Using Versioning to Simplify the Implementation of a Highly-Available File System

Dima Brodsky, Jody Pomkoski, Mike Feeley, Norm Hutchinson, Alex Brodsky
Department of Computer Science
University of British Columbia
{*dima,jodyp,feeley,norm,abrodsky*}*@cs.ubc.ca*

1 Introduction

One of the most urgent challenges facing systems research is to improve the reliability, availability, and maintainability of system software. The importance of these issues is being driven by new applications and more sophisticated users that increasingly expect key system software to work all of the time, even in the presence of failures or increasing load. To meet these goals, modern systems must minimize the human administration required for configuration, maintenance, and error recovery and must scale gracefully to handle loads of unprecedented size.

A modern file system must thus guarantee the availability and integrity of valuable data in the face of the widest possible range of human, software, and hardware failures. High-availability and fault tolerance are, of course, hardly new issues for file system research. A large number of systems have focused on these issues both in the context of file systems [1, 2, 4, 6] and lower-level storage [8, 5, 12]. We believe, however, that it is now possible and necessary to deliver these goals in a simpler and more automatic way.

This new approach is made possible by advances in disk storage technology. Commodity inexpensive disks currently have capacities near 100-GB and disk capacity continues to grow. As a result, modern workstation and PC nodes will have tremendous disk storage available. Enough, we believe, to substantially change how file systems use this storage to achieve key goals. In particular, abundant disk storage can be used to greatly simplify the replication of valuable data to ensure its availability and to protect it from failures.

This paper describes the design and implementation of Mammoth, a novel distributed file system that we are building. The key feature of Mammoth is that it stores files and directories as histories of immutable versions that are created each time a file or directory is modified. All Mammoth meta-data is stored in append-only fashion and all file data is read-only once it is created. As a result, Mammoth can replicate data on multiple loosely-connected nodes with minimal concern for consistency. The only overhead for creating and maintaining a replica is the storage it occupies.

Mammoth replicates file data and meta-data as necessary to satisfy *availability policies* specified by users. Users assign these policies to individual files or groups of files and the system uses these policies to determine which file versions it replicates and on which nodes. This approach allows users to determine how their files are replicated, a task that is typically left to a system administrator in a traditional system.

Mammoth handles node and network failures by allowing users to operate on whatever versions of files are currently accessible, even if more recent versions exist on other, currently unavailable nodes. This approach is similar to disconnected operation in Coda [4], with a key difference. Mammoth's versioning approach greatly simplifies consistency maintenance and it provides a natural way to represent update conflicts in the file system. If disconnected nodes modify the same version of a file in Mammoth, the file system stores both new versions and records a branch in the file's history.

In summary, our goal is to build a simpler, more robust, and more scalable distributed file system. We believe that the fact that our approach is loosely coupled and requires minimal global coordination indicates that it has promise to substantially improve on previous systems such as those that implement RAID-like striping across multiple nodes [1, 2], implement virtual-disk level RAID or mirroring [5, 11, 12], or perform synchronous file updates to multiple nodes [4, 6].

2 Overview of Mammoth

A Mammoth file system consists of a set of loosely-connected nodes that each replicate portions of the file system's meta data. The portion replicated at a node is determined by the set of directories to which the node has *registered interest*; to register interest in a directory, a node must also register interest in its parent. Nodes can register and unregister interest in a directory at any time by sending a request to all nodes interested in its parent or, in the case of unregister, the directory itself.

Each node replicates the meta-data of the directories it is interested in and of the files they contain. Lookup operations on these directories and files can thus complete locally. Nodes propagate the meta-data changes they make to all *interested* nodes so that the replicated meta-data eventually becomes consistent. This procedure is simple, because all updates take the form of records appended to the history of a directory or file.

File data is not propagated eagerly. Thus, while multiple nodes may replicate a file's meta data, they do not necessarily store all of the file's versions. Instead, a version is stored at a node only if that node has accessed the version or if the system has sent the version there to protect if from failure.

File versions are located using the file's meta-data, which lists the versions that exist and the nodes that store them. This list may, however, be incomplete or inaccurate due to the loosely-connected eventual consistency scheme used to update file meta-data. These temporary inconsistencies can cause a node to be unaware of versions created at other nodes or to think a node stores a version that has been removed by that node. We discuss how the system handles these conditions below.

A key feature that distinguishes Mammoth from other distributed file systems is that each Mammoth node stores a potentially autonomous local file system. Nodes that share files or directories replicate their meta-data locally, but each node need only replicate a portion of the file system. This design simplifies the overall implementation of the system and it simplifies how the system handles failures.

It is also worth nothing that Mammoth extends the ideas of the single-node Elephant file system [10], though it is implemented from scratch. As in Elephant, Mammoth users can rollback a file or directory to any point in the past by specifying a date and time as part of any pathname they provide to the file system. The system responds with the file and directory versions that existed at that point in time. The key differences between Mammoth and Elephant file histories are that Mammoth is distributed and that Mammoth allows a file history to store multiple concurrent branches in order to support partial failures.

The remainder of this section discusses four key issues for the design of Mammoth: locating the current version of a file, controlling replication, handling failure, and providing scalability.

2.1 Locating the Current Version

Locating the *current* version of a file is complicated when multiple nodes actively share a file. Typically, the replicated meta-data for a file is consistent on every node that has registered interested in the file and thus the current version can be easily determined locally on any of these nodes. When a new version is produced, however, a temporary inconsistency exists, until Mammoth propagates information about the new version to every interested node. If a node attempts to access the current version in this interval, it may select the out dated version just modified instead of the new version. If the node subsequently updates this version, it will create a branch in the file's history, with two *concurrent* versions being derived from the same older version.

Mammoth uses file locks to prevent this problem. Locks are multiple-reader, single writer. One lock holder is designated to be the *owner* and it keeps a list of all of the nodes that currently hold the lock. The owner is located by following a forwarding chain left behind when ownership changes or by broadcasting to all nodes interested in the file. Nodes that hold either a reader or writer lock can identify the current version locally, but a writer lock is required to produce a new version. All other nodes contact the owner to identify the current version. If the lock's owner cannot be located due to failure, any node can become the lock's new owner. It is only in this case that nodes may receive out-of-date versions and thus create history branches.

2.2 Replication

Mammoth users control replication by assigning *availability policies* to files or groups of files. A policy typically specifies the type of failure the file should be protected from and the amount of work, if any, that the user is willing to lose should such a failure occur. Multiple policies can be assigned to a file. The system uses these policies to decide which versions to replicate and on which nodes. Replicated versions can be reclaimed when they are no longer needed by a cleaner thread that runs periodically on each node.

A key feature of the system is that it is likely that not every version of a file will be replicated. This is particularly true, for example, for files replicated to a remote site to protect against catastrophic failure. In this case, we expect that versions will be replicated infrequently, say daily, trading off the amount of work that may be lost when such a failure occurs to reduce the cost of this distant replication. In addition, less important files may be replicated less aggressively than important ones.

In the event of a failure, it is thus possible that only limited versions of a file will be available. In particular, the current version may not be available. This situation is similar to what happens in systems that rely on off-line backup to protect file data, where failure typically results in the loss of the recently created versions of files. A key issue for Mammoth is to ensure that, in the event of failure, the versions that are available are consistent with each other. Backup systems — whether automated such as AFS [7], Plan-9 [9], or WAFL [3], or manual — deal with this problem by creating consistent checkpoints of the entire file system on the backup media. Mammoth takes a different approach.

In Mammoth, users establish consistent checkpoints at the granularity of arbitrary file groupings, defined by users. Users group files that are mutually interdependent and the file system replicates file versions as consistent checkpoints of these groups. When Mammoth replicates a file version that existed at time *x*, it thus also ensures that all other files in the group that have a version that existed at time *x* are replicated in the same way.

In summary, the key benefit of Mammoth's approach is that it provides users with flexibility for deciding which of their files should be replicated and how aggressively. This benefit would be lost, however, if inter-file consistency was defined at the granularity of the entire file system, as is the case in previous systems. File-group consistent checkpoints solve this problem, but require that users identify file inter-dependencies explicitly to the file system. We believe that this flexibility will prove valuable, but this will be one of the key questions we will investigate by studying how people use the Mammoth prototype.

2.3 Handling Failures

Mammoth handles failures by allowing users to access and update whatever file versions are currently available. While Mammoth ensures that this view is consistent, as defined above, it may nevertheless be out of date. In addition, even if the current file versions are available, network partitions can cause file histories to diverge. In either case, conflicting updates are detected when disconnected or failed nodes reconnect and they are recorded as branches in the file's history. This procedure is handled by Mammoth's meta-data consistency mechanism.

To see how this works, recall that when a node changes the meta-data of a file or directory, it propagates this change to all other nodes that have registered interest in the file or directory. In the event of a failure, this update may not be received by some of these nodes. If this is the case, the updating node detects the communication failure and records the meta-date update information in a local log. Each node periodically probes the nodes for which is has pending meta-data updates, sending these updates when communication is re-established. All other global information is propagated in a similar way. For example, if multiple nodes have become the lock owner for a file during disconnected operation, the redundant owners use this procedure to discover and resolve this situation.

2.4 Scalability

Mammoth simplifies lookup by requiring that for a node to register interest in a file, it must also register interest in all directories on the path from that directory up to the root of the file system. Directories near the root of the filesystem will thus tend to be replicated widely and, as a result, the consistency overhead for making changes to these directories will be higher than for less-replicated directories further down in the hierarchy. This wide-scale replication may limit scalability if high-level directories change frequently. If changes tend to be focused near the leaves, however, Mammoth will scale incrementally by partitioning interest in subtrees of the file system across an ever increasing set of nodes, a procedure that we intend will be handled adaptively by the file system. These issues for scalability will be a key focus of our evaluation of the Mammoth prototype.

3 The Prototype

3.1 Overview

At the present time we are building a prototype to evaluate our ideas. The current Mammoth prototype is implemented as an extension to the Linux user-level NFS server. All Mammoth meta-data and data are stored in files in the unmodified Linux file systems on the nodes that run the Mammoth server.

Mammoth clients can be unmodified NFS clients, but we have also implemented a modified NFS client for FreeBSD 4.0 that augments the standard NFS protocol with *open* and *close* operations. Mammoth needs to see opens and closes in order to determine when writes should result in the creation of a new version of the file. Mammoth follows the Elephant approach of creating a new version each time a file is opened for writing. When unmodified clients access a Mammoth server, the server uses a heuristic to attempt to guess when opens and closes may have occurred.

In addition, we have also implemented a single node version of Mammoth as a stackable file system in the FreeBSD 4.0 kernel. This version uses the same meta-data as the Mammoth NFS server and it also stores file data and meta-data in an unmodified file system. Our eventual goal is to provide an in-kernel solution that allows nodes to act as both a client and server to Mammoth and an NFS version that allows unmodified clients to access the system.

We expect that the NFS-server version of Mammoth will be fully operational by the time of the workshop and we will be prepared to provide a preliminary evaluation of its performance and other features.

3.2 NFS Clients

Mammoth clients should be able to dynamically choose a server that is currently available, nearby, and lightly loaded. In our prototype, however, clients use NFS to communicate with server nodes. Due to the nature of existing NFS clients our prototype is not able to dynamically assign clients to servers. Currently, a client must explicitly specify a server when it mounts it. For the time being, this approach is sufficient for us to evaluate our prototype.

3.3 File Meta-Data

Mammoth meta-data is stored as files in a shadow directory similar to that of AFS [7]. This structure optimizes for access to the current version of a file. This version can be read without reading any on-disk Mammoth meta-data, as long as the server has registered interest in the file and currently holds either a read or write lock. All meta-data is stored in an append-only fashion.

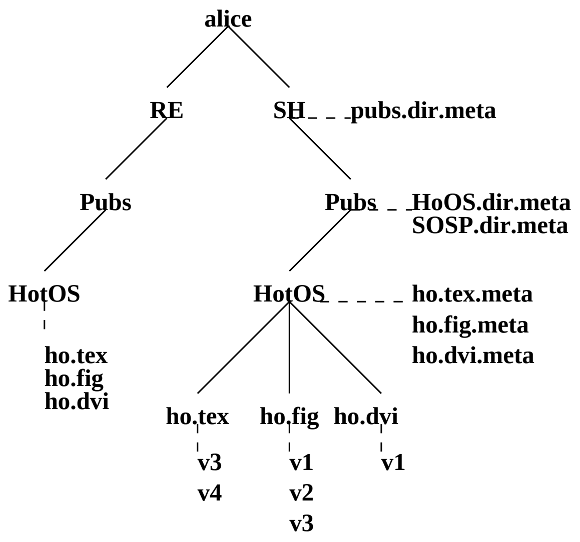

Figure 1: The file's meta-data.

Figure 1 presents an example of a directory tree rooted at `alice` and its associated meta-data. Current versions are kept in the `RE` subtree. Meta-data and previous versions are stored in the `SH` subtree.

When Alice opens `alice/Pubs/HotOS/ho.tex` the server returns `alice/RE/Pubs/HotOS/ho.tex`. On a *write* the file `alice/RE/Pubs/HotOS/ho.tex` is moved to `alice/SH/Pubs/HotOS/ho.tex/v5` and a new `alice/RE/Pubs/HotOS/ho.tex` is created. On a *close* the meta-data is updated by appending the new version record to `alice/SH/Pubs/HotOS/ho.tex.meta`.

A directory's meta-data file stores (1) a list of name entries annotated with the time they were created and removed and (2) a list of nodes that are interested in the directory.

A file's meta-data stores (1) a list of file version information as described below, (2) the file's current and previous owner and lock status, (3) a list of replication servers, and (4) the file's availability policies and groupings. The replication servers list is used by the system replication thread as a heuristic to provide a degree of locality when selecting nodes to store replicas of this file.

Finally, each node also stores a queue of pending meta-data updates for each node that is currently unavailable.

3.4 File History

A file's history chronicles its existence from its creation to its deletion; it is stored as a linear list. Each entry corresponds to a version of the file and contains information to determine where and when the version was created and where the previous and the next version is located. An entry is created and written in append-only fashion when the server receives a *close* for that file.

File versions are uniquely named by the pair consisting of the name of the node that created it and its creation time on that node. Each history entry stores the name of a version and the name of the history branch on which that version resides. Branches are named by a pair consisting of the name of the version that created the branch and its parent. This approach to branch naming is taken to simplify history maintenance.

A snippet of a file's history is shown in Figure 2. In this case, Node A has modified the file twice and node B once, before a branch occurs. At this point, both nodes B and node C produce new versions that are based on version (B, T_{b0}) and the two branches they create are named $((B, T_{b1}), B, T_{b0}))$ and $((C, T_{c0}), (B, T_{b0}))$, respectively.

$$\begin{array}{cc}
\multicolumn{2}{c}{(A, T_{a0}), ((A, T_{ao}), nil)} \\
\multicolumn{2}{c}{(A, T_{a1}), ((A, T_{a0}), nil)} \\
\multicolumn{2}{c}{(B, T_{b0}), ((A, T_{a0}), nil)} \\
\vdots & \vdots \\
(B, T_{b1}), ((B, T_{b1}), (B, T_{b0})) & (C, T_{c0}), ((C, T_{c0}), (B, T_{b0})) \\
(B, T_{b2}), ((B, T_{b1}), (B, T_{b0})) & (C, T_{c1}), ((C, T_{c0}), (B, T_{b0}))
\end{array}$$

Figure 2: Branching file history.

4 Issues

We fully understand that this paper raises as many questions as it answers. We plan to address several of these key issues through experimentation with the Mammoth prototype. We plan to evaluate:

- the performance and scalability of our meta-data replication and eventual-consistency mechanisms;
- what availability policies people find useful;
- issues that arise in practice when replicating versions as we suggest (e.g., how important are load balancing and locality); and
- whether our approach is indeed as simple and robust as we expect.

5 Conclusion

This paper has described the design and implementation of Mammoth, a novel distributed file system that protects valuable file system data from human, software, hardware, and site failures by replicating file versions among a set of loosely-connected nodes. The key idea of Mammoth is that file and directory meta-data is stored in append-only fashion and that file data itself is immutable; writing to a file creates a new version. This structure permits us to implement replication with minimal concern for consistency and without global coordination. Users control replication at the granularity of files and they group files to form fine-grain consistent checkpoints. The system automatically replicates file versions on remote nodes as necessary to implement these policies.

We are implementing two prototypes: one that runs as a user-level NFS server and the other implemented as a stackable file system in the FreeBSD kernel. Our goal is to allow unmodified (and mostly unmodified) NFS clients to access the file system and to use a real file system to store all meta-data and file data.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.

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

[4] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25, October 1991.

[5] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Computer Architecture News*, pages 84–93, October 1996.

[6] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38, October 1991.

[7] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[8] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data: 1988 Annual Conference, Chicago, Illinois, June 1–3*, pages 109–116, 1988.

[9] D. Presotto. Plan 9. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 31–38, April 1992.

[10] D. S. Santry, M. J. Feeley, N. C. H., 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*, pages 110–123, December 1999.

[11] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[12] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.