PersiFS: A Continuously Versioned Network File System

Austin T. Clements, Dan R. K. Ports, Ben A. Schmeckpeper, Hector Yuen
{aclements, drkp, bschmeck, hyz}@mit.edu

April 22, 2005

Abstract

Most file systems are ephemeral*, meaning that once a change has been made, there is no way to recall the previous contents of the file system. Backups, version control systems, and user interface improvements such as "trash cans" attempt to alleviate this problem; however, these are all rough approximations of* persistent *file system structures, giving users restricted access to a restricted set of past states of the file system.* PersiFS *is a* fully persistent *or* continuously versioned *file system, allowing access to* any *past state of the entire file system. Furthermore,* PersiFS *exposes the file system and its full set of past states via a conventional file system interface, allowing unmodified applications to access the past states of the file system.*

1 Introduction

PersiFS is a network file system which remembers all changes made to every file and allows the user to recall any and all modifications to any file, including deletions. Time stamps are used to select a view of the file system's past state. A user can specify a time in the past at which the file was correct and *PersiFS* will provide read-only access to that past version of the file. *PersiFS*'s time stamp interface provides users with a natural way to express accesses to past versions of files.

A *version-controlled file system* lets the user access not just the current state of the file system but previous states as well. For example, backups are one typical, restricted form of a version-controlled file system, allowing high-latency access to a highly restricted set of past copies of a file system.

PersiFS is a *continuously* versioned file system. A continuously version-controlled file system allows access to the complete state of the file system at *any* point in the past. Incorporating continuous versioning into the file system means that a file can never be lost and mistaken modifications can always be undone. In *PersiFS*, any change to a file forces the file system to archive a copy of that file, indexed by a time stamp.

1.1 Motivation

Users frequently wish to undo changes they have made, whether intentionally or inadvertently, to the file system — for example, inadvertent deletions, restoring files corrupted by application bugs, or simply reverting to an earlier revision of a document. Regular file systems do not support this operation natively, which results in a number of tools that address parts of this problem in different ways. Some operating systems provide a "trash can" interface to help users avoid mistaken deletions of files. However, this is only a partial solution because it only addresses the problem of deletions, and even then only until the trash can is emptied and the files are permanently removed. The proliferation of "undelete" tools for administrators suggests that this solution is inadequate. *PersiFS* makes these tools unnecessary, since files are never truly deleted from the file system. Fears of accidental overwrite, rename or deletion are unnecessary.

Versioning is a natural and desirable aspect of file systems. Often, critical files are versioned through the use of version control systems. However, these must be explicitly configured, maintained, and interacted with. Providing such support at the file system level gives the ability to easily track changes to *any* file over time. To eliminate the need for user interaction and the possibility of user error, systems exist which automatically create periodic snapshots (either of just critical system files, or of entire file systems). These, however, suffer from problems of time aliasing, failing to capture multiple changes made between snapshots and possibly creating inconsistent snapshots at inopportune times. By capturing every revision, *PersiFS* cannot suffer from time aliasing.

Because *PersiFS* is a network-accessible file system, it is ideal for multi-user systems. Such systems typically use some form of snapshotting to provide all users of the system with security for their files and to avoid administrative nightmares with recovering from user errors. By using a continuously versioned file system like *PersiFS*, system administrators can can provide users with an easy way to recover from a much wider range of problems.

1.2 Features

PersiFS's main contribution is a novel file system interface that allows users and applications to access the file system and its past states without any modifications to applications or the need for any special libraries. As a networked file system, *PersiFS* allows remote access to both current and archived files for multiple users. *PersiFS* is a *continuously versioned* file system, so it is capable of exposing *all* past states, unlike *snapshotting* file systems that only save states at periodic intervals. Users can access the entire file system tree as it existed at a particular time simply by accessing the automatically mounted directory `/persifs/servername@timestamp`.

PersiFS provides durable storage by storing all data on disk on a central server. For reliability and ease of implementation, the *PersiFS* data structures are stored on a normal file system, though with minor modifications it could operate on a physical disk as well. Each client system accesses the server via an NFS automount in the `/persifs` directory hierarchy.

2 Related Work

Many other systems provide some form of persistent versioned storage. *PersiFS* differs from these systems in that it provides access to all previous versions of the file system's state via a standard file system interface.

2.1 Version Control Systems

Version control systems such as CVS [1] are the standard mechanism for tracking revision histories in large projects. While *PersiFS* provides similar revision history operations, it does not provide the higher-level semantics for synchronizing the work of multiple users, such as file locking as in RCS [14] or merging as in CVS. *PersiFS* provides only revision history support; it does not provide the higher-level functionality because the appropriate semantics are dependent on the type of file and its mode of use (e.g. text vs. binary files), and so should not be applied at the file system level.

CVS organizes revision histories by assigning a version number to each revision of a particular file. This is a natural interface for the history of a particular file, but does not generalize well to tracking the history of the entire file system. Many of the weaknesses of CVS as a version control system are due to this interface: it does not capture the notion of changes that occur simultaneously to multiple files (changesets), and it does not effectively handle changes to the directory structure, such as moving or renaming a file. The latter is a major problem for a file system, since the directory structure can be highly dynamic.

Subversion [2] addresses many of the short-

comings of CVS by using a single global revision number that identifies a particular state of the entire repository. This is similar to *PersiFS*'s internal representation; however, the user interface uses date/timestamps instead because global revision numbers do not generally correspond to a useful identifier from the user's perspective.

2.2 Snapshots vs. Continuous Versioning

Previous states of the file system are commonly stored by a backup system that periodically archives the state of the filesystem. *Snapshot*-based version-controlled file systems are the natural extension of this idea: they periodically take a snapshot of the state of the filesystem, and make it readily available. The Plan 9 file server [9], for example, creates daily snapshots of the filesystem and stores them on a write-once mass-storage system such as a WORM jukebox or the Venti archival server [10]. WAFL [3] provides similar snapshotting functionality using copy-on-write disk blocks. AFS provides access to the most recent snapshot through the `OldFiles` mechanism.

While a great improvement over a standard ephemeral file system, snapshotting filesystems have the obvious disadvantage that they cannot track changes that occur between revisions. *PersiFS* is a *continuously* version-controlled file system: each change to the file system is stored as a new revision. Elephant [12] and CVFS [13] also use this technique. However, *PersiFS* provides a much more convenient interface for users.

2.3 Interfaces

Some versioned file systems require special tools to access previous versions of the file system. For example, CVFS is designed for performing post-intrusion forensic analysis, so it does not provide an interface for users to easily recover old versions of their files. A convenient way to provide access to old versions is via a filesystem interface. Plan 9 creates a directory hierarchy of snapshots; Pike et al. [9] report that providing access to snapshots via this interface is very convenient for users.

Ideally, it should be possible to interact with the versioned file system exclusively through the file system interface, such that unmodified standard UNIX utilities (`ls`, `cp`, etc.) suffice for revision control. This is not possible in Elephant, which adds new system calls. VersionFS [7] allows unmodified utilities to be used, but only via a library wrapper and the `LD_PRELOAD` mechanism. *PersiFS* uses a purely file-system-based interface, using an automounter based on that of SFS [4], allowing standard utilities to be used without any modification.

3 Design

3.1 User Interface

PersiFS provides access to both current and previous versions of the file system state via a standard file system interface. Volumes are accessed across the network using an automounter daemon in essentially the same way as SFS [4]: a `/persifs` directory is created, and the current version of the file system on server `foo` is accessed via `/persifs/foo`.

The automounter also provides access to previous versions of the file system, by combining the server name with a timestamp, such as `/persifs/foo@20050413120000`. This gives a read-only snapshot of the file system on `foo` as it appeared at noon on April 13$^{\text{th}}$, 2005.

The user interface may also include a number of small tools that improve the usability of *PersiFS*. For example,

- `persifs pwd` *datespec*
  Print the path to the current directory at the time specified by *datespec*. Datespec should allow a wide range of time stamp formats, including relative ones as supported by CVS.

- `persifs log` *file*
  Display information about when *file* was

modified. This will require a special interface to the server to operate efficiently.

The administrator of a *PersiFS* server should be able to set retention policy, deleting data when it becomes older than a pre-specified time, or merging together revisions that occur within a certain time interval. This may be relegated to the domain of future work depending on time and sanity constraints.

3.2 Client-Server Interface

All data in *PersiFS* is stored on a server, which contains the file system structure and manages the versioning. Clients interact with the server using a RPC-based protocol similar in spirit but somewhat simpler than that of NFS [11], with support for versioned operations.

Each client runs a user-level *PersiFS*client that exposes *PersiFS* under the `/persifs` directory. It is implemented using the SFS user-level file system toolkit [5], minimizing the need to change the behavior of the OS or the applications. The client acts as an automounter, translating timestamps into revision numbers and mounting directories under `/persifs` as necessary. It then forwards requests to the appropriate *PersiFS* server.

3.3 Internal Representation

The file system's internal representation divides into two main components: the *superblob* and the *inode log*. To improve system performance, these are further augmented by the *blob index* and the *inode map*. All of these structures except the inode map have efficient on-disk representations and the inode map is treated exclusively as soft state kept in memory.

Like traditional Unix file systems, *PersiFS* uses inodes identified with unique inumbers to store file and directory metadata. Unlike most file systems, however, inumbers are never recycled. While not strictly necessary, this eases inode generation and makes sense in a system in which inodes are never ultimately removed. Furthermore, this eases exportation over NFS by eliminating the need for generation numbers in order to uniquify file handles. Also unlike most file systems, *PersiFS* inumbers have no correlation to disk addresses.

Files in *PersiFS* consist of an inode and a sequence of *chunks*, which, concatenated, form the contents of the file. As in most Unix file systems, directories are simply files with a special directory flag and a particular, internal format, and, as such, are stored in the same manner as files.

The superblob is an append-only vector of all file content chunks. Whenever a file is written to, the modified chunks are appended to the superblob and the file's inode is updated to point to these chunks. In order to optimize the space used by the superblob, we apply two additional tricks: *chunk fusion* and *content-sensitive chunking*. Chunk fusion allows us to coalesce multiple chunks with the same content into one chunk, thus avoiding redundant storage costs. This is combined with content-sensitive chunking, which places chunk boundaries based on the contents of files such that modifications, including insertions and deletions of content, only affect chunks local to those modifications.

Every chunk has a fingerprint, which is simply the 160-bit SHA1 of its contents. The blob index maps chunk fingerprints to the locations of those chunks in the superblob. Whenever a chunk is added to the superblob, the blob index is first checked for that chunk's fingerprint. If the fingerprint is found, the chunk being inserted can be fused with the existing chunk in the superblob, requiring no additional space. Otherwise, it is appended to the superblob and indexed. This is similar to the approach employed by Venti [10] for archival block storage. While the blob index could be recomputed by reading the entire superblob, this would lead to prohibitively long recovery, so the blob index is maintained on disk.

The effectiveness of chunk fusion is further improved by content-sensitive chunking of file

contents. This technique places chunk boundaries based on file contents, instead of at regular intervals, such that local modifications to file contents, including insertions and deletions, only affect chunks in that region of the file. We use the same Rabin fingerprint-based algorithm as LBFS [8] uses for content-sensitive chunking. Combined with content fusion, this allows us to efficiently store local modifications to files, sharing most file contents between versions of that file. Furthermore, identical content in multiple files (for instance, due to file copying) will consume very little space beyond the single copy.

While the superblob stores the contents of the file system, the structure of the file system is maintained by the inode log. Every time an inode is modified or created, the inode is stored in the superblob and an update entry for the inode is appended to the inode log. Specifically, the inode log contains entries of the form ⟨timestamp, inumber, blob address of inode⟩. When an inode is no longer needed, a deletion entry for that inode is appended to the log.

The mapping of inumbers to blob addresses at a particular point in time is an inode map. There exists an inode map for every point in the file system's history. The inode map for the current state of the file system is always kept in memory and inode maps for past versions may be cached. Because the inode log is append-only, the inode map can be reconstructed for any point in the file system's history by simply replaying the log up to the appropriate timestamp; however, since the time required to do this may be prohibitive, we periodically store a snapshot of the entire inode map in the log. Thus, in order to reconstruct the inode map for any version of the file system, the last snapshot preceding that point in time can be loaded and the log replayed up to the desired point.

4 Implementation

The main aspects of the implementation of *PersiFS* are described in this section. A logical subdivision of the modules involved in *PersiFS* is described in Figure 1. In particular, the lowest levels of the implementation are modules corresponding to the information structures represented on disk, the *inode log*, *superblob*, and *blob index*. Layered atop this are *file* and *directory* modules, which provide an abstraction based on the standard file system concepts. Finally, interaction with users is handled by a NFS interface, including the automounter.

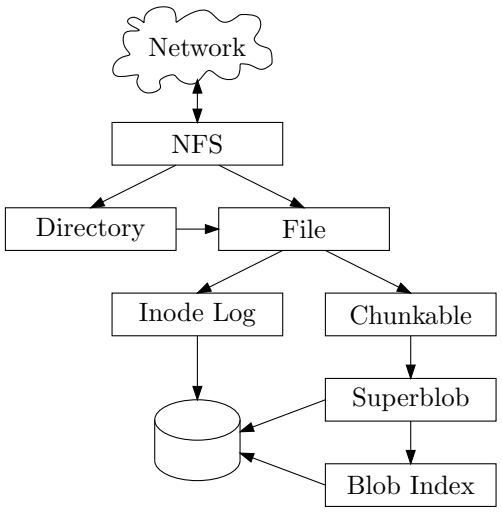

Figure 1: Implementation model

4.1 Representation Management

Each of the three information structures is stored on disk, and has a corresponding module that provides an interface to the higher levels of the *PersiFS* implementation. For ease of implementation, our initial implementation stores each structure as a separate file on a standard file system, though it would be reasonably straightforward to adapt it for direct storage on a physical disk.

The Superblob module implements content-addressable storage, providing a standard `get/put-block` interface that uses the hash fingerprint of the block to identify it. It makes use of the Blob Index module, which uses an on-disk B^+-tree to map block fingerprints to indexes into

the superblob structure on disk.

The Inode Log module stores variable-length inodes in a time-indexed log. Each inode contains the inode number, the file metadata, and the list of chunks that make up the file content (which is the marshalled form of the chunkable string containing the file content, as described in Section 4.2). The inode log interface allows new or modified inodes to be stored in the log, and supports scanning the log to obtain the inode contents at any point in the past as well as its current state. It also provides the interface for generating new unique inode numbers.

4.2 File System Abstraction

To simplify the implementation, a file system abstraction consisting of *file* and *directory* representations is built atop the persistent data structures. Using these abstractions also makes it feasible to experiment with different on-disk representations of the file system data structures.

The File module bundles together a file's inode metadata and its content, stored as chunks chunks. The File module API defines meaningful file operations, such as `create`, `read`, `write` and `getAttr`.

The contents of a file are represented by a mutable *chunkable string* backed by the superblob and aware of the LBFS-style content-sensitive chunking. The Chunkable module provides a substring-read and substring-write interface. As file contents are large, the full contents of the string are not read from the superblob until necessary. Multiple batched changes to the chunkable string can be made without being committed to disk; when a `flush` operation is performed, the chunk boundaries are recalculated and new chunks are added to the superblob as necessary. The chunkable string can be converted to a marshalled representation suitable for storing in inodes that lists all of the chunks contained in a file and their offsets in the superblob.

The Directory module provides the standard directory operations of accessing or modifying the list of files in the directory. Each directory is stored as a file whose content happens to be directory entries, distinguished from other files through special flags.

Ideally, multiple related changes to the file would be aggregated and committed as a whole only when the file is closed, in order to avoid inconsistent states. Often one write from the user is in fact broken into a series of writes, and forcing the filesystem to create versions for each 'sub-write' would result in unnecessary and logically incorrect versions. Since NFS v3 does not provide file closure notification to the server, the File module attempts to approximate these semantics by grouping together successive writes (in a span of about a second) to a single file.

4.3 Network Interface

An NFS server provides the external network interface. Clients access files on the *PersiFS* server using standard NFS operations by mounting a volume whose name contains an encoding of the desired revision information. NFS requests are handled by a server similar in structure to that of CCFS [6], which translates them to operations on the versioned file and directory abstractions above.

Revisions are accessed via the modified SFS automounter on the client side. Whenever an archived version of a file is referenced, the archived directory is automatically mounted and the file read using typical NFS protocols. Once the automounter has executed, the archived file system behaves exactly like an ordinary file system.

References

[1] P. Cederqvist, editor. *Version Management with CVS*. Free Software Foundation, 2005. Available at `https://www.cvshome.org/docs/manual/`.

[2] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subver-*

sion. O'Reilly Media, 2004. Available at `http://svnbook.red-bean.com`.

[3] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[4] D. Mazieres. *Self-certifying File System*. PhD thesis, Massachusetts Institute of Technology, May 2000.

[5] D. Mazieres. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, pages 261–274, June 2001.

[6] R. Morris and A. Muthitacharoen. CCFS: 6.824 labs, 2005. Available at `http://pdos.lcs.mit.edu/6.824/labs/index.html`.

[7] K.-K. Muniswamy-Reddy. VERSIONFS: A versitile and user-oriented versioning file system. Master's thesis, December 2003.

[8] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[9] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[10] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey,CA, 2002.

[11] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[12] 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 *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[13] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems, 2003.

[14] W. F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985.