File System Support for Collaboration in the Wide Area

Vasile Gaburici, Pete Keleher, and Bobby Bhattacharjee
Department of Computer Science
University of Maryland
College Park, MD 20742
{gaburici,keleher,bobby}@cs.umd.edu

Abstract

We describe the design, implementation, and performance of MFS, a new file system designed to support efficient wide-area collaboration. MFS is structured around the twin abstractions of lightweight *sessions* and *snapshots*, along with a highly configurable capability-based security architecture. Sessions simplify and clarify collaborative semantics. Snapshots allow atomic access to arbitrary collections of files, and allow sharing to be defined in a simple and expressive fashion.

MFS's security architecture is a layered system that allows diverse usage scenarios. Pure capability-based access allows clients to access data without needing expensive public key or authentication servers, or complicated administration. However, MFS's capabilities can also be *watermarked*, allowing a range of services to be added on a per-mount basis, up to and including traditional user authentication based on passwords or public keys.

Basing the system around the use of immutable snapshots enables the underlying system to use several performance optimizations aggressively. Performance results from our MFS prototype show that, far from adding overhead, the use of snapshots allows the system to perform comparably to NFS in the local-area case and significantly outperform existing systems in wide-area environments.

1 Introduction

This paper describes the design and implementation of MFS, a new file system intended to support small, dynamic, collaborating groups in the wide-area. MFS differs from previous systems in its support for both transparent and non-transparent replication, and the expressiveness of its capability-based security architecture. In the rest of this section we argue that all three characteristics are necessary to efficiently support wide-area collaboration, and make the claim that neither existing file systems nor applications are sufficient.

Replication in data management systems is usually motivated by the need for either high performance or availability [17]. Such replication is transparent in that the view of data seen by users and applications preserves single-copy semantics. However, allowing users to temporarily see diverging replicas is desirable in some cases, even when all clients are fully connected.

For example, consider CVS [5] and similar version control systems. The most important innovation that CVS pioneered is the *unreserved checkout* feature. This feature allows developers to work in parallel on potentially diverging replicas (working copies) of multiple files containing mutual dependencies. However, developers must be aware of changes committed by others, and merge those changes into their working copies before committing their own changes. As a simple example, a source file and its header might need to agree on the prototype of a function defined therein, or other clients will not be able to compile the program's source. A change to the function's parameter list must be reflected in both the function's definition in the source file, and in the prototype definition in the header. The changes to both must be made visible together in a single atomic action, much like transactional semantics.

A generalization of this type of collaboration is often termed either *asynchronous* or *autonomous collaboration* (see Edwards et al. [9] for a discussion), where participants temporarily work independently before combining their efforts. Asynchronous collaboration is characterized by the need for users to explicitly authorize files being merged, or made visible to others. In other words, such systems require replication to be non-transparent. By implication, a file system intended to support collaboration must support both transparent *and* non-transparent replication, and in the latter case mutual dependencies between files must be preserved.

A related issue is that of *naming*: clients can only share collections of files if there is some way to specify exactly which files, and versions, are being shared.

Sessions and Snapshots MFS is structured around lightweight sessions and snapshots, both of which are mountable entities. A session is similar to a mounted directory in a traditional file system, providing readable and writable files and directories. A snapshot is an immutable view of a session at a single instant in time. Snapshots are extremely lightweight, having only constant time and space requirements. New sessions, in turn, are created by writing to mounted snapshots.

MFS supports the usual transparent replication through *shared sessions*, sessions explicitly joined by multiple users. All clients joining a shared session see each other's updates

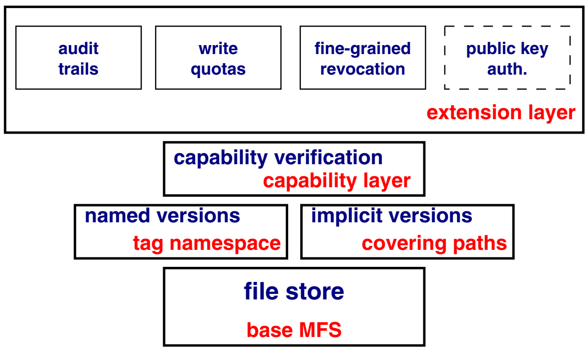

Figure 1: MFS Architecture: the components in solid rectangles have been implemented and are described in this paper.

in best-effort manner, similarly to traditional file systems. Sessions can continue indefinitely, as they survive machine reboots and have no monotonically growing state other than file updates. Joining a shared session, then, is conceptually quite similar to mounting a directory in a traditional file system. Permanently mounting a session merely requires specification of a server name, a session ID, and an appropriate capability.

MFS supports non-transparent replication through the implicit forks potentially resulting from writes to mounted snapshots. Writing to a mounted snapshot generates a new session because snapshots themselves are immutable. File and directory modifications made in one session are invisible in other sessions, so two clients mounting and writing to the same snapshot implicitly create a fork.

Sessions and snapshots can be combined through explicit *merge* procedures. While MFS can automatically merge entities that differ only in non-conflicting directory operations, more generalized merging must be controlled by the application layer. This is appropriate, as the semantics of file contents differ according to the application, and no application-independent merger can handle all cases.

Security MFS's security architecture consists of two layers: a *capability layer*, in which the basic access control functionality is defined, and an *extension layer*, which allows a variety of additional services and policies to be defined on a per-mount basis. As a result, MFS can be configured to implement a variety of very different security policies. These policies can range from pure capability-based access, which allows clients to access data without needing expensive public key or authentication servers, to identity-based approaches that rely on password or public-key authentication.

User-based access control is particularly awkward in wide-area environments when multiple administrative domains are involved. The problem is that clients must be mapped to credentials accepted at the server, which entails slow and cumbersome out-of-band communication. The common practice of mapping remote users to existing local user classes also poses the threat of implicit rights amplification, where users requiring only limited rights are given stronger rights than necessary.

MFS's capability layer grants access purely on the basis of capabilities: access is granted to any client that can provide the proper capability. As such, an MFS capability consists of a session or snapshot ID and a hash of a corresponding secret, without any specification of valid users, or indeed anything to authenticate the intended user of a capability.

This minimalist approach has two primary advantages. First, access to sessions and snapshots can be granted to a new client simply by securely transferring an appropriate capability, without involving any authentication server, or even the server that hosts the session or snapshot. The client is never authenticated, and does not need an account on the server. Second, clients can locally derive new capabilities with reduced rights (e.g., access to a restricted set of files in the snapshot), allowing access rights to be granted at exactly the desired level. The ability to share files and directories without requiring clients to be authenticated, or have accounts at the server, makes sharing quite flexible and efficient.

The protections provided by the base capability layer are augmented by a variety of services in the extension layer. These services work by using *watermarked* capabilities. Watermarking adds an indelible tag to a capability. Services such as auditing, quota-checking, and fine-grained revocation can be added by tracking such tags. Further, the capability layer could be augmented with a translation layer that can implement traditional authentication services that use passwords or public keys.

MFS's security architecture is described in more detail in Gaburici et al. [12].

Why Not CVS? Application-layer tools, such as CVS [5], Subversion [3], and ClearCase [7], address both the non-transparent replication and naming problems by allowing users to specify versioned sets of files. MFS may be thought of as an exercise in understanding the issues involved in moving these abstractions into the operating system. There are several reasons why moving this functionality into the operating system might be appropriate.

The first is performance. Integrating the versioning into the file system allows several low-level performance optimizations, including fast caching in the kernel, prefetching, and "trickle" writebacks rather than a single large write at commit time. Since the sharing is visible to the underlying file system, the system can pipeline, aggregate, and sometimes eliminate updates when it detects that a session is not shared (i.e., single-writer semantics). Section 6 discusses performance in some detail.

The second issue is that of convenience, i.e., the same reason that local area file systems are preferred over using `ftp` to emulate shared access. Integrating the versioning and replication support into the file system allows explicit check-ins to be avoided because all files are versioned. Relationships between files need not be specified until after the fact, as a single snapshot can cover any set of files, not just those that were entered together in the same initial check-in. Automating the versioning removes one source of user error. Moving the versioning functionality into the file system allows it to be used transparently by any application that can read or write files.

The last major issue is that of access control. MFS has an integrated security architecture that grants access to any client presenting a session or snapshot ID and an appropriate capability. The client does not have to establish an account on the server, and inter-administrative domain agreements need not be negotiated. By contrast, a user must have an account at a CVS server before checking out files from a repository. Authentication mechanisms could be duplicated at the application level, but would not be integrated into the operating system mechanisms, possibly exposing confidential data.

MFS does not implement the full functionality of CVS, much less that of ClearCase or Subversion. Supporting versioning abstractions in the file system does not eliminate the need for such applications, but it potentially makes their implementation much simpler. A variety of "thin" CVS-like applications can be built on top of MFS far more easily and efficiently than equivalent applications on conventional file systems.

1.1 Road Map

Figure 1 shows the principal layers of the MFS design. An MFS server exports sessions and lightweight snapshots, and implements a versioning file store (Section 2). Since snapshots are immutable, they cannot be overwritten or obscured. File modifications are written to new versions that only become visible to other clients when they are included in new snapshots.

Section 2 describes the base consistency-related abstractions, and Section 3 describes the *tag namespace* and *covering path* abstractions, which allow convenient naming and searching for session and snapshot IDs. Above this is the base security layer, which is not discussed further in this paper. Section 4 gives a brief overview of the use of our system, Section 5 describes the current prototype, and the results in Section 6 show that MFS generally performs significantly better than the alternatives. Finally, we discuss related work in Section 7, and summarize our work in Section 8.

2 Sessions and Snapshots

This section describes MFS's central consistency-related abstractions. A *session* [40] is conceptually a materialization of the complete execution of a log of file system update operations performed to a to an initial immutable state. These actions include file and directory creates, deletes, and writes. Each session has a unique identifier, and can be mounted by specifying it directly.

An *anonymous snapshot*, defined by a session ID and a timestamp, is an immutable materialization of the execution of log entries up to and including the time specified by a timestamp. Such a snapshot is completely defined by a session ID/timestamp in the context of a single server. All timestamps are monotonically increasing server clock times, but direct use of anonymous snapshots is seldom necessary. Thus clock synchronization is generally not important in MFS and will not be discussed further.

A *named snapshot* has an identifier, and possibly other attributes. Named snapshots are created by users, usually specify the current time, and can be mounted via the identifier. A *primordial snapshot* is a named snapshot that gives access to the "empty" file system state, a file system containing only the empty root directory. The primordial snapshot identifier is chosen by the administrator at server file store initialization time.

An *aggregate view* is a tree-structured specification that defines a set of session and snapshot mounts, together with their relative mount points in the local file system. Aggregate views can be used to automatically perform multiple mounts, possibly from different servers. Views can be used similarly to the Unix `/etc/fstab`, except that paths are relative. The view can therefore can be mounted anywhere, including on clients other than the one on which it was created. Furthermore, views can encapsulate capabilities.

Named snapshots automatically provide a root restriction feature similar to Unix `chroot` jails. When a snapshot is taken, the snapshot's directory is recorded as the snapshot's root. On subsequent mounts of the snapshot, that directory is used as root directory for the mount point. Sessions derived from a rooted snapshot inherit the snapshot's root.

2.1 Shared Sessions

By default, two clients writing to the same base snapshot create distinct sessions; writes made in one session are not visible in the other. This non-transparent replication is very different than traditional file system semantics, but is the basis for autonomous collaboration using MFS. Clients possessing the correct credentials can join existing sessions, creating *shared sessions*, which are much closer to traditional semantics. Shared sessions attempt to provide transparent replication to all session participants, but the session as a whole is still isolated from other sessions.

Consistency Shared sessions can be created in two consistency modes. The default "best-effort" mode is similar to existing file systems: updates (or invalidates) for modified files are propagated quickly to other session participants, but near-simultaneous updates by multiple clients could result in some updates being lost.

A shared session in "fork" mode differs from the above in that when the server detects unordered updates (a write arrives from a client whose cache was not up to date), the out-of-date client's view of the session is forked off of the shared session.

This raises the question of how to notify clients/users that a fork has occurred. The file system could either raise some sort of user-visible exception (as in Ficus or Coda), or rely on the user to notice through polling. Our prototype currently takes the latter approach.

Update Propagation Shared sessions also come in two flavors with respect to how aggressively updates/invalidates are propagated to session participants.

Synchronous shared sessions are intended to support closely-collaborating users. For example, two users collaborating on a new program release might sit at adjoining terminals, verbally ensuring they never work on the same file concurrently. They might both expect to see modifications made by the other immediately, or at least once a file is closed.

In a synchronous shared sessions, updates (or invalidates) generated at one client are synchronously pushed to the server, which pushes them to all other clients in a best-effort fashion.

Asynchronous sessions are intended for single users who keep a single session open on multiple machines, moving from one machine to another without ever closing or remounting the session explicitly. Movement from one machine to another is assumed to be slow relative to network speeds, and therefore updates are no longer propagated synchronously. Instead, the MFS client can take advantage of several techniques, such as aggregation, available to lazy updaters.

Consistency While local file systems can make strong consistency guarantees for local clients, systems like NFS [6] provide only best-effort consistency across networks, relying on file locking when needed. Wide-area file systems like Coda and Ficus provide "close-to-open" semantics, which insures that all updates made by a client to a single file between an open and close are seen atomically by the rest of the clients sharing the file.

We argue that neither of these approaches is appropriate for MFS because MFS needs to support consistency *between* files. In other words, consistency needs to be defined across all files covered by a session or snapshot, rather than just a single file.

The analogue to a database's transaction in MFS is a non-shared session, so we define consistency with respect to such sessions. Together with a simple merging algorithm, sessions can be seen to provide a well-studied database consistency property called *snapshot isolation* [4, 10].

2.2 Snapshot Merging

We discuss two ways of merging multiple snapshots: joining and aggregation. The former results in another snapshot, the latter in an aggregate view.

Recall that a snapshot is specific to a single directory closure (the directory and its subdirectories) on a remote server. Hence, two snapshots must pertain to the same remote directory to be joined into a single new snapshot. The desired semantics are clear when there are no conflicts: the most recent version of each file under the snapshots' directory should be chosen. Conflicting modifications to the same file are more properly resolved at the application or user level. However, determining whether there is a conflict is not always trivial.

Let S_i and S_j be the snapshots to be merged, and S_k be the first common ancestor. In Figure 2, for example, the first common ancestor of S_4 and S_5 is S_2. We say that S_i and S_j conflict if there is any file whose versions in both S_i and S_j differ from the version in S_k, where the notion of "differs" includes deletion, as well as creation of distinct files with the same name.

If S_i and S_j have no conflicts, a new joined snapshot, S_{join}, can be defined as follows. Let S_{join} initially be defined as the set of file versions from S_k. For each distinct file, replace it with the most recent modification described in either S_i or S_j (where modification includes deletion). Finally, add each new file described in either S_i or S_j.

We have written a short Perl script to perform this merging operation. The Perl script is able to determine the last common ancestor of two snapshots by querying a commandline tool called "mfs". The new snapshot is created without copying any files over the network. The script makes use of a copy primitive that allows a file to be copied from a snapshot to a session on the same server without network communication.

Building a joined snapshot in this manner is correct, but requires time proportional to the number of files covered by the snapshots. A more efficient approach can be based on marking entries in snapshot logs. The basic idea is to modify the lookup procedure of files covered by such a snapshot to pursue both paths of the fork in the snapshot ancestry (see Section 5).

Snapshot joins result in another snapshot. By contrast, an *aggregate view* consists of a document containing a set of snapshot identifiers, each with an offset into the aggregate mount. The offset specifies the name of a local directory under which the associated snapshot should be mounted. The local directory specified in the offset is relative to the mount point of the aggregate view. For example, consider the following aggregate view:

```
{snapshot-ID: 1, offset: ""}
{snapshot-ID: 2, offset: "backup"}
```

Mounting this aggregate at `/mnt/mfs` is equivalent to mounting S_1 at `/mnt/mfs` and S_2 at `/mnt/mfs/backup`. Any two snapshots can be aggregated, whether from the same or distinct servers. Aggregate views are therefore an extremely useful tool for saving, restoring, and sharing client views.

3 Naming and Searching

One consequence of structuring file system semantics around snapshots is that updates to individual files and directories are not necessarily serialized. A snapshot is only ordered with respect to the snapshot on which it is based, and with respect to snapshots based upon it. Hence, snapshots are only partially ordered, and there is no semantics-based method of determining the "last" snapshot to modify a given file or directory.

For example, snapshots S_1 and S_2 in Figure 2 are both ordered with respect to S_0, but not each other. The relationships among snapshots (hosted at a single server) can be represented as a DAG with snapshots as vertices, and directed arrows from "parent" snapshots to each of the snapshots based on them (Figure 2). Each snapshot has a single parent and an arbitrary number of children.

As a result, *naming* of snapshots becomes an issue. Snapshot IDs (even with optional comments) are relatively opaque; they do not provide either ordering information, or detailed information about which files are named by snapshots.

3.1 The Tag Namespace

MFS provides two ways to map file and directory names

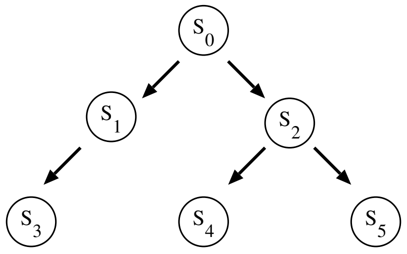

Figure 2: Snapshot relationships

back to appropriate snapshot IDs. The first method is via the *tag namespace*. The tag namespace maps arbitrary names to snapshot identifiers. Including a tag on a snapshot command allows clients to "publish" snapshot IDs under that tag. For example, snapshots of nightly builds of the Linux kernel might be published as "linux nightly". Clients can mount the last nightly build by providing the tag to the mount command.

We have taken the minimalist approach of designing a linearly versioned tag namespace, though more sophisticated organizations are also possible. However, the minimalist approach is powerful enough to implement a lightweight CVS-equivalent application on top of MFS, and to implement snapshot isolation.

Snapshot tags are publicly readable, but the overwriting of tags is controlled by checking *tag capabilities*. When tags are instantiated, both a snapshot ID and a tag capability must be provided. The instantiation succeeds unless the tag already exists. Presentation of the tag capability is required for later modifications of the tag's mapping. The orthogonality of naming and access rights means that the ability to name a snapshot does not imply permission to read the files. Capabilities for the named files must be acquired through some other method.

Though the discussion above has concentrated on snapshots, session IDs can also be stored in the tag namespace.

3.2 Snapshot Naming and Covering Paths

The second approach to identifying relevant snapshot IDs is the *covering path search*, which returns the last snapshot whose directory is a prefix of the search parameter. In this case, "last" means the snapshot that arrived at the server last. For example, suppose Alice mounts a snapshot over local directory `/home/alice`, and later creates a snapshot of the local directory hierarchy `/home/alice/work/paper`. The *covering path* for this new snapshot is `/work/paper`. The server could return this new snapshot as a covering-path result on a search for `/work/paper`, or `/work/paper/fast05`, but not for `/work`.

Our prototype currently limits the covering path method to searching for the last snapshot committed at the server, but this approach can be trivially generalized to searching for the last snapshot before a specific time, or searching for arbitrary characteristics of the snapshot. This method is convenient in that it requires no action from the user to create the naming association. By definition, however, the result is not defined under concurrent updates, and our current experience is that it is little used.

4 Using MFS

This section gives a variety of usage scenarios, and describes how MFS commands can be used to handle them. For purposes of this section, a capability is merely a short file containing identifiers (including server identifiers) and secrets that provide access to snapshot and/or sessions.

Producer-Consumer The canonical producer-consumer sharing pattern is realized in MFS by having the producer create a non-shared session, write new data, take a snapshot, and pass the snapshot and associated capability to the consumer. We have combined these four discrete steps into a shell script, allowing this sequence to be realized as follows:

```
> mfs-mount -snap 4382 /m1
     <write files>
> mfs-capa /m1 "item.capa"
```

The first line mounts snapshot 4832 over local directory /m1. Files under /m1 are then read and written through normal file operations, and then `mfs-capa` takes a new snapshot, and creates a capability providing access to the snapshot in the local file "item.capa". The producer would then mail or pass by other means the capability to the consumer. The consumer mounts the resulting data at /mnt/item as follows:

```
> mfs-mount -capa "item.capa" /mnt/item
```

Migrating User A user might wish to mount her home directory on each of several different machines (home, work, laptops, etc.). Since the user will presumably only be using a single machine at a time, she can create an asynchronous shared session on one machine and connect to it from the other machines as follows:

```
> mfs-mount -tag "a tag" /m1
> mfs-capa -sess /m1 "home"
```

The first line mounts an initial snapshot described by tag "a tag" at /m1 in the local file system. The "mfs-capa" call creates a new shared session from the snapshot mounted at /m1 if it does not already exist, creates a capability granting access to the session, and stores it in a local file called "home." On each of her other machines, she does the following:

```
> mfs-mount -capa -async  "home" /m1
```

and the resulting session is mounted at /m1 on all other machines, persisting until explicitly canceled. Once the session has been attached to in the asynchronous mode, all subsequent attachment attempts will fail unless they are also asynchronous.

Close Collaborators Close collaborators establish their sharing session as above, with a "-sync" flag instead of "-async".

Release Testing Consider an open-source software project where any member can mount and modify source, but only a single trusted user can "check in" new, presumably verified source. The software project will use two versioned tags: "test" and "release". A developer checks code out by merely by mounting the snapshot shown in the current release version:

```
> mfs-mount -tag "release" /m1
```

The developer makes local changes on the code in /m1, and uploads a new version to be validated for possible inclusion in a release by taking a snapshot of the changes and posting the snapshot ID:

```
> mfs-snap -tag "test" /m1
```

The gatekeeper periodically peruses all submitted code by checking all snapshots written to the "test" tag since the last check, chooses some modifications to include in the next release, and does the following:

```
> mfs-mount -tag "release" /m1
    <apply selected changes>
> mfs-snap -tag "release" /m1
```

This only works if write access to "release" is protected by a write capability held only by the gatekeeper, and "test" is not protected by a write tag at all. Tags are versioned, with each new version being assigned a monotonically increasing integer. The current version of a tag is retrieved by:

```
> mfs-tag -top <tag name>
```

The value of a specific version a tag is retrieved as follows:

```
> mfs-tag [-version n] <tag name>
```

where n is the requested version of the tag. The query defaults to the current version if the version argument is not supplied.

5 MFS Prototype

Figure 3 shows the structure of our prototype. The prototype is approximately 27,000 lines of C, and includes all of the functionality discussed in this paper. The prototype consists of two multi-threaded user-level daemons, one running on the server, and one running on the client. The MFS server daemon serves data from the underlying file system (ext3 in this case). Data is relayed via XDR [38] to the client machine through the MFS client daemon, which talks to the client kernel by posing as an NFS version 3 [6] server, similar to the operation of the sfstoolkit [26]. Our implementation has only been tested on GNU/Linux, but we expect that the lack of a kernel component will simplify ports to other systems.

Locating the MFS client on the client host is not strictly necessary, but improves performance by caching data locally. However, much of this benefit is achievable in the wide area by locating the MFS client "near" the client machine in network latency. This approach allows the MFS client to serve architectures and OS's that we do not currently support.

The MFS client uses a disk-based cache to improve performance. This cache operates in write back mode: writes are first logged to an operational log on disk. By default, writes are sent to the server asynchronously via periodic flushes of log blocks. This raises the possibility of a snapshot token being presented to a server before it has been flushed by the client. However, users can explicitly flush the log if they wish others to use a snapshot immediately. The log is always flushed synchronously after each write for shared sessions.

Snapshot creation is lightweight, in that it involves little more than appending an appropriate entry to the log. Snapshots are, therefore, either fully materialized at the server, i.e., all preceding writes have already arrived at the server, or the server does not know anything about that snapshot. This approach ensures that snapshots appear atomic and consistent to clients.

MFS uses the rsync library [41] to create *delta encodings* of new versions of files that are to be written back to the server. These delta encodings only include new data, and hence are often smaller than a file's size. We currently only perform delta encodings against previous versions of the same file, but more sophisticated approaches are possible (e.g., LBFS [29]).

The MFS server consists of a small set of tables (one each for snapshots, open sessions, mount points, etc.) and a set of logs, one for each file and directory.

A read operation on a file mounted from a snapshot may be thought of as a search for a query consisting of the file's inode, the server time when the snapshot was taken, and the snapshot's ID. The inode identifies the per-file log, which contains a record of all modifications to the file sorted by server time. We use a binary search to get close to the last relevant record, followed by a linear search backward until we find the last version created before the target time in an acceptable context. An acceptable context is either the query's snapshot, or one if its ancestors. The number of entries that have to be scanned linearly is likely to be small, assuming that the number of concurrent writes to the same file inode on different snapshots is small.

We do not currently garbage-collect server logs, but the mechanisms for doing so are straightforward because the primitive snapshot mechanism only names state on a single server. Defining appropriate policies, however, is more difficult [36].

6 Performance

We designed MFS to provide both new functionality and better performance than existing systems in the wide area. These two goals conceivably conflict, but we provide both micro-benchmarks and application benchmarks to show that

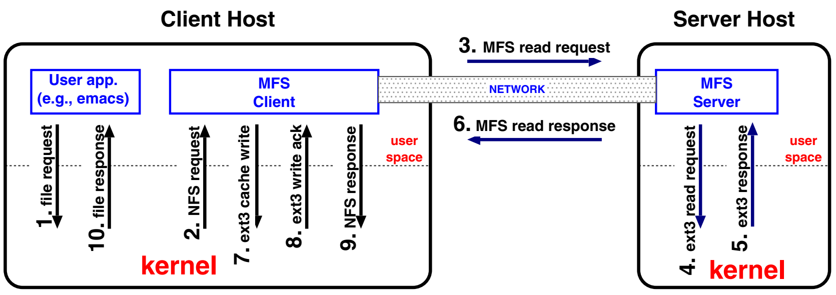

Figure 3: System structure, and anatomy of a read miss

the new features do not come at a significant cost. On the contrary, MFS performs quite well compared to a representative local-area file system, NFS version 3 running on top of TCP, and a representative wide-area file system, Coda.

We used Coda 6.0.8, configured with a 90MB file for RVM data, a 22MB file for the RVM log, and 256,000 inodes. We present results for Coda's client operating in each of the strongly connected (Coda-sc) and write-disconnected (Coda-wd) modes. The latter mode disables all writebacks until connectivity improves. Coda uses UDP for its traffic, and does not offer a TCP option.

File systems differ on whether they require the server to wait until a write has reached stable storage before returning an acknowledgment. Waiting is the default for most file systems, but we test MFS and NFS variants with both approaches. System names with a "-a" suffix refers to the asynchronous approach, where the server responds immediately rather than waiting on stable storage. A "-nd" suffix signifies that delta encoding of writebacks has been turned off. Delta encoding is generally a disadvantage in the local area due to CPU overhead, but can be beneficial when communication costs increase.

We conducted all our experiments on a pair of identical machines, each with two Athlon MP 1.66GHz processors, 1GB of RAM, a 7200RPM disk, and running the 2.6.10 Linux kernel. The disk transfer rate as measured by hdparm peaks at 46MB/second on these drives.

The machines were connected by a 100Mb Ethernet switch. We used the Linux traffic control tool tc to simulate ADSL-like bandwidth (768 Kb downstream and 128 Kb upstream), and wide-area latencies (100 msec round trip, but with the full 100 Mb bandwidth).

Finally, MFS can use TLS to provide message integrity and encryption, though neither is needed for correct operation. However, we disabled both for message payloads, using TLS only to establish a shared key used to encrypt capability secrets.

We ran an extensive set of LAN micro-benchmarks to determine the overhead of the MFS abstractions. Although space limitations do not let us present the results here, our results indicate that the overhead of the abstractions is minimal. The primary overhead is due to MFS's current implementation as a user-level server.

We also conducted a series of application-level benchmarks designed to capture typical workloads of software development. The test sequence is loosely inspired by the Andrew benchmark, but we test some aspects not covered in that suite. All numbers represent file systems in their synchronous modes, i.e., writes are committed to stable storage before acknowledgments are sent.

The sequence is centered around a hypothetical user who downloads and builds the source to a program (MFS in this case). The user first extracts source from a gzip-compressed tar file (Table 1), and then builds the binary, taking advantage of the cached sources (Table 2). A patch derived from an actual CVS commit is then applied against the sources (Table 3), and the binary is re-built (Table 4). Finally, we perform a full build with cold caches (Table 5). A table entry of a dash ("-") means that the column is not applicable for that system.

The four systems we discuss are MFS, both versions of Coda, and NFS. All four are configured to commit writes to stable storage before returning acknowledgments to clients. For each of the three environments ("LAN", "ADSL", and "WAN), we show both the user-observable time ("user"), and the time required to flush the data back to the server ("flush"). We have split the synchronous and asynchronous components of system overhead to gain a better understanding of system performance. However, the two may overlap considerably in real situations.

Our first finding is that local caching is very important. This is somewhat obvious, but we were still surprised when the two wide-area systems, MFS and Coda-wd, performed better than NFS even in the LAN environment. NFS sends data directly from the server kernel to the local kernel. This efficiency is evidently more than offset by the gains from having a disk-based cache at the client. In the ADSL and WAN environments, however, the differences were much more striking: up to two orders of magnitude in some cases.

The second major finding is that MFS is much less latency-sensitive that Coda. MFS and Coda-wd have similar `user` numbers for most of the tests. However, they differ significantly in the `flush` numbers, especially in the WAN environment. Considering all of the experiments except the patch (which has little data to flush), MFS's flushes always take less that one fifteenth the time of Coda-wd's.

Part of this gain in performance is undoubtedly due to optimizations that could be used in either system. For exam-

	LAN		ADSL		WAN	
	user	flush	user	flush	user	flush
NFS	2.0	-	66.8	-	140.6	-
Coda-sc	2.3	-	65.4	-	66.9	-
Coda-wd	0.4	1.6	0.4	74.1	0.6	38.0
MFS	0.2	0.1	0.3	52.5	1.1	1.9

Table 1: `untar`, cold cache (seconds))

	LAN		ADSL		WAN	
	user	flush	user	flush	user	flush
NFS	12.9	-	118.6	-	283.6	-
Coda-sc	13.1	-	74.7	-	36.3	-
Coda-wd	12.2	1.3	12.2	78.8	12.2	31.3
MFS	12.4	0.1	12.4	64.1	12.5	1.8

Table 2: Full build, warm cache (seconds)

	LAN		ADSL		WAN	
	user	flush	user	flush	user	flush
NFS	0.14	-	10.07	-	13.78	-
Coda-sc	0.20	-	9.40	-	3.92	-
Coda-wd	0.04	0.16	0.04	9.45	0.04	3.25
MFS	0.04	0.03	0.04	0.50	0.04	0.27

Table 3: Apply source patch, warm cache (seconds)

	LAN		ADSL		WAN	
	user	flush	user	flush	user	flush
NFS	13.0	-	116.6	-	292.1	-
Coda-sc	13.0	-	72.1	-	39.0	-
Coda-wd	12.3	0.9	12.3	77.1	12.2	29.1
MFS	12.5	0.3	12.5	12.3	12.5	0.9

Table 4: Incremental build, warm cache (seconds)

ple, MFS aggressively prefetches the attributes of all files in a directory when one is requested. Coda-wd is also fast because it does not actually re-validate *any* files in their cache, at least for the examples that we have instrumented. As another example, MFS clients choose inode numbers for new files locally, rather than requesting them from the server.

However, a great deal of this performance gain is due to optimizations directly enabled by MFS's abstractions. Mounting only immutable snapshots allows the protocol to assume single-writer semantics. While other protocols have to check for, and handle, conflicts and race conditions, single-writer semantics allow MFS to use aggressive pipelining and selective acknowledgments. Both techniques allow bandwidth to be used more efficiently, but are especially effective in high-latency environments.

Finally, our results do not include evaluations of the times required for verifying or delegating credentials. Though the MFS security architecture is extensive and admits a rich set of functions (especially using watermarks), all of the functions are implemented using extremely efficient HMACs. Further, the HMACs only need to be computed once per mount (when the client's access to the requested snapshot is authorized). The time required to compute the cryptographic hashes one time is entirely negligible compared to even the latency on

	LAN		ADSL		WAN	
	user	flush	user	flush	user	flush
NFS	13.0	-	129.5	-	291.9	-
Coda-sc	13.2	-	81.5	-	58.0	-
Coda-wd	12.6	1.3	18.9	79.1	33.5	31.2
MFS	12.5	0.1	18.9	64.3	24.6	2.1

Table 5: Full build, cold cache (seconds)

the LAN.

7 Related Work

Version-control systems Basing a file system on immutable snapshots is similar to version control systems such as CVS [5], subversion [3], or ClearCase [7]. The latter is even based on snapshots, but subversion snapshots describe the entire state of the repository, and are totally ordered. Vesta [18] includes the ability to model the entire build environment, and to recreate any build at a future time. MFS's snapshots are significantly more lightweight and flexible, and as we show in Section 6, integrating snapshots into the file system admits several performance optimizations. Further, the security architecture in MFS significantly reduces administrative overhead compared to these version control systems.

File Systems MFS differs from prior file systems in basing the user experience around snapshots. Distributed file systems [20, 23, 31, 8, 35] generally try to make the distribution transparent through caching and relaxed consistency. These systems usually support close-to-open consistency, but make no guarantees across files. MFS also caches aggressively, but uses snapshots to ensure that mutual consistency is guaranteed at all times.

Finally, the MFS file store is similar to versioning file systems such as Elephant [36], Cedar [14], and 3DFS [34]. Unlike in these systems, however, MFS file versions are not strictly ordered and version histories may fork. Further, Elephant does not have a concept corresponding to MFS snapshots, although related files can be associated together in a way that causes the garbage collecting algorithms to treat them as a single unit.

Felix [11] supports serializable transactions on file sets using two-phase locking. In this respect it has database-like features, including the possibility of aborted transactions due to deadlocks. Felix versions files linearly.

Distributed security and authentication Security is the focus of much work in the field of mobile and distributed file systems, from work on staging data at untrusted surrogates [37], to using untrusted servers [25, 21], to work using agreement protocols to deal with Byzantine failures [1, 24]. S4 uses such a versioning subsystem to prevent data loss with untrusted clients.[39, 13]

Most systems perform access control by first authenticating users through trusted certificate authorities [27, 43, 19]. SFS [27] uses self-certifying pathnames to authenticate servers, and trusted certification authorities to authenticate users via public keys. Kerberos [30] is a centralized, shared-key system that allows access by remote users only through inter-realm authentication, where the realms must have pre-existing reciprocity. Kaminsky [22] allows ACLs to contain chains of indirection, periodically prefetching remote credentials to allow faster authentication when users connect.

Many file systems have used capabilities of various sorts. Felix [11] and Swallow [32] use randomly-generated file IDs as capabilities. Felix also supports *file sets*, a means of atomically committing updates to an arbitrary set of files at once, though each file must have its own distinct capability.

Cryptographically-protected capabilities were introduced in Amoeba's Bullet file system [42], and were concerned only with authorization. Rights could be delegated and narrowed without contacting the server by relying on properties of commutative hash functions.

It was later observed by Gong [15] that Amoeba's solution lacked the ability to control the set of principals that a capability may be given to. Gong introduced an identity-based capability system [16] by including the user identity in the hash calculation. Subsequent file systems using cryptographic capabilities offered this feature, at least as option.

CapaFS [33] and DisCFS [28] are two recent systems based on cryptographic capabilities. CapaFS can encrypt access rights, and the intended recipient's public key, within the file name. Delegation and narrowing is achieved by appending to the name new encrypted extensions. Soft links are used on the clients to map server files into user defined names. This is clearly cumbersome when many files are involved. In contrast, MFS does not overload file names with capabilities. CapaFS uses RSA encryption, which imposes an overhead of one to two orders of magnitude compared to NFS on a 100Mbit LAN.

DisCFS uses certificate chains based on KeyNote, which allow more elaborate graph-based inferences of delegation when compared to CapaFS. DisCFS avoids some of the high costs of asymmetric cryptography and the logic inference engine by assuming trusted clients, and only verifying a few operations. MFS does not make this assumption.

Capabilities in the systems discussed above are used to protect individual files or directories. MFS has more expressive capabilities, allowing both spatially and temporally related groups of files to be described by a single capability. A major focus of research in access control has been grouping suitable rights into named protection domains [2] (or roles), which are in turn granted to users. MFS may be viewed as an attempt to automatically provide such groups (albeit without naming) by exploring the rich structure of a versioning file system.

8 Summary

This paper has described the design, implementation, and performance of MFS, a new file system supporting collaboration in the wide area. MFS differs from prior work in its support for both transparent and non-transparent replication, its elevation of sessions and snapshots to first-class objects, and its ability to support mutual dependences in sets of files via snapshot isolation.

The second primary innovation is the use of a layered security architecture, which allows a range of policies to be specified. The base system uses capability-based access control to allow sharing without cumbersome identity checks and account management. Capabilities can grant access to single snapshots, or to a range. Capabilities can be duplicated, watermarked, and weakened by clients without communication with the corresponding servers. "Watermarking" adds indelible labels, allowing several optional services, including audit trails, write quotas, and fine-grained revocation. Finally, capabilities could be managed by a translation layer to allow more conventional user authentication, either with passwords or public keys.

Far from imposing performance costs, these abstractions enable several performance optimizations that allow snapshot-based systems to compete with highly optimized commercial file systems in the local area, and outperform other systems in the wide area.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, 2002.

[2] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *IEEE Symposium on Research in Security and Privacy*, pages 116–132, 1990.

[3] B. Behlendorf, C. M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman. Subversion project homepage, 2003.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM Press.

[5] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[6] B. Callaghan and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, June 1995.

[7] Web. http://www.ibm.com/software/awdtools/clearcase/.

[8] R. C. Davis, J. A. Landay, V. Chen, J. Huang, and R. B. Lee. Notepals: Light weight note sharing by the group, for the group. In *Proceedings of CHI 1999*, pages 338–345, 1999.

[9] W. K. Edwards and E. D. Mynatt. Timewarp: Techniques for autonomous collaboration. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI'97)*, pages 218–225, 1997.

[10] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, 2004.

[11] M. Fridrich and W. Older. The Felix file server. In *Proceedings of 8th ACM Symposium on Operating Systems Principles*, pages 37–44, 1981.

[12] V. Gaburici and P. Keleher. Distributed file systems, capabilities, and motefs. *In preparation*, 2006.

[13] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture.

In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 92–103, 1998.

[14] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the Association for Computing Machinery*, 31(3):288–298, 1988.

[15] L. Gong. On security in capability-based systems. *Operating Systems Review*, 23(2):56–60, 1989.

[16] L. Gong. A secure identity-based capability system. In *IEEE Symposium on Security and Privacy*, pages 56–65, 1989.

[17] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[18] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management. Technical Report 168, Compaq Systems Research Center, Mar. 2001.

[19] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and Consistency Tradeoffs in the Echo Distributed File System. In *The 2nd Workshop of Workstation Operating Systems*, pages 49–54, Sept. 1989.

[20] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, 1988.

[21] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[22] M. Kaminsky, G. Savvides, D. Mazieres, and M. Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[23] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[24] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of ASPLOS*, 2000.

[25] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI-04)*, 2004.

[26] D. Mazieres. A toolkit for user-level file systems. In *In Proc. Usenix Technical Conference*, pages 261–274, 2001.

[27] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 124–139, Dec. 1999.

[28] S. Miltchev, V. Prevelakis, S. Ioannidis, J. Ioannidis, A. D. Keromytis, and J. M. Smith. Secure and flexible global file sharing. In *Proceedings of the USENIX 2003 Annual Technical Conference*, 2003.

[29] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, October 2001.

[30] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9), Sept. 1994.

[31] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software–Practice and Experience*, 28:123–133, 1998.

[32] D. P. Reed and L. Svobodova. SWALLOW: a distributed data storage system for a local network. In *International Workshop on Local Networks*, Zurich, Switzerland, 1981.

[33] J. Regan and C. Jensen. Capability file names: Separating authorisation from user management in an internet file system. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[34] W. D. Roome. 3DFS: A time-oriented file server. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 405–418, Berkeley, CA, USA, Jan. 1991. Usenix Association.

[35] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[36] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 110–123, Dec. 1999.

[37] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, , E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Data staging on untrusted surrogates. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[38] R. Srinivasan. XDR: External data representation standard. RFC 1832, Aug. 1995.

[39] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 165–180, Berkeley, CA, Oct. 23–25 2000. The USENIX Association.

[40] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS '94: Proceedings of the third international conference on on Parallel and distributed information systems*, pages 140–150, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[41] P. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1996.

[42] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The design of a high-performance file server. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS)*, pages 22–27, Washington, DC, 1989. IEEE Computer Society.

[43] M. A. E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269, Systems Research Center SRC, DEC, Dec. 1993. ACM SIGOPS, ACM Press.