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

May 12, 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 *file system, providing access to* any *past state of the entire file system.* PersiFS *achieves full persistence without sacrificing access time to either current versions or past versions, using inordinate amounts of disk space, or requiring modification to existing applications.*

1 Introduction

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 network 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. *PersiFS*'s time stamp interface provides users with a natural way to express accesses to past versions of files.

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 and automatically 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, may fail to capture multiple changes made between snapshots and can create inconsistent snapshots at inopportune times. By capturing every revision, *PersiFS* does not suffer from these problems.

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 Challenges

The primary challenges to such a system lie in not only providing access to past versions, but doing so reasonably fast; efficiently utilizing disk space; providing access and modification to the current version at speeds comparable to non-versioned file systems; and achieving all of this without requiring modifications to existing applications.

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. Because the server must be able to retrieve any past version of any file from durably storage, *PersiFS* introduces many optimizations in order to achieve an efficient durable representation. As further discussed in Section 3.2, it uses a compact metadata log to track changes over time and an underlying content store that efficiently fuses common data between files and versions. These structures are designed to optimize access to the current version so it is nearly indistinguishable from accesses a regular file system. Access time to past versions is less critical than to the current version; accordingly, the *PersiFS* structures provide reasonably fast recall of past versions, but do not optimize for it.

PersiFS exposes 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. For improved interoperability, *PersiFS* exposes this file system interface over an unmodified NFS protocol by providing a special root directory containing automatically generated subdirectories for every point in time. Section 3.1 discusses this interface further. While the NFS interface allows unmodified applications and operating systems to interface with *PersiFS*, it introduces further complications to its design in order to accommodate the quirks of NFS. Primarily, the statelessness of certain aspects of the NFS protocol directly conflicts with the persistence goals of *PersiFS*, as further discussed in Section 4.2.

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 [10] 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 shortcomings 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 [6], 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 [7]. 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 [8] and CVFS [9] 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. [6] 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 [4] 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 to allow 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 interface: *PersiFS* is mounted over NFS, say as `/persifs`, and the current version of the file system is exposed as `/persifs/now`.

The automounter provides access to previous versions of the file system using timestamps as names, such as `/persifs/2005-04-13-12-00-00`. This gives a read-only snapshot of the file system as it appeared at noon on April 13$^{\text{th}}$, 2005.

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

- `persifs now`
  Prints the archive path to the current directory as of the current time.

- `persifs info`
  Prints statistics about the server.

- `persifs log` *file*
  Displays a log of the modification times for *file*.

3.2 File System Structure

A trivial representation for a file system that achieves persistence is a log of every operation ever performed on the file system. In order to answer a query for a past point in time, the server can replay the log to this point in time and answer the query from that snapshot of the file system.

However, this fails to achieve the goals of *PersiFS*: reading from any version requires vast amounts of time in order to replay the log (though writing is very fast!), and space utilization will be poor because many applications rewrite the full contents of a file to disk even when only a small portion has been changed.

3.2.1 Read Optimization

To optimize this, we separate file contents from metadata. File contents are stored in a separate block store called the *superblob*, while metadata changes (including pointers to the file contents in the superblob) are stored in an *inode log*. Because the inode log stores only metadata changes, it is very compact. Furthermore, the metadata is small enough compared to the file contents that it becomes reasonable to store snapshots of the entire state of the file system metadata in the inode log at periodic intervals, allowing a fast replay to be performed by starting from the last snapshot before the desired time.

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 because inodes are never completely removed. A NFS file handle is thus simply an inumber, plus a timestamp if it does not reference the current version; no generation numbers are required. Also unlike most file systems, a *PersiFS* inumber is simply a unique identifier for a file, and has no correlation to physical disk addresses.

In order to further compact the inode log, the actual contents of the inodes are also stored in the superblob, and inode log entries contain only pointers into the superblob. This allows more inode log entries to be stored in each disk block, thereby dramatically increasing the rate at which a specific entry can be located during a replay. Furthermore, it is quite practical to keep the mapping of inumbers to superblob addresses for the current version of the file system in memory, allowing operations on the metadata of the current file system to be performed without replaying the log.

Together, the capabilities of the inode log allow *PersiFS* to nearly achieve its speed goals. Reads from the current version of the file system need only read the file contents from the superblob and writes to the current version of the file system need only append to the inode log and insert the file contents into the superblob. Answering reads for past versions of the file system only requires replaying a small amount of log from a past snapshot.

3.2.2 Write Optimization

The superblob can be optimized in order to improve the write performance of *PersiFS* and achieve its space efficiency goals. First, instead of storing entire files as blocks in the superblob, files are divided into chunks which are stored in the superblob. Thus, a modification to some part of a file need only rewrite the affected chunks. Because files in these scheme cannot be addressed by a single location in the superblob, inodes must store the sequence of chunk locations for a file.

In a further refinement, chunk boundaries are not placed at regular intervals, but instead use *content-sensitive chunking*. This technique places chunk boundaries based on file contents 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 [5] for content-sensitive chunking.

Chunking allows *PersiFS* to achieve its speed goals because it has only minor effects on read performance, while greatly improving write performance by avoiding the need to manipulate file contents that have not been modified. Chunking also improves *PersiFS*'s space efficiency by avoiding rewriting some redundant data.

3.2.3 Space Optimization

To further optimize space efficiency, the superblob leverages *chunk fusion*. 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 [7] 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 as an on-disk B^+-tree.

Combined with content-sensitive chunking, chunk fusion allows *PersiFS* to efficiently store local modifications to files, sharing most file contents between versions of that file. Furthermore, it also ensures that identical content in multiple files (for instance, due to file copying or automatic backup copies) will consume very little space beyond the single copy.

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.

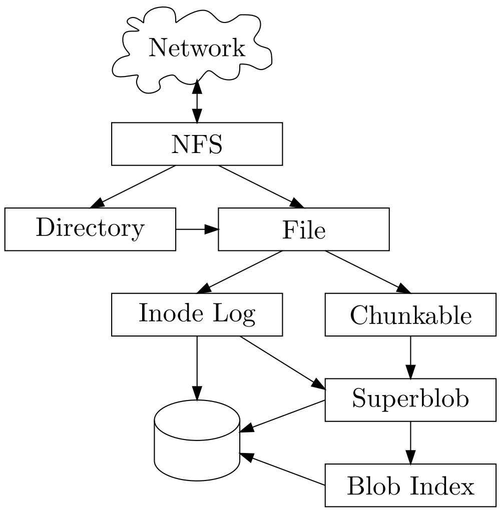

Figure 1: Implementation model

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 inumber mappings in a time-indexed log. Each mapping contains the inode number and the address of the inode in the superblob. The inode log interface allows creation, modification, and deletion of inodes to be recorded in the log, and supports scanning the log to obtain the inode map at any point in the past as well as its current state. This module also periodically records snapshots of the entire inode map in the log for rapid replay from any past point.

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`, `getAttr`, and `setAttr`.

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 contents of the string are not read from the superblob unless 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 distinct versions for each 'sub-write' would be less than desirable. It is both inefficient, since it creates many unnecessary versions that will not need to be accessed, and logically incorrect, since a read could conceivably access an inconsistent state created by a partially-performed change. Unfortunately, NFS v3 does not provide file closure notification to the server, so the desired commit-on-close semantics are impossible to achieve. Instead, the File module attempts to approximate these semantics by grouping together successive writes (in a span of five seconds) to a single file.

4.3 Network Interface

Clients access files on the *PersiFS* server using standard NFS. The *PersiFS* NFS layer exposes a magic root directory that contains a `now` directory reflecting the current file system state, as well as directories created on-the-fly that expose any past version of the file system. The NFS layer translates requests for current or past versions of the file system into operations on the underlying file and directory structures, which in turn operate on the underlying disk representation.

5 Evaluation

Upon implementing *PersiFS*, we found that it achieves performance on operations in the current version of the file system comparable to that of other NFS-based file systems. Snapshots were sufficiently compact (80K for 10,000 files) that the occasional blocking write operation was barely noticeable.

Navigating past versions of the file system was slower than expected, though still usable. Reading a past version of the file system containing 10,000 files required reading approximately 80K from the inode log (the cost was dominated by reading the previous snapshot), in addition to the time required to read the file data from the superblob (which was no more expensive than reading file data for the current version).

While examining why read performance did not reach expectations, we found that standard UNIX tools typically made more queries than expected or repeated queries. `ls`, for example, performed multiple NFS calls for each file in the listed directory in order to obtain metadata. We suspect that better caching of historic data (in particular, metadata) would greatly improve the performance of *PersiFS*.

Chunk fusion proved quite advantageous for some operations, while introducing no noticeable overhead in general. We experimented with editing a 160K text file. After four revisions, the file system had grown by merely 30K. Reducing the average chunk size (currently 8K) would have improved this further, though would have increased the time required for both read and write operations. Chunk fusion also abated the effects of programs which implement their own backup. For example, backup copies created by `Emacs` required less than a kilobyte of additional file system space, regardless of the size of the original file.

6 Future Work

In the future, we would like to provide improved administrative control over the file system. Primarily, we would like the implement user-controlled retention policy, allowing old revisions to be deleted or merged with other revisions, selectively reducing the granularity of time in order to conserve space. This capability would eliminate the primary space drawback to using a continuously versioned file system such as *PersiFS*. How to effectively implement retention policies with the existing file system structures is unclear, so new structures may be necessary.

We would also like to improve the user interface of *PersiFS*, perhaps adding support for some of the version management features typical in version control systems. One such example is tagging, in which a user is able to associate a symbol with a particular version of a file for easy reference later. It would also be advantageous to support a wider range of date specifications, including relative ones, somewhat like CVS's datespecs.

We are currently developing $PersiFS_2$, an reimplementation of *PersiFS* that applies theoretical results from the field of data structures to the underlying file system structures, primarily building on work with persistent, external memory structures. From this, we hope to gain insight into the applicability of these advanced data structures in solving real problems and to examine the trade-offs between the increased implementation complexity of these structures and the realization of their theoretical promise.

7 Conclusions

PersiFS successfully overcomes the challenges described at the outset of this paper by achieving both time and space efficiency and transparently providing persistent file system services that can be utilized without the need to modify applications. Time and space efficiency are achieved with the separation of the log from the bulk data, content-sensitive chunking, and chunk fusion. *PersiFS* behaves like a regular NFS server, providing access to different versions of the file system through a naming convention that otherwise retains regular UNIX semantics and thus does not require any changes to existing applications to work.

PersiFS has the potential to give users ease of mind and ease of use by eliminating the need to worry about the integrity of their files. For administrators of multi-user systems, most user error can be trivially dealt with by just backing up until before the mistake.

Continuously versioned file systems like *PersiFS* have the potential to change the way users view and interact with their files. By adding a time axis to the file system and giving it a very tangible archeology, users gain an extra dimension of expressive power over their manipulations of the 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 Subversion*. 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] K.-K. Muniswamy-Reddy. VERSIONFS: A versitile and user-oriented versioning file system. Master's thesis, December 2003.

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

[6] 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.

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

[8] 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.

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

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