorized *fchown* system call. NFS required an average of 186 μsec, *dumbfs* 320 μsec. Fortunately, raw RPC latency is a minor component of the performance of most real applications. In particular, the time to process any RPC that requires a disk seek will dwarf *dumbfs*'s latency. As shown in Figure 7, the time to compile emacs 20.7 is only 4% slower on *dumbfs* than NFS. Furthermore, NFS version 4 has been specifically designed to tolerate high latency, using such features as batching of RPCs. In the future, latency should present even less of a problem for NFS 4 loopback servers

To evaluate the impact of *dumbfs* on data movement, we measured the performance of sequentially reading a 100 MByte sparse file that was not in the client's buffer cache. Reads from a sparse file cause an NFS server to send blocks of zeros over the network, but do not require the server to access the disk. This represents the worst case scenario for *dumbfs*, because the cost of accessing the disk is the same for both file systems and would only serve to diminish the relative difference between NFS 3 and *dumbfs*. NFS 3 achieved a throughput of 11.2 MBytes per second (essentially saturating the network), while *dumbfs* achieved 10.3. To get a rough idea of CPU utilization, we used *top* command to examine system activity while streaming data from a 50 GByte sparse file through *dumbfs*. The idle time stayed between 30–35%.

6.2 *cryptfs* – An encrypting file system

As a more useful example, we built an encrypting file system in the spirit of CFS [3]. Starting from *dumbfs*, it took two evenings and an additional 600 lines of C++ to build *cryptfs*—a cryptographic file system with a very crude interface. *cryptfs* takes the same arguments as *dumbfs*, prompts for a password on startup, then encrypts all file names and data.

cryptfs was inconvenient to use because it could only be run as root and unmounting a file system required killing the daemon. However, it worked so well that we spent an additional week building *cryptfsd*—a daemon that allows multiple encrypted directories to be mounted at the request of non-root users. *cryptfsd* uses almost the same encrypting NFS translator as *cryptfs*, but is additionally secure for use on machines with untrusted non-root users, provides a convenient interface, and works with SFS as well as NFS,

6.2.1 Implementation

The final system is 1,920 lines of code, including *cryptfsd* itself, a utility *cmount* to mount encrypted directories, and a small helper program *pathinfo* invoked by *cryptfsd*. Figure 6.2 shows a breakdown of the lines of code. By comparison, CFS is over 6,000 lines (though of course it has different features from *cryptfs*). CFS's NFS request handing code (its analogue of `nfs.C`) is 2,400 lines.

cryptfsd encrypts file names and contents using Rijn-

# Lines	File	Function
223	`cryptfs.h`	Structure definitions, inline & template functions, global declarations
90	`cryptfsd.C`	*main* function, parse options, launch *nfsmounter*
343	`findfs.C`	Translate user-supplied pathname to NFS/SFS server address and file handle
641	`nfs.C`	NFS dispatch routine, encrypt/decrypt file names & contents
185	`afs.C`	NFS server for `/cfs`, under which encrypted directories are mounted
215	`adm.C`	*cryptfsadm* dispatch routine—receive user mount requests
26	`cryptfsadm.x`	*cryptfsadm* RPC protocol for requesting mounts from *cryptfsd*
63	`cmount.C`	Utility to mount an encrypted directory
134	`pathinfo.c`	Helper program—run from `findfs.C` to handle untrusted requests securely
1,920	**Total**	

Figure 6: Lines of code in *cryptfsd*. The last two source files are for stand-alone utilities.

dael, the AES encryption algorithm. File names are encrypted in CBC mode, first forwards then backwards to ensure that every byte of the encrypted name depends on every byte of the original name. Symbolic links are treated similarly, but have a 48-bit random prefix added so that two links to the same file will have different encryptions.

The encryption of file data does not depend on other parts of the file. To encrypt a 16-byte block of plaintext file data, *cryptfsd* first encrypts the block's byte offset in the file and a per-file "initialization vector," producing 16 bytes of random-looking data. It exclusive-ors this data with the plaintext block and encrypts again. Thus, any data blocks repeated within or across files will have different encryptions.

cryptfsd could have used a file's inode number or a hash of its NFS file handle as the initialization vector. Unfortunately, such vectors would not survive a traditional backup and restore. Instead, *cryptfsd* stores the initialization vector at the beginning of the file. All file contents is then shifted forward 512 bytes. We used 512 bytes though 8 would have sufficed because applications may depend on the fact that modern disks write aligned 512-byte sectors atomically.

Many NFS servers use 8 KByte aligned buffers. Shifting a file's contents can therefore hurt the performance of random, aligned 8 KByte writes to a large file; the server may end up issuing disk reads to produce complete 8 KByte buffers. Fortunately, reads are not necessary in the common case of appending to files. Of course, *cryptfs* could have stored the initialization vector elsewhere. CFS, for instance, stores initialization vectors outside of files in symbolic links. This technique incurs more synchronous disk writes for many metadata operations, however. It also weakens the semantics of the atomic rename operation. We particularly wished to avoid deviating from traditional crash-recovery semantics for operations like rename.

Like CFS, *cryptfs* does not handle sparse files properly. When a file's size is increased with the *truncate* system call or with a *write* call beyond the file's end, any unwritten portions of the file will contain garbage rather than 0-valued bytes.

The SFS toolkit simplified *cryptfs*'s implementation in several ways. Most importantly, every NFS 3 RPC (except for NULL) can optionally return file attributes containing file size information. Some operations additionally return a file's old size before the RPC executed. *cryptfs* must adjust these sizes to hide the fact that it shifts file contents forward. (Also, because Rijndael encrypts blocks of 16 bytes, *cryptfs* adds 16 bytes of padding to files whose length is not a multiple of 16.)

Manually writing code to adjust file sizes wherever they appear in the return types of 21 different RPC calls would have been a daunting and error-prone task. However, the SFS toolkit uses the RPC compiler's data structure traversal functionality to extract all attributes from any RPC data structure. Thus, *cryptfs* only needs a total of 15 lines of code to adjust file sizes in all RPC replies.

cryptfs's implementation more generally benefited from the SFS toolkit's single dispatch routine architecture. Traditional RPC libraries call a different service function for every RPC procedure defined in a protocol. The SFS toolkit, however, does not demultiplex RPC procedures. It passes them to a single function like `dispatch` in Figure 5. When calls do not need to be demultiplexed (as was the case with *dumbfs*), this vastly simplifies the implementation.

File name encryption in particular was simplified by the single dispatch routine architecture. A total of 9 NFS 3 RPCs contain file names in their argument types. However, for 7 of these RPCs, the file name and directory file handle are the first thing in the argument structure. These calls can be handled identically. *cryptfs* therefore implements file name encryption in a switch statement with 7 cascaded "case" statements and two special cases.

(Had the file names been less uniformly located in argument structures, of course, we could have used the RPC traversal mechanism to extract pointers to them.)

Grouping code by functionality rather than RPC procedure also results in a functioning file system at many more stages of development. That in turn facilitates incremental testing. To develop *cryptfs*, we started from *dumbfs* and first just fixed the file sizes and offsets. Then we special-cased read and write RPCs to encrypt and decrypt file contents. Once that worked, we added file name encryption. At each stage we had a working file system to test.

A function to "encrypt file names whatever the RPC procedure" is easy to test and debug when the rest of the file system works. With a traditional demultiplexing RPC library, however, the same functionality would have been broken across 9 functions. The natural approach in that case would have been to implement one NFS RPC at a time, rather than one feature at a time, thus arriving at a fully working system only at the end.

6.2.2 *cryptfs* performance

To evaluate *cryptfs*'s performance, we measured the time to untar the emacs 20.7 software distribution, configure the software, compile it, and then delete the build tree. Figure 7 shows the results. The white bars indicate the performance on FreeBSD's local FFS file system. The solid gray bars show NFS 3 over UDP. The solid black bars show the performance of *cryptfs*. The leftward slanting black bars give the performance of *dumbfs* for comparison.

The untar phase stresses data movement and latency. The delete and configure phases mostly stress latency. The compile phase additionally requires CPU time—it consumes approximately 57 seconds of user-level CPU time as reported by the time command. In all phases, *cryptfs* is no more than 10% slower then NSF3. Interestingly enough, *cryptfs* actually outperforms NFS 3 in the delete phase. This does not mean that *cryptfs* is faster at deleting the same files. When we used NFS 3 to delete an emacs build tree produced by *cryptfs*, we observed the same performance as when deleting it with *cryptfs*. Some artifact of *cryptfs*'s file system usage—perhaps the fact that almost all file names are the same length—results in directory trees that are faster to delete.

For comparison, we also ran the benchmark on CFS using the Blowfish cipher.[2] The black diagonal stripes labeled "CFS-async" show its performance. CFS outperforms even straight NFS 3 on the untar phase. It does so by performing asynchronous file writes when it should perform synchronous ones. In particular, the *fsync* system call does nothing on CFS. This is extremely dangerous—particularly since CFS runs a small risk of deadlocking the machine and requiring a reboot. Users can loose the contents of files they edit.

We fixed CFS's dangerous asynchronous writes by changing it to open all files with the `O_FSYNC` flag. The change did not affect the number system calls made, but ensured that writes were synchronous, as required by the NFS 2 protocol CFS implements. The results are shown with gray diagonal stipes, labeled "CFS-sync." CFS-sync performs worse than *cryptfs* on all phases. For completeness, we also verified that *cryptfs* can beat NFS 3's performance by changing all writes to unstable and giving fake replies to COMMIT RPCs—effectively what CFS does.

Because of the many architectural differences between *cryptfs* and CFS, the performance measurements should not be taken as a head-to-head comparison of SFS's asynchronous RPC library with the standard *libc* RPC facilities that CFS uses. However, CFS is a useful package with performance many that people find acceptable given its functionality. These experiments show that, armed with the SFS toolkit, the author could put together a roughly equivalent file system in just a week and a half.

Building *cryptfsd* was a pleasant experience, because every little piece of functionality added could immediately be tested. The code that actually manipulates NFS RPCs is less than 700 lines. It is structured as a bunch of small manipulations to NFS data structures being passed between and NFS client and server. There is a mostly one-to-one mapping from RPCs received to those made. Thus, it is easy for *cryptfs* to provide traditional crash-recovery semantics.

7 Summary

User-level software stands to gain much by using NFS over the loopback interface. By emulating NFS servers, portable user-level programs can implement new file systems. Such loopback servers must navigate several tricky issues, including deadlock, vnode and mount point locking, and the fact that a loopback server crash can wedge a machine and require a reboot. The SFS file system development toolkit makes it relatively easy to avoid these problems and build production-quality loopback NFS servers.

The SFS toolkit also includes an asynchronous RPC library that lets applications access the local file system using NFS. For aggressive applications, NFS can actu-

[2] Actually, CFS did not execute the full benchmark properly—some directories could not be removed, putting `rm` into an infinite loop. Thus, for CFS only, we took out the benchmark's `rm -rf` command, instead deleting as much of the build tree as possible with find/xargs and then renaming the `emacs-20.7` directory to a garbage name.

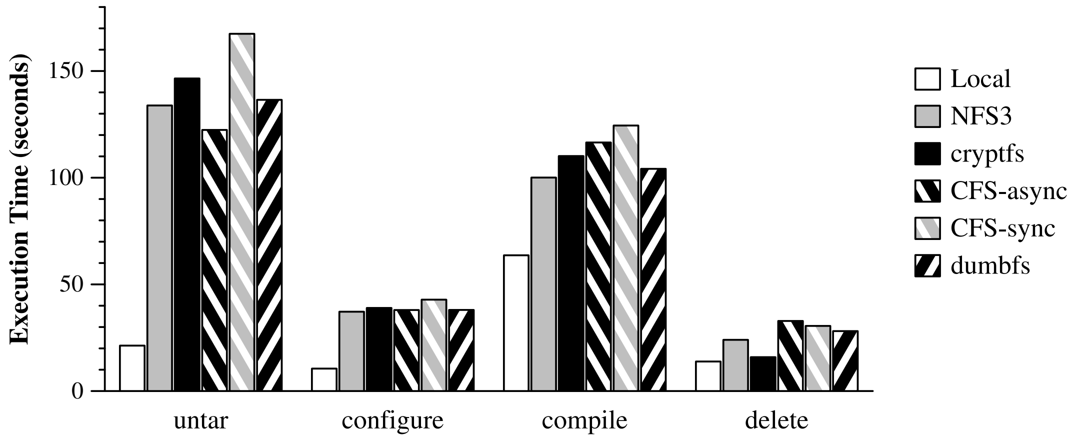

Figure 7: Execution time of emacs untar, configure, compile, and delete.

ally be a better interface than the traditional *open/close/read/write*. NFS allows asynchronous access to the file system, even for operations such as file creation that are not asynchronously possible through system calls. Moreover, NFS provides a lower-level interface that can help avoid certain race conditions in privileged software.

Acknowledgments

The author is grateful to Frans Kaashoek for implementing the POSIX-based SFS server mentioned in Section 5, for generally using and promoting the SFS toolkit, and for his detailed comments on several drafts of this paper. The author also thanks Marc Waldman, Benjie Chen, Emmett Witchel, and the anonymous reviewers for their helpful feedback. Finally, thanks to Yoonho Park for his help as shepherd for the paper.

Availability

The SFS file system development toolkit is free software, released under version 2 of the GNU General Public License. The toolkit comes bundled with SFS, available from `http://www.fs.net/`.

References

[1] 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.

[2] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

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

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

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

[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, February 1999.

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

[8] Jeremy Fitzhardinge. UserFS. `http://www.goop.org/~jeremy/userfs/`.

[9] 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.

[10] 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.

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

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

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

[14] Jan-Simon Pendry and Nick Williams. *Amd The 4.4 BSD Automounter Reference Manual*. London, SW7 2BZ, UK. Manual comes with amd software distribution.

[15] 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.

[16] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, Network Working Group, December 2000.

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

[18] Erez Zadok and Alexander Dupuy. HLFSD: Delivering email to your $HOME. In *Proceedings of the Systems Administration Conference (LISA VII)*, Monterey, CA, November 1993. USENIX.

[19] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In *Proceedings of the 2000 USENIX*. USENIX, June 2000.