Using File-Grain Connectivity to Implement a Peer-to-Peer File System

Dmitry Brodsky, Alex Brodsky, Jody Pomkoski, Shihao Gong, Michael J. Feeley, and Norman C. Hutchinson

Department of Computer Science
University of British Columbia

{dima,abrodsky,jodyp,shgong,feeley,norm}@cs.ubc.ca

Abstract: *Recent work has demonstrated a peer-to-peer storage system that locates data objects using $O(\log N)$ messages, by placing objects on nodes according to pseudo-randomly chosen IDs. While elegant, this approach is constrains system functionality and flexibility: files are immutable, directories and symbolic names are not supported, data location is fixed, and access locality is not exploited. This paper presents Mammoth, a peer-to-peer, hierarchical file system that, unlike alternative approaches, supports a traditional file-system API, allows files and directories to be stored on any node, and adapts storage location to exploit locality, balance load, and ensure availability. Our approach is to handle all coordination at the granularity of files instead of nodes. In effect, the nodes that store a particular file act as its server independently of other nodes and files in the system. The resulting system is highly available and robust to failure. Our experiments with our prototype have yielded good results, but an important question remains: how the system will perform on a massive scale. We discuss the key issues, some of which we have addressed and others that remain open.*

1 Introduction

Recent interest in peer-to-peer storage systems is motivated by the realization that tight-coupling and excessive coordination, used in traditional distributed systems, are an obstacle to scalability. Current research has demonstrated an intriguing mechanism that locates file data using $O(\log N)$ messages among N peer node [4, 10]. Each node is assigned a quasi-random ID; similarly, files (or blocks) are assigned an ID taken from the same space and are stored on nodes with numerically closest IDs. The key to tolerating failure is the random assignment of node IDs; this ensures that nodes with similar IDs are unrelated to each other in the underlying network topology. Data replicated on nodes with consecutive IDs is thus reasonably invulnerable to a single point of failure.

The compelling simplicity of this approach comes with endemic limitations. First, files are immutable and, symbolic names and directories are eliminated to avoid synchronization and consistency issues. Second, a file's (or block's) location is fixed when it is first added to the system. This constraint allows the file to be located from any node efficiently, but also has performance drawbacks. A new node immediately takes responsibility for all of its neighbor's files whose ID is numerically closer to its own, (roughly half the files on each adjacent node): these files must be copied to the new node. Load balancing and caching is also complicated because storage location takes no advantage of access locality. Client-side caching is impossible because clients with network proximity to each other may follow largely disjoint paths to objects they that share. Finally, while the $O(\log N)$ messages is small compared to N, the cumulative latency of locating a file is $O(\log N)$ times the average access latency in the network, which for the Internet can be quite large.

This paper describes the design of an alternative peer-to-peer approach, named *Mammoth*, that provides the functionality of a traditional file system while preserving the scalability and the benefits of the peer-to-peer approach. Unlike other peer-to-peer storage systems, Mammoth allows files and directories to be stored on any node and adapts storage location dynamically to exploit locality, balance load, and ensure availability.

Our approach is to handle all coordination at the granularity of files instead of nodes. Each directory and file in the system is stored on at least one node and may be replicated on additional nodes for performance or availability reasons. Per object policies are used to specify how an object is stored and replicated. These policies induce inter-node dependencies that are not present in existing peer-to-peer storage systems.

We have implemented a Mammoth prototype as a user-level NFS server. The prototype currently runs on Linux and Solaris, and is easily portable to other POSIX-compliant platforms. We are preparing a publicly available prototype for release.

The remainder of this paper describes the design of Mammoth, the additional inter-node dependencies, and the additional communication complexity induced by the per object policies. We present an overview of the system and describe the policy mechanisms. Lastly, we discuss how Mammoth operates in a wide area environment, how it deals with failures, and some of the ramifications of setting particular policies.

2 Mammoth

A Mammoth file system consists of a collection of peer-to-peer nodes that cooperate to store a hierarchical file system. Each file or directory is stored on one or more nodes, but no node stores everything. An object's storage location is chosen adaptively to provide access locality and to ensure high availability. Files are replicated in the background according to flexible, user-controllable, per-file policies. Consistency is ensured using locks during normal connectivity and switches to an optimistic approach when failures occur. Nodes are allowed to read and write whatever data they can currently access. Eventual consistency is simplified by storing directory and file metadata as logs and by storing file data as a collection of immutable versions. Write conflicts are represented in the metadata as version history branches, and must be resolved by higher-level software or the user.

2.1 Directory and file metadata

The key data structures that connect nodes to each other in Mammoth are directory and file metadata. Until a node stores a file or directory, it knows nothing about the rest of the file system (apart from its well-known root nodes) and the system knows nothing about the node. Adding a node simply involves storing a file or directory there, updating only that object's metadata to reflect the identity of this new storing node.

The format of Mammoth metadata is similar to that of a typical UNIX file system (e.g., directories and inodes), with three differences. First, each object's metadata lists the nodes that host it; nodes are named by their network address. Second, the metadata updates are stored as a timestamped change log. Third, file data, which is separate from metadata, is named symbolically and can be stored on a different node from the metadata.

The *interest set* is the set of nodes that store the metadata of a directory or a file. Multi-node interest sets are used to replicate metadata. One node in each interest set is the object's *owner* and is the only node allowed to modify the object. Directories are modified via a remote procedure call to the owner. Files, on the other hand, are modified by moving ownership to the updating node, thus optimizing for update locality common to files. In both cases, the owner synchronizes updates and sends them to the object's other interested nodes.

File and directory metadata is organized as a change log with each entry timestamped by the originating node, thus inducing an order on the entries. This design allows metadata updates to be applied in any order. Consistency is thus ensured as long as updates are eventually delivered to every interested node, even when some updates are delayed by a node or network failure.

Finally, unlike traditional UNIX directories, which map names to inode numbers, in Mammoth both metadata and data are referenced symbolically. This approach decouples data from metadata and presents a uniform reference model for both remote and local objects. Objects — files and directories — are named internally by a globally unique identifier (GID) comprising the creator node's network address and a local timestamp. File data is named by a tuple comprising the file's GID and the data's version timestamp.

2.2 File data

File data is stored as a journal of immutable file versions. A file's metadata tracks these versions so that they can be located when needed. Each entry comprises the version's timestamp, the timestamp of its history branch, and the network addresses of nodes that store it. Versions are timestamped by tuples consisting of the address of the node that created them and their creation time on that node. The history branch timestamp is explained in Section 3.5.

File versioning is done for two reasons. First, consistency issues associated with replication are confined to the file's metadata. Version immutability ensures that versions can be independently replicated without becoming inconsistent. Second, the system can avoid replication of some versions as a performance optimization; in Mammoth, only selected versions are replicated. In contrast, in a typical replicated file system, each update to a file is replicated using a consistency protocol that ensures that replicas see file operations in the same order.

2.3 Replication

An important goal of Mammoth is to ensure high availability and to protect data from all forms of failure, without requiring regular manual intervention. Since not all files are created equal, not all require the same level of protection against failure. Realistically, users require protection from multiple failure modes that differ in their likelihood and avoidance cost. For example, protection from node failure can be handled by replicating data to a nearby node. Protection from network failure or natural disaster, however, requires replication to a distant node that is unlikely to be affected by the failure.

Our solution is to support a set of replication policies to address the majority of such failure modes. The policies have the general form of placing an upper bound on the number of hours of work that could be lost for a given failure. Users, administrators or the system assign policies to files based upon the value of the data. The system then implements the object's replication policy using a background process that tracks recently modified objects to determine when they should be replicated and to what nodes.

3 Dealing with failure

Mammoth is designed to be robust in the face of intermittent node and network failure. It achieves this goal using an optimistic replication approach that allows nodes to read and write any version of data that can currently access, even if more current versions exist, but are currently inaccessible. This section briefly describes how the system handles various types of failures.

3.1 Owner failure

Owner failure is detected whenever a node, which owns a particular file, fails to respond to a request regarding the file. When a node detects this failure it initiates an election among the remaining nodes in the interest set. The newly elected owner is responsible for completing all duties that were left unfinished; including the propagation of any notifications that did not complete. Such information is gathered during the election process from the remaining nodes in the interest set. If the owner failure was due to a network partition, multiple owners can arise, resulting in potential inconsistencies; the mechanism for resolving these inconsistencies is described in subsection 3.5.

Failure can also cause the current version of the file to be inaccessible since the current version is always stored on the current owner. If the current version is inaccessible the accessing node retrieves an older version by consulting the object's metadata.

3.2 Metadata eventual consistency

Metadata in Mammoth is stored only by nodes in the interest set of a given object and is organized as an append-only log of timestamped metadata entries, as described above. The state of an object depends only on the entries in the log, and not on their order. As a result, the only way for metadata to be inconsistent is if the log is missing some entries. Provided that a failure is transient, any missing entries will eventually be propagated to the node and will be appended to the log, resolving the inconsistency.

Update propagation is a two phase process. The owner first sends an update to each node in the interest set. Each node that receives the update starts monitoring the liveliness of the owner. In the event that the owner fails, a node in the interest set that has already received the update takes over the responsibility of propagating the update to the other nodes.

If the owner can not deliver the update to a node, the owner monitors the node and delivers the update when connection is re-established. If the node remains inaccessible for a sufficiently long period, the owner deems the failure permanent and follows the procedure outlined in Section 3.4 to remove the node from the object's interest set.

Once the owner has propagated the update to all interested nodes, it sends a second message to these nodes that allows them to retire the update and stop monitoring the owner for liveliness. This procedure will eventually deliver updates to all interested nodes, and thus ensure eventual consistency, as long as every node has a consistent view of the object's interest set.

3.3 Inaccurate interest sets

Interest sets may become inaccurate during a network partition because our design favours availability over consistency: a node may be elected owner of an object whenever the current owner is unreachable.

Inaccuracies in interest sets arise if the interest set changes during a network partition. If this happens, each partition will have different interest sets for the object, and thus metadata updates will not be fully propagated to all interested nodes, even when the network partition heals. When the partition heals this inconsistency is easily detected by any of the nodes in the intersection of the divergent interest sets. Such a node will contact both owners, initiating a reconciliation that causes all metadata updates to be propagated to a new merged interest set, which comprises the union of the divergent interest sets.

If the partitioned interest sets diverge to the point that their intersection is empty, a different mechanism is used. Since this situation only occurs when all nodes in the intersection of the partitioned interest sets have *permanently* failed, this case is handled as a special case of permanent failure, which is described in the next section.

3.4 Permanent failure

The permanent failure of a node is detected when another node has held updates for it that have been undeliverable for a long time. In this case, the node with the unpropagated update removes the failed node from the updated object's interest set and informs the other interested nodes to remove the update from their differed-update tables. Any updates to this object made by the failed node that were not propagated to at least one other node are now permanently lost.

Such interest-set removals are registered centrally to resolve the partitioned interest-set problem we just described. The table is replicated on a subset of the nodes in the root directory's interest set and is indexed by a pair consisting of the object's GID and the address of the failed node. An entry stores an object's interest set at the time the failed node was removed. If an interest-set partition exists, multiple nodes will register the same removal, but with different interest sets. In this case, the registry updates the object's interest set to the union of the registered interest sets, thus joining the possibly disjoint interest sets. An entry is no longer needed and can be removed when no node will subsequently detect the permanent failure. Holding an entry for one full permanent-failure-timeout period is sufficient, because any node that takes more than this amount of time to detect the failure will have itself been deemed to have permanently failed.

3.5 File update conflicts

Update conflicts occur during a network partition when concurrent modifications to the same object are performed on both sides of the network partition. In the simple case, two or more owners may exist simultaneously, but no object modifications occur. In this case, once the partition heals, the system has no real inconsistencies to resolve since no modifications to the object occurred; only the interest set and owner may have changed. This situation is resolved by demoting one of the owners and sending out updates to all the nodes in the merged interest set. Unfortunately, if the object was modified concurrently, object inconsistencies occur that cannot be directly resolved.

As in any optimistic replication scheme, update conflicts create inconsistencies in files or directories. In Mammoth these conflicts can occur in two situations when there are multiple owner nodes or when the current version of a file is inaccessible and a node retrieves and modifies a previous version. The goal of Mammoth is to ensure eventual consistency of the logs used to store directory and file metadata, and not to resolve the branches in the version history. Mammoth leaves the resolution of conflicts in file data to higher-level software or to users. If such a conflict arises, it is stored as a branch in the version history of the directory or file. This history is visible to users, but a node that accesses a file or directory on a particular branch will by default continue to access that branch of the history. Users or application-level tools can inspect these histories and reconcile conflicts by merging branches.

A file version's immediate predecessor is normally determined by chronological timestamp order. If a conflicting update occurs during a period of disconnection and history branches are created, timestamp-order alone is insufficient to capture this relationship. In this case, the branch timestamp field in each version entry, which uniquely names the branch to which the version belongs, is used in conjunction with the version timestamp. New branch timestamps are assigned whenever a new owner is elected.

3.6 Failure and replication

The permanent failure of a node that stores replicated data typically requires the re-replication of that data to preserve availability requirements specified by the associated availability policies. In Mammoth this re-replication is handled by the nodes that store replicas.

Each node maintains summary information about the replicated data it stores. Included in this formation is a list of the other nodes that also replicate the same data. Should a replica node fail, an election is initiated among the other nodes that share replicas with the node. This step is designed to ensure that as few nodes as possible proceed to the next step. Each node that has been elected selects a new replica node for the affected objects and sends its copies of these objects to the new replica node. Finally, the replica node sends a message to every node in the affected object's interest set informing it of the change in replication nodes. These messages are batched when possible to minimize message overhead. During this process the other replica nodes continue monitor this node and call a new election should this node fail.

3.7 Monitoring node liveliness

Mammoth nodes monitor other nodes for liveliness in three cases. First, when attempting to propagate an update to an interested node that appears to be down. Second, when processing an update that has not been fully propagated. Third, when storing replicated data. In each case, the nodes in question are registered with the liveliness module along with an upcall procedure that is triggered when the node's status changes. The monitor determines liveliness by monitoring all in-bound and out-bound messages and pinging nodes when necessary.

4 Understanding global performance

Mammoth's file-grain approach to coordination provides tremendous flexibility. It allows any number of replicas of any file or directory to be stored anywhere. This flexibility presents both opportunities and risks. The risk is that bad placement decisions may degrade system performance by increasing the dependencies among nodes unnecessarily. As the system grows larger, the impact of these bad decisions will accumulate and the ability of humans to take corrective action will degrade. It is thus crucial for a peer-to-peer system to administer itself to indemnify itself against these risks.

The crux of the problem is that Mammoth's file-grain connectivity implicitly introduces node-grain dependencies. When a significant system event such as node or network failure or recovery occurs in the system, the actions of the individual nodes concerning individual files as described in this paper appear to be both necessary for correctness and reasonable in cost. The number of nodes and files that may be impacted by a single system event, however, may be very large.

To understand the global impact of significant system events it is necessary to understand the topology of the system: the connections and dependence among nodes. In Mammoth these connections are defined by the interest and replication sets of directories and files. ¿From the outset Mammoth was designed to instantiate these connections as required. Initially, a node starts as a singlely-connected component, and as requests from clients to add the node to an interest set or to replicate a file on that node are processed, additional connections are established. Good global performance, however, requires the system to control this growth to guide it away from problematic topologies.

We do not yet have a precise description of what constitutes a good or bad topology; the issue, however, boils down to the formation of cliques. If node interconnections tend to form cliques, the impact of a single event will be mostly confined to the clique and thus global performance will not be threatened.

The remainder of this section describes our current approach for encouraging clique formation of replication and interest sets. While we believe that we are on the right track, evaluation of our ideas is difficult, because doing so empirically requires the construction of very large Mammoth file systems. As future work, we plan to investigate both formal and experimental evaluations.

4.1 Deciding where to replicate

Deciding where to replicate an object is the first key issue affecting global performance; there are two constraints. The first constraint is that the replica nodes used by an object must satisfy its availability policies. This may require locating nodes that are in different buildings or are in completely different locations in the underlying network topology.

The second constraint is that recovering from the failure of a replica node should be efficient. Replica nodes monitor each other to ensure that enough replicas are available. When a long-term failure is detected the nodes that share replicas with the failed node initiate a process to re-replicate these objects. The overhead of this process is determined by the total number of nodes affected by the failure of the replica node. In the worst case, every object on the failed node could be replicated on one distinct other node, thus requiring at least one message for every replicated object. If, on the other hand, the total number of affected nodes is small, the overhead is constrained.

The above constraints are enforced when a replication set is assigned to an object. To assign a replica set to an object, a node A consults its local database to choose a single replica node. A chooses node B arbitrarily from among a set of candidate nodes that satisfy the object's policy. A then sends a message to B requesting that B accept replicas for the object and that B choose the other nodes that will replicate the object. When choosing the other nodes to replicate the object, B consults its list of the replicas it already stores. The list names all other nodes that share replicas with B; these are the nodes that will be affected should B fail. B selects additional replica nodes so that the number of nodes in this list small; these nodes thus form a replica clique. If three or more replica nodes are required to satisfy a policy, B consults these other nodes offering a list of acceptable candidates. These nodes select the candidates from this list that also tend to keep this number of nodes they share replicas with small. This process is simplified by the fact that objects can adequately be protected from failure by keeping only a few replicas, typically two or three.

4.2 Deciding when to cache metadata

The second key issue affecting global performance is to control interest-set additions and removals. Recall that nodes are placed in an interest set to exploit access locality. The access time of an object that is frequently accessed by a node is improved if the node is added to the object's interest set because most accesses become local. In addition, access time is also improved if frequently read objects have large interest sets, because the read load is distributed among these nodes; this is important for directories near the root of a large Mammoth file system, for example. On the other hand, increasing the size of an object's interest set increases the cost of updating the object, because updates must be propagated to all interested nodes. The costs of failure also increase because the probability that some interested nodes will be down during an update also increases.

We believe that an adaptive policy can strike a good balance between these concerns. Our first order approach is to use a simple frequency based watermark scheme to determine when to remove or add a node to the interest set. Access frequency statistics used to implement this policy are collected by owner nodes.

5 Open issues

Our prototype implementation is nearing maturity and we have conducted small scale experiments involving a dozen or so nodes that indicate that it performs well in these cases. The system is designed to scale massively, however, and we have yet to evaluate its performance at this level. Four key open issues remain to be investigated.

- Interest and replication sets make the connections among objects explicit. It is not clear, however, how difficult it will be to infer the global performance implications of these connections from only this file-grain information.
- A node that stores a large number of files with mostly non-overlapping interest sets is dependent on a large number of nodes. This topology prevents metadata-update message batching and thus increases message load, particularly in the even of the node's failure. While we are able to limit the number of interest sets a node belongs to, we don't fully understand how to control the dispersion of interest sets.
- We have not yet addressed the issue of consistent checkpoints in Mammoth. Given that we version all data, it is likely that users may want to create a consistent checkpoint for a set of files. We plan to support this feature, by allowing users to specify files that should be replicated as a group. It is unclear, however, how these groups should be specified and what performance implications there may be for creating large groups.
- Our approach to dealing with long-term failure requires centralization to ensure that divergent interest sets cannot prevent eventual consistency. This design represents a particular tradeoff between flexibility and implementation complexity. It is not clear whether it is the best tradeoff, however.

6 Related Work

Recently, a large body of work has been done in the area of peer-to-peer storage systems. Systems like Gnutella [5] and Napster [8] were devised for the primary reason of sharing information. While other systems such as FreeNet [3] and Eternity [1] were designed to function as deep archival repositories. CFS [4] is a read-only peer-to-peer file system that operates at the block level. To improve the latency associated with the

$O(\log N)$ lookup cost, blocks are cached on nodes that are on the path to the node that stores the primary copy. PAST [10] was designed to be a general purpose replicated object store.

Systems such as Coda [7], Echo [2], Ficus [9], JetFile [6], and Locus [11] bridge between traditional distributed file systems and peer-to-peer storage systems. They use techniques from both camps. Many of these systems still rely on tightly coupled nodes and or partitioning the file system to achieve the desired robustness and scalability.

7 Conclusions

Mammoth is a peer-to-peer file system that provides clients with a traditional UNIX-like API while also providing the scalability and sharing benefits exhibited by peer-to-peer storage systems such as CFS and PAST.

The key idea of Mammoth is that all inter-node coordination is handled at the granularity of files. This approach allows the system to scale arbitrarily, as long as the number of nodes that store a particular file is not too large. It also simplifies system design. To make the system robust to failure, for example, files and directories are replicated in an optimistic fashion by a background process. When the nodes that share an object are connected to each other, consistency is ensured by using one of these nodes to coordinate updates. In the event of failure, eventual consistency is simplified by logging metadata updates and versioning file data.

Along with these benefits comes a key issue. How will Mammoth's file-grain interconnections impact performance when viewed more globally, at the granularity of nodes? The relationship between these two levels of granularity is complex, because each node may store a huge number of files. Understanding and controlling the performance of Mammoth thus requires viewing the system at both granularities.

Our approach is to devise policies that guide the establishment of file-grain connections to ensure good node-grain performance. These policies encourage the formation of cliques of nodes, which we believe will tend to limit the number of nodes (and thus messages) that can be affected by a single event such as a node failure or network partition. The effectiveness of these policies has not yet been evaluated, however, and a number of open issues remain. We believe that the potential benefits of this approach to peer-to-peer storage are compelling and that the answers to these questions are thus important. To this end, we are making our prototype implementation publicly available and are pursing both formal and experimental approaches to gain further insight.

References

[1] R. Anderson. The eternity service, 1996.

[2] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corporation, Palo Alto, CA, USA, October 1993.

[3] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[4] Frank Dabek, M. Frans Kasshoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, October 2001.

[5] Gnutella. http://www.gnutella.com.

[6] Bjorn Gronvall, Assar Westerlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Operating Systems Design and Implementation*, pages 251–264, 1999.

[7] 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.

[8] Napster. http://www.napster.com.

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

[10] Antony Rowstron and Peter Drushel. Storage management and caching in past, a large-scale, persistent peer-to-peer stroage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

[11] 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.