Loosely Coupled Optimistic Replication for Highly Available, Scalable Storage

Dima Brodsky, Jody Pomkoski, Michael J. Feeley, Norman Hutchinson, and Alex Brodsky

Department of Computer Science
University of British Columbia
{*dima,jodyp,feeley,norm,abrodsky*}*@cs.ubc.ca*

Abstract

People are becoming increasingly reliant on computing devices and are trusting increasingly important data to persistent storage. These systems should protect this data from failure and ensure that it is available anytime, from anywhere. Unfortunately, traditional mechanisms for ensuring high availability suffer from the complexity of maintaining consistent, distributed replicas of data.

This paper describes Mammoth, a novel file system that uses a loosely-connected set of nodes to replicate data and maintain consistency. The key idea of Mammoth is that files and directories are stored as histories of immutable versions and that all meta-data is stored in append-only change logs. Users specify availability policies *for their files and the system uses these policies to replicate certain, but not necessarily all, versions to remote nodes to protect them from a variety of failures. Because file data is immutable, it can be freely replicated without complicating the file's consistency. File and directory meta-data is replicated using an* optimistic *policy that allows partitioned nodes to read and write whatever file versions are currently accessible. When network partitions heal, inconsistent meta-data is reconciled by merging the meta-data updates made in each partition; conflicting updates manifest as branches in the file's or directory's history and can thus can be further resolved by higher-level software or users. We describe our design and the implementation and performance of an early prototype.*

1 Introduction

Humans at home, at play, and at work are becoming increasingly dependent on computing devices and their storage systems. Persistent storage plays a central role in most interactions people have with computers, because it holds the data they produce and manipulate. As people's dependency on computers increases, this data becomes an increasingly valuable asset. It thus seems only reasonable for users to expect that this data be protected from failure and be available anytime, from anywhere. The capabilities of most file systems, however, fall far short of these expectations.

Commodity file systems typically offer users only one simple protection from failure: periodic backup of their data to offline storage. This traditional backup and restore approach, however, provides limited protection, has high cost, and scales poorly. While offline backups protect most data from loss, they do nothing to increase availability and they leave updates that occur between backups vulnerable to loss. Backup is costly because it typically requires substantial human administration. As a result, for example, people rarely backup data themselves and many professionally backed-up file systems provide optional unbacked-up storage, which has a lower cost. Finally, offline backup systems are being strained to the breaking point by dramatic increases in the amount of data users store and by the number of users they are expected to service.

A few systems improve things somewhat by automatically maintaining a collection of file system checkpoints [13, 10, 3]. If available online, these checkpoints automate the recovery of files that users have recently, accidentally deleted or overwritten. They do not, however, protect data from failure or increase its availability, because for efficiency checkpoints are copy-on-write images of the active file system and thus unable to stand-in for the active data in the event of a failure. In fact, checkpoints are even limited in their ability to protect users from their mistakes, because they typically limit either the frequency of backups or the number of backups the system maintains. In an earlier paper, we described a solution to this limitation that uses per-file, rather than per-file system checkpoints [15].

Some research systems [5, 17, 14, 9, 8, 16, 2, 18], hardware storage [11, 19], and high-end file systems actively replicate data as it is modified. This approach can be used to both protect data from failure and to increase its availability. Replication is complicated, however, by the requirement of keeping replicated files consistent. Furthermore, to protect data from node and network failures increases complexity substantially, because data is replicated among multiple network nodes and thus consistency becomes a distributed problem. In such systems, replica nodes are typically tightly coupled so that they can impose a global order on updates seen by clients and so that they can recover from failures in an orderly fashion. This tight coupling and high complexity limit scalability and increase the difficulty of building, maintaining, and administering highly available file systems.

Dealing with network partitions is particularly challenging, because systems must either prohibit updates to partitioned replicas or deal with the possibility of conflicting updates to replicas in different partitions. The later approach, called *optimistic replication*, is the only choice if users are to be guaranteed access to their files in a system that is large enough and distributed enough that partitions are common.

In systems that support optimistic replication, a complex, tightly-coupled algorithm is typically required to reconcile partitioned replica nodes when communication between them is re-established. In addition, algorithms commonly assume that full-connectivity is the normal state of the system. This assumption is useful, because it allows the algorithms to rely on global communication to arbitrate the reconciliation process and to, for example, know when update logs can be trimmed. In a sufficiently large collection of server nodes, however, failures will be common, particularly if the servers are geographically distributed. Supporting optimistic replication thus tends to limit scalability in these systems.

In fact, scalability and geographic distribution are particularly important goals for meeting user expectations for completely reliable, always available storage [4], because of the role that *storage utilities* can play in this arena. The idea of a storage utility is to offload storage-management responsibilities from users to logically-centralized, network-connected servers operated by trusted third parties. The key advantages of this approach are that data protection does not depend on the actions of users themselves, to backup their data for example, and that their files can be easily accessible from any network-connected device. The success of this idea, however, likely requires that servers scale to many thousands of users and petabytes of storage.

The advancement of file system reliability and scalability thus appears to face a dilemma. The techniques required to provide the protection and availability semantics that users are likely to demand are difficult to implement, administer, and scale, because they require that nodes cooperate in a tightly-coupled fashion to replicate data, maintain its consistency, and reconcile partitioned replicas. This paper examines an alternative approach that replicates data in a simple, flexible, automatic, and loosely-coupled way.

Mammoth

This paper describes the design and initial prototype implementation of a novel file system called Mammoth. Our goal is to develop a reliable, highly-available file system that utilizes simple mechanisms, requires minimal human administration, and scales to utility-class levels.

The key features of Mammoth are that it stores files and directories as collections of immutable versions and that server nodes operate in a loosely connected fashion. Immutable versions simplify replication and consistency management and increase replication flexibility. Loose connectivity simplifies scalability and fault tolerance, by allowing nodes to have partial, weakly-consistent knowledge of the system and to operate with whatever file versions are currently available, without requiring coordination with unavailable nodes.

Mammoth users control when files are replicated by assigning files *availability policies*. These policies allow users to indicate the types of failure they want protection from, and the amount of work they are willing to loose should such a failure occur. For example, a user might be willing to lose an hour's work in the event of a disk failure and a day's work in the event their building burns down. Giving this control to users simplifies system administration, because it leaves policy decisions in the hands of users, not administrators. We believe that this transfer of responsibility is safe as long as policies closely reflect the high-level model that users already have regarding protection. The system translates these high-level descriptions into the actions necessary to guarantee the protection and availability specified by the policies.

Mammoth's use of immutable versions gives the system flexibility to delay replication of newly created versions, taking replication overhead off the critical path of updates and allowing multiple updates to be absorbed into a single replication. Unlike other systems that ensure the availability of the current version of the file system, Mammoth has the flexibility to ensure availability of any version. The version that is available is determined by user policies, allowing users to tradeoff increased availability and protection for increased cost of replication. Utilities could use this flexibility, for example, to charge users more money to get higher degrees of protection.

Mammoth's version-history approach also simplifies replication consistency management. The immutability of versions means that no consistency action is needed for replicated file data; the creation of a new version, logically invalidates older replicated versions. File versions can be freely replicated, anywhere at any time, without increasing consistency overhead. Of course, for users to see a consistent view of the file system, the version-history meta-data of a file or directory must be kept consistent.

Mammoth simplifies meta-data consistency maintenance by storing this information as append-only logs. A node can update a remote copy of a modified file's meta-data, for example, simply by transmitting a history entry for the newly created version that consists mainly of its creation timestamp and a list of nodes that store the data associated with that version. The receiving node incorporates this and other meta-data updates in a straightforward way, by appending them to the history it already stores. In particular, the order in which updates are applied to replicated meta-data — a key issue for tightly-coupled approaches — is unimportant for its eventual consistency in Mammoth.

Mammoth supports optimistic replication with similar simplicity. When nodes that share a file or directory are connected, Mammoth uses a simple locking mechanism coordinated by one of the sharing nodes to ensure that nodes see consistent versions of the meta-data in question. If a partition or other failure makes it impossible for a node to acquire a lock, however, Mammoth allows the requesting node to become a lock manager and proceed with its update.

This optimistic approach can result in conflicting updates when the same object is modified in multiple partitions. These conflicts are detected when partitioned nodes re-establish communication with each other. The nodes then reconcile their shared file and directory histories to bring each other up to date with the changes that occurred while they were out of touch. Unlike tightly-coupled approaches, conflicting updates are recorded in the file system without requiring that they be further reconciled. Instead, conflicts are stored as branches in the object's history.

Our approach has the advantage that complete reconciliation, which can be complicated and can require application-level involvement, is moved from the critical path of partition recovery. Our goal is to divide reconciliation cleanly between two layers of the system. The underlying storage system layer is responsible for avoiding conflicts when possible and storing complete information about conflicts when they are unavoidable. Like a traditional file system, however, the storage layer has no knowledge of the data semantics typically required to resolve true conflicts. This task is handled by a higher layer: either the user, an application, or middleware. We expect that strategies used by systems such as Bayou [12], OceanStore [6], and Coda [5] can be built on top of the basic storage facilities provided by Mammoth and we plan to pursue this idea in future work.

2 Mammoth Design

This section describes the design of Mammoth. We begin with an overview and follow that with a detailed description of key elements of the design and the issues it raises.

2.1 Overview

A Mammoth system consists of a set of loosely-connected server nodes that replicate portions of a shared file system. Users access files directly on server nodes or on unmodified client nodes running a standard protocol such as NFS.

Directory and file meta-data are stored as append-only version histories and file data is stored as a sequence of immutable versions. Mammoth replicates both file data and meta-data among server nodes to protect it from failure, to ensure its availability, and to improve performance. The system controls replication according to user-assigned *availability policies*, pushing information to remote nodes as necessary. In addition, data is pulled on demand from remote servers when accessed by clients.

Mammoth's use of immutable versions means that consistency issues are confined to replicated meta-data; versions themselves can be replicated without concern for consistency. Nodes that replicate an object's meta-data are responsible for keeping the object consistent. Each node registers with the others and requests either invalidation or update-based consistency, depending on how frequently the node expects to access the object. When a node changes an object's meta-data it performs the necessary invalidations and updates the other nodes. Sharing is coordinated in a similar way using advisory locks associated with each object. These locks prevent conflicting updates when possible, but allow them when necessary due to network partitions.

To maximize availability, Mammoth adopts an *optimistic replication* policy for directories and files. As a result, network partitions can cause an object's meta-data to become inconsistent when it is updated concurrently by partitioned nodes. When partitioned nodes re-establish communication with each other, inconsistent meta-data they store must be reconciled.

In Mammoth inconsistent versions of an object's meta-data differ only by missing some history entries stored by other versions on other nodes. The consistent view of the object can thus be formed by taking the union of all replicas. In the merged meta-data, conflicting updates appear as branches in its history.

2.2 Version History Meta-data

Mammoth adopts the basic versioning model of the Elephant file system [15], extending it to deal with distribution and replication. In both systems, files and directories are stored as histories of immutable versions created each time a file or directory is created, deleted, or renamed, or when a file is updated (i.e., when a modified file is closed). Users access older versions of files and directories by adding a timestamp to their names. The system responds with the versions that existed at the specified time, essentially providing the user with a *continuous* consistent checkpoint of the system. Unimportant old versions are deleted by a background cleaner thread that is guided by *retention policies* that users assign to files to indicate which versions are important.

In Mammoth, the history of a file or directory is represented logically as a tree that links each version to its predecessor. It is a tree because Mammoth allows multiple new versions to be created from a single ancestor. Branches are created explicitly by users and implicitly by the system to handle network partitions. A branch is implicitly created whenever a replicated object is modified on two sides of a network partition. Users can examine the history of a object and can combine the tails of of multiple branches into a single new version.

Branches introduce ambiguity in version naming, because they allow multiple versions of an object to be *current* at the same time. To avoid this ambiguity, versions are named internally by combining the node number on which the are created with their creation timestamp. When selecting versions, users can use these combined names or they can specify a default history branch to be used to resolve the timestamps they provide. When a node experiences a network partition and is involved in the creation of a branch, that branch remains its default history branch after the network partition is healed and the conflict discovered. In this way, users don't experience surprising discontinuities in the files that they access.

Directory and file meta-data also contain a list of history entries and some additional information. In addition to the history, the meta-data stores the object's lock state, *replication set*, and *interest set*. The use of these fields is described later in this section.

A directory's history records an entry for every change made to the directory. There are two types of entries, one for name creation and the other for name deletion. Each entry stores its type, timestamp, and name. Creation entries also store a copy of the file's interest set, for reasons that are explained below. For simplicity we prohibit hard links across directories; a similar restriction exists in Coda [7].

Finally, a file's history records an entry for every change made to the file. The most common type of change is the creation of a new version. New-version entries store the internal name of the version, the name of the version it was derived from, and the version's *storage set*. The storage set is the list of nodes that are known to store the version. Additional entries are added anytime a version's storage set changes, which occurs when it is replicated or when the cleaner thread at a replica node deletes the version.

2.3 Accessing Meta-data

To access a file or directory, a server node requires the object's meta-data. As mentioned previously, meta-data is cached at servers on demand and its consistency is based on either an invalidation or update protocol. The default protocol in Mammoth is invalidation; registering *interest* in a file is the mechanism by which a node selects an update protocol. A node can register, or unregister, interest in any directory or file. Once registered, the node's locally-stored meta-data will be kept up-to-date by other nodes whenever any of them generates a new history entry.

A node can also cache meta-data without registering interest if it instead requests a call-back *read lock* for the meta-data. In this case, lock revocation acts as an invalidation for the cached meta-data. Locks are discussed below.

To register interest in a directory or file (or to lock it) a node sends a registration request to the object's owner node. The owner sends an updated copy of the object's meta-data to the requesting node and it adds the node to the object's interest set by sending a message to all nodes in either its or its parent directory's interest set. If the object's owner node is unreachable, a new owner is selected from among the object's interest set and that new owner completes the protocol. If none of the nodes in an object's interest set is reachable then access to the object is denied.

If the requesting node is currently interested in the target object's parent directory, it can easily locate the object's owner using the interest set stored in the object's directory entry. In this case, the node selects one of the object's interested nodes and sends its registration request to that node. This node will know the file's owner and will thus forward the request there. If the requesting node is not interested in this directory, however, a sequence of messages are needed to track down a node that does store this information.

If necessary, the search for an object's owner begins with the object's closest ancestor directory that the requesting node *has* registered interest in. This directory stores the interest set for the next directory on the path to the target, and so the node picks one of these nodes and forwards the registration request to it. This process is repeated at that node and each subsequent node until the object's owner node, or one of the nodes in its interest set, is located.

If a node has not registered interest in any directory on the path to the target object, a different process is required to locate a node interested in the root directory. This process is required, for example, when a new Mammoth server is accessed for the first time. Mammoth ensures that enough nodes cache the root directory so that it is highly available. The identity of these nodes is provided to Mammoth servers as a set of *first contact nodes* when they are initialized.

2.4 File Locks

To access the *current version* of a file, a node must hold either a shared-read or an exclusive-write lock for the file, depending on the type of access. The acquisition of this lock is implicit in all open calls that do not name a particular version of the file by appending a timestamp to its name. This use of locks allows the system to ensure that the open call returns the globally current version of the file at the time of the open. Our prototype implements NFS sharing semantics and thus does not synchronize concurrent updates to a file, though this could easily be added.

If a node does not already hold a lock, it acquires it by sending a request to the file's owner node. The owner adds the node to the file's lock state and sends the timestamp of the file's current version back to the requesting node. When the requesting node has received both the owner's response and the history entry named by this timestamp, it holds the lock and its access can proceed.

For write-lock requests, the owner also sends invalidation messages to all nodes in the locks reader set and transfers lock ownership to the requesting node. The other nodes are informed of this ownership transfer the next time they receive a new history entry, usually when the new owner closes the file. In the meantime, the old owner forwards requests it receives to the new owner.

If a node is unable to contact a file's owner to acquire a lock, it becomes the files owner and grants itself the appropriate access. If the failure is due to a network partition, the file will now have multiple owners, potentially one in each partition. When partitions heal, these inconsistency are resolved and a single owner re-established.

2.5 Replication

Users assign one or more *availability policies* to files and optionally group files together into an *availability group* for each policy. The system uses this information to decide when to replicate versions of files and to which nodes. If the policy calls for every version of a file to be replicated, then replication occurs each time a new version is produced. Otherwise, a background replication thread triggers replication of the current version of a file when necessary. Most policies, for example, place a bound on the maximum amount of time between the creation of a new version of a file and its replication.

Files in an availability group are replicated together to ensure that every replica contains a consistent snapshot of all files in the group. For example if a replication policy calls for the version of a file created at time t to be replicated, the system ensures that the version of all files in the group that were current at t are also replicated. This constraint may require the replication of additional versions, but only for files that have changed since their last replication.

The replication thread is driven by a course-grain priority queue of scheduled replications stored at each node. The system inserts entries into this queue as necessary when files are modified. When a new version is created, the system examines each of its availability policies in turn. For each policy, it first checks the file's meta-data to determine whether the node has already scheduled the file for replication, when an earlier version was created; if so, no further action is needed. Otherwise, a new scheduled-replication time is computed, an entry is inserted into the node's schedule queue, and the file's meta-data is updated.

The replication thread periodically examines the head of the schedule queue to determine which files should be replicated. To replicate a file, it selects one or more replication nodes, sends the current version of the file there, updates the version's storage set, and sends this update to all nodes that are interested in the file.

Replication nodes are chosen by consulting two data structures. First, each node is provided with a database of replication candidates at startup that lists a subset of the nodes in the system, annotated with information needed by the replication policies. For example, a policy designed to protect data from a building fire must know in which building each node resides. Second, a *replication set* is maintained as part of the meta-data of each file and is used as an heuristic to achieve locality for replicas of all files in its availability group.

If multiple nodes modify a file, each of them independently schedules replication of the file. This redundancy is necessary to ensure that the file is replicated even if one of the nodes fails before completing the replication. Note, however, that if *all* such nodes fail and thus the file isn't replicated on schedule, there are no versions available for replication anyway. The replication thread can often avoid actually performing redundant replications by checking the file's meta-data before it replicates the file to see if replication by another node has already occurred. It will know about these replications if it is interested in the file, which it usually will be, because it will receive a meta-data update each time a replica node is added to the storage set of any file version.

2.6 Cleaning

A background cleaner thread runs on every node to delete unimportant file versions as necessary to economize local disk storage. The cleaner determines which versions can be deleted, by examining each file's meta-data and their retention and availability policies. For example, replicating a new version of a file typically makes an older replica of the file obsolete. In addition, old versions of files are reclaimed in the same way as in Elephant. Finally, meta-data can be deleted from a node when the node is not in its interest set, does not hold a lock on it, and does not store any file versions associated with it.

The cleaner can also compress file meta-data histories to remove entries for versions that no longer exists. As in Elephant, however, the system retains sufficient information in the meta-data of a file to determine where versions are missing from the file's history. Mammoth does this by replacing the entries for deleted versions that span an interval of time with a single entry that indicates that now-deleted versions existed in that interval. Consistency of compressed meta-data is handled by retaining an epoch number for the file that is incremented whenever its meta-data is compressed. Finally, when the cleaner removes the last version of a file from the system, it can also remove the file's meta data and remove it from its parent directory.

2.7 Meta-data Consistency and Failure

A key issue for any optimistic replication scheme is reconciling updates made to an object in different network partitions. The goal of our design is to handle this reconciliation as simply and with as little distributed coordination as possible. We believe that this idea is interesting and promising, but one important issue remains to be addressed.

When a node updates a file or directory, it attempts to update all nodes in the object's interest set. It enqueues any updates it is unable to send until communication with unreachable nodes is re-established. This procedure eventually reconciles partition inconsistencies as long as (1) a node does not permanently fail while holding enqueued updates and (2) a node has an accurate interest set for each object at the time it modifies it. The interest set may be incomplete, however, if a node was added to the interest set in another partition prior to the update.

The first problem — failure of a node while it holds incompletely propagated updates — is solved by requiring that nodes track updates they receive from other nodes. To do this, each node maintains a log of updates it receives that are not known to have been fully propagated. A log entry stores the object's name and a list of suspect nodes; the details of the update itself, however, are stored naturally in the object's meta-data. When updating an object, two messages are now required to each interested node: one that contains the update and a second that lists the nodes that successfully received it; this process can be optimized by delaying the second set of messages and batching them with other messages. Nodes monitor the accessibility of suspect nodes stored in their log and forward unpropagated updates to them when they become reachable.

The second problem — updating with an incomplete interest set — is more challenging. The problem is caused by our decision to allow a node to become the owner of a file or directory whenever its current owner is unreachable. If this does not happen, then an object's owner can easily ensure that interest-set changes and other meta-data updates are properly ordered. In the case of files, the owner ensures that a node's interest set is up-to-date before granting a write lock to it. In the case of directories, which are modified without locking, the owner adds a timestamp to its interest-set-change message so that receiving nodes can determine if they have made any updates ordered after the change, and if so forward them to the newly interested node.

If an object does have multiple partitioned owners, we have a problem only if one of them adds a new node to the object's interest set. If this happens, nodes in each partition will have different interest sets for the object, and thus meta-data updates may not fully propagate to all interested nodes. This inconsistency will be easily detected, however, as soon one of the nodes in the intersection of the divergent interest sets becomes available in both partitions. When this happens that node will receive meta-data updates from both owners and thus know to initiate reconciliation of the object. A problem remains, however, should partitioned interest sets diverge to the point that none of the nodes they have in common ever become available in both partitions. We expect this situation to be rare, but we do not at this point have a complete solution to handle it when it does happen. We believe that the solution will require maintaining extra information about ownership transfer and we continue to explore possible solutions.

3 The Prototype

3.1 Overview

At the present time we are building a prototype to evaluate our design. The current Mammoth prototype is implemented as an extension to the Linux user-level NFS server. All Mammoth file data and meta-data are stored in files in the unmodified Linux *EXT2* 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 Linux 2.4.6 and FreeBSD 4.3 that augments the standard NFS protocol with a *close* operation. Mammoth needs to be able to track file *open* and *close* in order to determine when to create a new version of the file. Mammoth is able to determine a file *open* by tracking the *setattr*, *write*, and *read* RPC calls to the NFS server. When unmodified clients access a Mammoth server, the server uses a heuristic to attempt to guess when closes may have occurred.

In addition, we have also implemented a currently-single-node version of Mammoth as a stackable file system in the FreeBSD 4.3 kernel. This version stores file data and meta-data in the same way as the user-level server. 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.

3.2 Status

Our current prototype implements a subset of the design described in Section 2. Features that have been fully implemented include: basic file system operations, propagation of meta-data updates, interest registration and unregistration, and file locking. Background replication has been mostly implemented, but only for a simple set of policies; replication groups have not been implemented. Of the reconciliation mechanisms described in Section 2.7, only update enqueue and re-transmission has been implemented. Finally, there is currently no cleaner.

3.3 Meta-data

Mammoth meta-data is stored as files in a shadow directory similar to that of AFS [10]. This structure is designed to optimize 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. 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 `..Pubs..fast.tex` the server returns `..RE/Pubs..fast.tex`. On a *write* the file `..RE/Pubs..fast.tex` is moved to `..SH/Pubs..fast.tex/fast.tex.nd1-v3` and a new `..RE/Pubs..fast.tex` is created. On a *close* the meta-data is updated by appending the new version record to `..SH..fast.tex.md`.

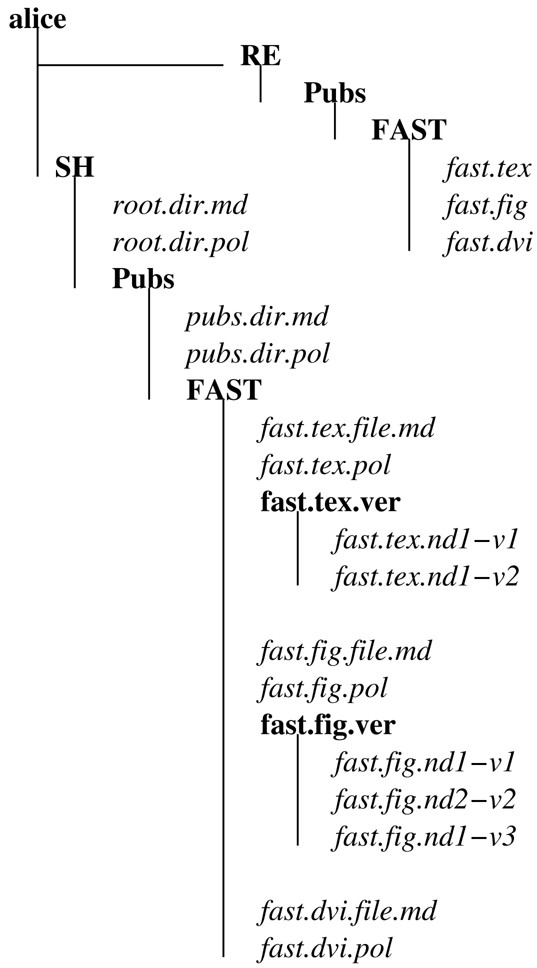

Figure 1: The file's meta-data.

A directory's meta-data consists of two files. The file `dirname.dir.md` stores (1) directory state information and (2) a list of name entries annotated with the time they were created or removed. The file `dirname.dir.pol` contains (1) a list of nodes that are interested in the directory and (2) a list of replica nodes.

A file's meta-data consists of two files and a directory. The file `filename.file.md` stores (1) the file's current state, whether it is present or deleted, (2) the file's current and previous owner and lock status, (3) whether the file is up to date, and (4) a list of file version information as described below. The file `filename.file.pol` stores (1) a list of interested nodes, (2) a list of replication nodes, and (3) the file's availability policies and groupings. The replication server 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. There is also a `filename.ver` directory that is associated with every file that stores file versions.

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 (and beyond); 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. This naming scheme is used to enable us to rebuild a file's history when parts of it are unavailable due to node failures.

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

Figure 2: Branching file history.

4 Performance

4.1 Experimental Setup

Unless otherwise noted, all numbers reported in this section are the median of 1000 trials on otherwise unloaded machines and network. Pentium II PCs running at 266MHZ with 128MB of ram were used for both the client and the server machines. They were connected by a 100Mb ethernet network.

4.2 Opening the Current Version of a File

To open the current version of a file, a Mammoth node must first acquire a read or write lock. The overhead for this operation breaks down as follows: (1) locate lock owner, (2) request lock, (3) invalidate other locks, and (4) grant lock. If the request results in a change of lock ownership (i.e., a write request to a remote owner), the identity of the new owner is propagated as part of the meta-data update sent when the modified file is closed. This overhead is described in Section 4.6.

A requesting node will typically either be the lock owner or will know what node holds the lock, because lock ownership changes are eagerly propagated to interested nodes. Locating the owner thus either requires 71μs or 4740μs. In the rare instance where the ownership forwarding chain is longer, a 4670μs overhead is incurred for every additional node visited.

Requesting and granting the read lock requires 71μs if the owner is local and 120μs if it is remote.

Finally, if the request is for a write lock, then zero or more read locks held by other nodes may need to be invalidated. We measured this case for zero and one reader nodes at 0μs and 426μs respectively. The actual invalidation is asynchronous to the lock request and adds an additional overhead of 1860μs to each lock-holder node. This overhead has no effect on open time, but can impact overall system throughput.

4.3 Reading

Reading the current version of a file should have roughly the same performance as NFS. Open in Mammoth is 690μs, in NFS is 589μs and in *EXT2* is 162μs. Mammoth's open is slightly more expensive than NFS's due to the initialization of several extra data structures during the *lookup* procedure. Sequentially reading a 65536 byte file in Mammoth is 33500μs, compared to 19800μs for NFS and 14500μs for *EXT2*.

To read a random 4-KB block from the server takes 8550μs for Mammoth, 8490μs for NFS, and 8400μs for *EXT2*.

We notice that in Mammoth reading a random 4-KB block is approximately the same as reading the block in NFS and from the local file system. Yet reading a 64-KB file sequentially is considerably more expensive. The reason for this discrepancy is that the first read request is taken as a cue that a file open has occurred and thus extra work, such as opening meta-data, is performed. Subsequent reads are comparable to those in NFS and the local file system.

Operation	Mammoth (μs)	NFS (μs)	EXT2 (μs)
create	245	1270	307
open	690	589	162
read - random 4KB	8550	8490	8410
read - sequential 64KB	33500	19800	14500
write with *trunc* - 1KB	3400	1260	2060
write with *trunc* - 64KB	10000	8040	20300
write - 1KB	2560	1210	8500
write - 64KB	10800	7980	26400

Table 1: Timings for *create*, *open*, *read*, and *write* operations.

Operation	Local Node (μs)	Remote Node (μs)
Lock acquisition	71	120
Lock invalidation	426	1860
Versioning - rename	1990	N/A
Versioning - copy	1330 + 92 per 4-KB	N/A
Meta-data updates - file create	123 + 69 per interested node	9660
Meta-data updates - file update	10 + 69 per interested node	3010
Replication	145 + 102 per 4-KB	417 + 203 per 4-KB

Table 2: Microbenchmarks and overhead for a set of common operations such as lock acquisition and invalidation, versioning, propagating meta-data, and replication. The timings presented are for the sending and the receiving node.

4.4 Writing

Writing to a Mammoth file has extra overhead compared to NFS and *EXT2* because it requires the creation of a new version of the file.

In this experiment we wrote to two different files, one small and one large. In each experiment, we show the elapsed time for creating the new version, performing the described write operation, closing the file, and performing *sync*, to synchronously write the modified data to the server.

To begin, we measured 3400μs to *truncate* the small file, 1024 bytes, and 10000μs to trunc the large file, 65536 bytes, compared to 1260μs and 8040μs for NFS and 2060μs and 20300μs for *EXT2*. Since *truncate* completely erases the contents of a file we are able to use *rename* to version the file. The overhead for doing so is 1990μs.

We then computed the time for writing various numbers of bytes to each file. Writing 1024 bytes requires 2560μs for Mammoth, 1210μs for NFS, and 8500μs for *EXT2*, while a larger write of 65536 bytes takes 10800μs in Mammoth, 7980μs in NFS and 26400μs in *EXT2*. Since we are updating the file we are forced to copy it to create the new version. This versioning technique gives us a constant overhead of 1330μs with an additional overhead of 92μs per-4KB write. From this we conclude that Mammoth is 10 % slower than NFS.

4.5 Andrew File System Benchmark

We ran the Andrew file system benchmark to (1) determine Mammoth's performance with respect to standard NFS and a local file system and (2) to determine the impact of eagerly propagating updates and of replication on the system.

We ran Mammoth with no interested nodes or replication nodes to compare its performance with NFS and a local file system. The total elapsed time for Mammoth was 18.6s, compared to 15.3s for NFS and 12.6s for *EXT2*.

To determine the performance penalty of propagating meta-data we ran a Mammoth server node with 0, 1, 2, and 3 interested nodes and 0 replication nodes, see Figure 3. We ran a Mammoth server node with 0 interested nodes and 0, 1, 2, and 3 replication nodes to determine the cost of replication, see Figure 4.

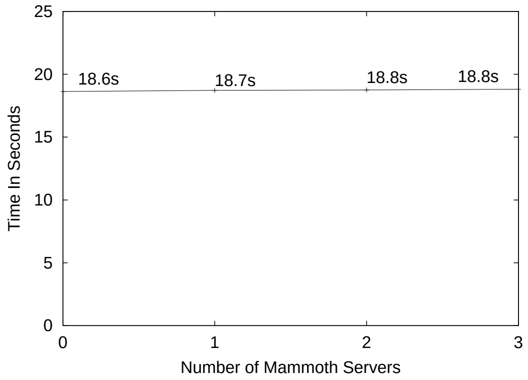

Figure 3: The cost of propagating meta-data. The graph shows the cost of propagating meta-data for 0, 1, 2, and 3 interested nodes using the Andrew Benchmark.

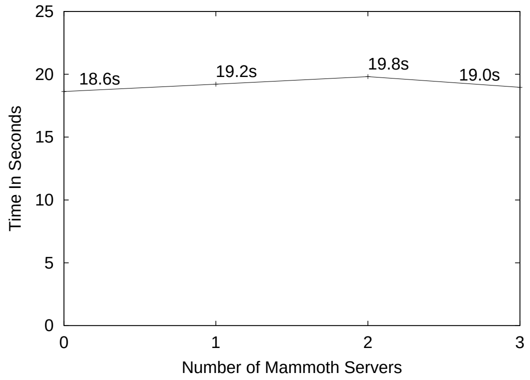

Figure 4: The cost of replicating data. The graph shows the cost of replicating data for 0, 1, 2, and 3 replication nodes using the Andrew Benchmark.

4.6 Propagating Meta-data to Other Nodes

A key feature of Mammoth is that the meta-data cached at *interested* nodes is updated eagerly whenever it changes. We measured the overhead of this update propagation for two operations that create new meta data: (1) creating a file and (2) modifying a file. These two operations are good representatives for other operations such as creating or removing directories and renaming a file. We measured the overhead on both the updating node and on the receiving node.

If there is no interested node, there is no overhead. A file creation consists of creating the update message, 123μs, and sending the message to an interested node, 69μs. The overhead for creating the message is incurred only once for each update. The overhead for sending it is incurred for each interested node. The overhead on the receiving node for a file create is 9660μs. The cost is substantial due to the need to create several meta-data files and directories. For a file update, the cost of creating the update message is 10μs. Creating a file update message is less expensive than a file create message because we need to pack additional information such as the interest list, the replication list, and replication policies into the file create message, which involves additional memory copies. On the receiving end the cost of a file update is 3000μs.

Ideally the cost of propagating meta-data should linearly increase as more nodes register interest in a file. Figure 3 shows the total time it takes for the Andrew Benchmark to run with 0, 1, 2, and 3 interested nodes. We notice that the time increases as the number of interested nodes increases in a linear fashion. The runtime difference between 0 interested nodes and 3 interested nodes is 0.18 seconds.

4.7 Other Operations

There is no associated overhead with registering interest in a file. The request is implicitly piggy-backed on top of a remote read or a remote write request. As explained earlier interest is registered automatically for a file if the access frequency is above a certain threshold. To register interest a single in-memory data structure is accessed thus no measurable overhead is present.

4.8 Replication

There are two different types of replication: replication that occurs when a modified file is closed, and replication that occurs as the result of the background task that runs periodically. In both cases the performance cost is the roughly same. Delayed replication, however, has higher space cost, because updates are not sent immediately but stored and propagated at a later time.

When a version is replicated the overhead for the sender consists of the time it takes to instantiate a file create update message, 123μs, plus an additional 22μs for a total of 145μ. There is also a 102μs of overhead for each 4-KB of file data sent. On the receiving side the overhead to receive a replication message is 417μs plus 203μs for each 4-KB of file data received.

The cost of replicating data should also linearly increase as the number of nodes in the replication group increases. Figure 4 shows the total time it takes for the Andrew Benchmark to run with 0, 1, 2, and 3 replication nodes. For 0, 1, and 2 replication nodes we see a linear increase in replication overhead. For 3 replication nodes we notice that the overhead is less than for 1 and 2 nodes. We believe the replication thread takes more time to execute and thus the system thrashes less although the replication process takes longer to complete. The Andrew Benchmark exercises the main thread while the replication thread runs when there is replication to be done. In the case of 3 replication nodes the replication thread is still running after the Andrew Benchmark finishes.

5 Related Work

Mammoth's use of versions to simplify replication was inspired by the Cedar[1] file system, which used versioning to simplify cache consistency. In Cedar, each version was named by a unique serial number assigned when it was created. In Mammoth, on the other hand, versions are named by any timestamp contained in the interval between its creation and either the creation of the next version or deletion of the file.

Locus [17], Coda [5, 7], and Ficus [14] all use tightly-coupled forms of optimistic replication. In Coda, for example, a tightly-coupled collection of server nodes replicate a portion of the file system. Connected to this are clients that actively cache file data and depend on the servers for synchronization and consistency. Disconnected clients can modify locally cached objects; these changes are reconciled with the servers when clients reconnect. Mammoth differs in that it avoids tight coupling and that it uses version histories to simplify consistency and conflict reconciliation. In contrast, Coda resolves partitioned updates as part of the partition reconciliation process, marking files with conflicts as unusable until fully reconciled by a user or application.

Mammoth's division of reconciliation responsibilities between the low-level storage system — Mammoth — and higher-level reconciliation, contrasts with Bayou [12] and OceanStore [6]. Bayou supports full application-aware reconciliation integrated with a relational database. It uses operation logging to resolve conflicting updates by merging the logs from conflicting nodes, rolling back the database, and replaying the merged log. OceanStore extends this basic idea for very wide-scale storage.

6 Conclusions

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 are stored in append-only fashion and that file data itself is immutable; writing to a file creates a new version. Immutable versions simplify replication and consistency management and increase replication flexibility. Loose connectivity simplifies scalability and fault tolerance, by allowing nodes to have partial, weakly-consistent knowledge of the system and to operate with whatever file versions are currently available, without requiring coordination with unavailable nodes. We believe that this loose connectivity and the simplicity it implies will make the system scalable and easy to administer.

Our design is still in its formative stages and some issues remain to be resolved. To date we have implemented two prototypes: a user-level NFS server and an in-kernel stackable file system in FreeBSD. Our plan, yet incomplete, is to allow clients to access the file system either directly on a Mammoth node or remotely via an unmodified (or mostly unmodified) NFS client; remote access via NFS is fully implemented. Finally, our evaluation of the user-level server presented in this paper shows that performance is reasonable.

Our future work will be on three fronts. First, we continue to refine our design to deal with issues presented in this paper and to develop a complete prototype implementation. Second, we plan to explore the utility of a variety of availability policies and to gain experience from real users. Third, we plan to extend the design to build a Bayou-like middleware service on top of Mammoth to handle application-level conflict reconciliation.

References

[1] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.

[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] M. Ji, E. W. Felten, R. Wang, and J. Pal Singh. Archipelago: An island-based file system for highly available and scalable internet services. In *Proceedings of 4th USENIX Windows Systems Symposium*, August 2000.

[5] 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.

[6] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi,

Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[7] Puneet Kumar and Mahadev Satyanarayanan. Log-based directory resolution in the Coda file system. In *Proc. Second Int. Conf. on Parallel and Distributed Informat ion Systems*, pages 202–213, San Diego, CA (USA), 1993.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marv in M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Sixteenth ACM Symposium on Operating Systems Principles*, Saint Malo, France, 1997.

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

[14] Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the ficus file system. In *USENIX Summer*, pages 183–195, 1994.

[15] 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.

[16] 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.

[17] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th Symposium on Operating Systems Principles, Operating Systems Review*, pages 49–69, October 1983.

[18] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 71–78, October 1993.

[19] 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.