Efficient Metadata Management in Large Distributed Storage Systems[†]

Scott A. Brandt *scott@cs.ucsc.edu*
Ethan L. Miller *elm@cs.ucsc.edu*
Darrell D. E. Long *darrell@cs.ucsc.edu*
Lan Xue *lanxue@cs.ucsc.edu*

Storage Systems Research Center
University of California, Santa Cruz

Abstract

Efficient metadata management is a critical aspect of overall system performance in large distributed storage systems. Directory subtree partitioning and pure hashing are two common techniques used for managing metadata in such systems, but both suffer from bottlenecks at very high concurrent access rates. We present a new approach called Lazy Hybrid (LH) metadata management *that combines the best aspects of these two approaches while avoiding their shortcomings.*

1. Introduction

In large distributed storage systems, avoiding bottlenecks is critical to achieving high performance and scalability. One potential bottleneck is metadata access. Although the size of metadata is generally small compared to the overall storage capacity of such a system, 50% to 80% of all file system accesses are to metadata [12], so the careful management of metadata is critical. We present *Lazy Hybrid (LH)* metadata management, a new metadata management architecture designed to provide very high-performance, scalable metadata management.

Traditionally, metadata and data are managed by the same file system, on the same machine, and stored on the same device [9]. For efficiency, metadata is often stored physically close to the data it describes [7]. In some modern distributed file systems, data is stored on devices that can be directly accessed through the network, while metadata is managed separately by one or more specialized *metadata servers* [5].

We are developing LH in the context of a large high-performance object-based storage system [17]. Object-based storage systems separate the data and metadata management as depicted in Figure 1. Semi-independent object-based storage devices (OBSDs) manage low-level data storage tasks such as request scheduling and data layout, and present a simple object-based data access interface to the rest of the system. A separate cluster of metadata servers manage the namespace and directory hierarchy, file and directory permissions, and the mapping from files to objects. The metadata server cluster is otherwise not involved in the storage and retrieval of data, allowing for very efficient concurrent data transfers between large numbers of clients and OBSDs.

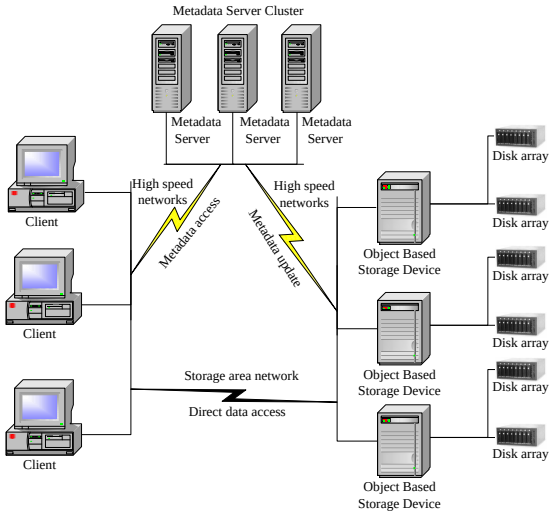

Figure 1. Storage system architecture

The goal in systems with specialized metadata management is to efficiently manage the metadata so that standard directory and file semantics can be maintained, but without negatively affecting overall system performance. This includes handling large numbers of files ranging from bytes to terabytes in size, supporting very small and very large directories, and serving tens or hundreds of thousands of parallel accesses to different files in different directories, different files in the same directory, and even to the same file. A key question in the design of such a system is how to

[†]This research is supported by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract 520714.

This paper will appear in the *20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, CA, April 2003.

partition the metadata among the metadata servers to provide both high performance and scalability.

Currently, popular approaches to metadata allocation employ one of two techniques. The first, which we call *directory subtree partitioning*, partitions the namespace according to directory subtrees [6]. In directory subtree partitioning, the metadata of complete directory trees is managed by individual metadata servers, analogous to directory mounting in NFS [14]. This technique suffers from severe bottleneck problems when a single file, directory, or directory subtree becomes popular. Furthermore, the directory hierarchy must be traversed to determine the permissions for each file that is accessed. This is often mitigated somewhat by client-side prefix caching. However, prefix caching doesn't help when large numbers of clients simultaneously access the same file or directory.

The second metadata allocation technique, which we call *pure hashing*, uses hashing to widely distribute the namespace among the metadata servers [4]. Pure hashing assigns metadata to metadata servers based on a hash of the file identifier, file name, or other related values. This results in more balanced workloads than directory subtree partitioning. However, a directory hierarchy must still be maintained and traversed in order to provide standard hierarchical directory semantics and to determine access permissions, negating some of the apparent benefits. If the hash is based on the full pathname, a good candidate since it may be the only information the client has about the file, then a large amount of metadata may have to be moved when a directory name is changed. This is the result of the hash output – indicating which metadata server to store the metadata on – changing as a result of the changed input. If the hash uses only the filename (instead of the full pathname), as is done in Lustre [3], then files with the same name will hash to the same location, even if they are in different directories. This would lead to a bottleneck during a large parallel access to different files with the same name in different directories.

Both directory subtree partitioning and pure hashing encounter difficulties when adding metadata servers to or removing them from the cluster. Under both techniques, a disproportionate amount of metadata may have to be moved. In directory subtree partitioning this is because whole subtrees are stored on each metadata server, requiring the entire namespace to be repartitioned to maintain a balanced workload. In pure hashing, the hash function itself will have to change to produce outputs in a different range, possibly requiring almost all of the metadata to move to a new server as indicated by the new hash function.

Lazy Hybrid metadata management combines the best of both hierarchical directory subtrees and pure hashing. LH addresses the problems mentioned above using a combination of hashing, hierarchical directories, lazy metadata relocation and lazily updated access control lists (ACLs). The location of metadata for individual files is determined by a hash on the full pathname, distributing metadata and avoiding "hot spots" where a disproportionate percentage of active metadata requests are to a single server. Hashing the pathnames allows direct access to file metadata without involving all of the metadata servers storing directories along the path. Hierarchical directories are maintained in order to provide standard directory semantics and operations (such as *ls*). Lazy update policies allow for efficient metadata updates when file or directory names or permissions are changed or when metadata servers are added to or removed from the system. Finally, a unique dual-entry Access Control List structure allows file permissions to be determined directly, without traversing the entire path.

LH is designed to meet the following general goals:

- **High performance**
  The metadata server cluster must provide very fast metadata access. A typical request should involve a single message to a single metadata server. The metadata server cluster should:
  - support very large numbers of parallel accesses to a single file, directory, or directory subtree;
  - handle very large numbers of files, with extremely high variance in size, ranging from a few bytes to a few terabytes;
  - efficiently manage very large directories containing tens of thousands of files, partitioning them over multiple metadata servers to avoid hot spots.
- **Scalable**
  Metadata performance should scale with the number of metadata servers in the cluster.
- **Flexible**
  The metadata server cluster should efficiently support directory renaming, file and directory permission changes, and the addition, removal, and replacement of metadata servers.
- **Uniform namespace**
  All clients should have the same view of the directory tree.
- **Standards compliance**
  We want to provide a standard interface that will make the storage system useful in existing systems. Our overall Object-based Storage System, in which LH will be one component, will export a POSIX-compliant interface.

2. Related Work

Directory subtree partitioning and pure hashing are methods used to distribute the namespace and workload among metadata servers in existing distributed file systems. Directory subtree partitioning provides a natural way to partition the namespace among multiple servers in distributed systems. Each server manages one or more sub-tree(s) (also called file sets or volumes) of the hierarchy. LOCUS [15], NFS [13], AFS [9], Coda [16], and Sprite [11], partition the namespace using this technique.

One advantage of directory subtree partitioning is that metadata for a file can generally be accessed by contacting relatively few metadata servers, as a particular metadata server will store some or all directories in the path of a given file. Directory subtree partitioning, together with prefix caching, allows for relatively efficient metadata access if the same directories are accessed repeatedly by the same client. The major disadvantage of directory subtree partitioning is that the workload may not be evenly balanced among the metadata servers, leading to a system bottleneck and resulting in lower overall performance. When a file group in a directory subtree becomes popular, the server on which the subtree resides may be disproportionately busy, causing longer response times or even dropped requests. Replication addresses this problem to a certain degree, and may be employed in LH to alleviate bottlenecks involving many clients simultaneously opening the same file. In general, replication can involve significant storage overhead and managing consistency among the replicas can result in additional performance overhead. Adding or removing metadata servers is also costly with this partitioning scheme because the partitions must be carefully crafted and whole subtrees of metadata must be moved from one metadata server to another.

Hashing eliminates the problem of unbalanced workload among servers. For example, Vesta [4] and InterMezzo [2] use pathname hashing for both data allocation and location. If a hierarchical directory structure is used then the directories themselves may still be hot spots, even if the metadata for the files they contain is widely distributed. If a hierarchical directory structure is not used, then an alternative naming scheme that mimics directory hierarchies and directory-based permissions must be employed. Vesta does not address this question, and it is not trivial as it involves other performance trade-offs. Hash-based namespace partitioning also has trouble when the number of servers changes—the hash functions used may have to change and large amounts of metadata may need to be moved. Unless a hash update can be found that moves only the data that really needs to move, the overhead is tremendous.

In RAMA [8], a file system designed for parallel computers, metadata is divided into positional metadata and intrinsic metadata. Positional metadata tells the operating system where to find the data of the files. Intrinsic metadata describes the data in the file, which includes file modification time, access time, ownership and file sizes. By hashing on the global file identifier, the positional metadata can be located. Nevertheless, because intrinsic metadata is stored at the start of a file's data, RAMA has the trade-off of slow directory operations such as *ls*.

Since the idea of specialized metadata management in large distributed file systems is relatively new and several such systems are proprietary, there has been relatively little published work on metadata management in specialized metadata server clusters. Lustre [3] is a non-proprietary Object-based Storage System that uses a hash on the tail of the filename and the identifier of the parent directory to determine which metadata server will store the metadata for a given file, but must use hierarchical directory traversal to retrieve the metadata. The strategy Lustre uses for metadata allocation and access has the advantage of distributing the metadata, but all of the other inefficiencies of hierarchical schemes. LH goes a step further, eliminating these inefficiencies through a combination of all the techniques of hashing, caching, dual ACLs, and lazy updates.

3. LH Design

Like pure hashing, LH uses hashing to distribute the metadata across the metadata server cluster. However, LH also maintains hierarchical directories to support standard directory hierarchy and permission semantics. LH uses pathname hashing for metadata allocation and location, avoiding the overhead of hierarchical directory traversal, and maintains hierarchical directories to provide *ls* and other directory operations.

To access data, clients hash the pathname of the file to produce a hash value indicating which metadata server contains the metadata for that file. The client then contacts the appropriate metadata server to open the file and obtain the file-to-object mapping. The result is extremely efficient metadata access, typically involving a single message to a single metadata server.

When a metadata server is contacted, one of three situations may occur. One is that the metadata exists on the server and the client has permission to do the requested operation. In that case, the operation is completed and the client is given the information needed to obtain the file data from the OBSDs. Another possibility is that the metadata exists on the server and the client does not have permission to do the requested operation. In that case, a response is sent to the client indicating that the requested operation is

not permitted. A third possibility is that the metadata does not exist, indicating that the file itself does not exist. In that case, the appropriate response is sent to the client.

3.1. Metadata Look-up Table

Rather than using the hash value directly to indicate which metadata server to contact, LH uses the hash value as an index into the *Metadata Look-Up Table (MLT)*. The MLT is a small and infrequently updated global table available to all clients and metadata servers that provides an additional level of indirection between the clients and the metadata servers. The index found in the entry of the MLT specified by the result of hashing a filename indicates which metadata server should store the metadata for that file.

The use of the MLT makes the addition and removal of metadata servers straightforward by the simple modification of one or more entries in the MLT. When a metadata server is added or removed from the system, the MLT is updated and broadcast to all of the metadata servers. Client file systems get the MLT from the metadata server cluster by contacting a designed "master" metadata server at a well-known address during client file system initialization. Clients are also sent updated MLTs as needed. When a client contacts a metadata server, it includes which version of the MLT it used. If the contacted metadata server determines that the client has an out-of-date MLT, it will respond with the updated MLT. If the metadata server is the correct one (in spite of the updated MLT), then the metadata server also sends back the appropriate response to the client request. If the metadata server no longer contains the requested metadata, the client can reattempt the transaction with the correct metadata server. If the metadata server listed in an outdated version of the MLT no longer exists, the client can contact the "master" metadata server for an updated MLT after the unsuccessful attempt to contact the nonexistent metadata server.

3.2. Access Control

Traditional directory-based access control checks permissions by traversing the pathname and checking the permission for each directory along the path. Without modification, conventional access control is incompatible with hashing. By hashing to the current location of the file, the directory path is bypassed entirely, and the permissions along that path cannot be checked. Systems using pure hashing to partition the namespace must therefore traverse the entire path in order to determine the appropriate permission. This incurs as much overhead as directory subtree partitioning, even though the metadata can be located directly by hashing. It can also lead to bottlenecks if there are many concurrent metadata accesses to files in the same directory or directories.

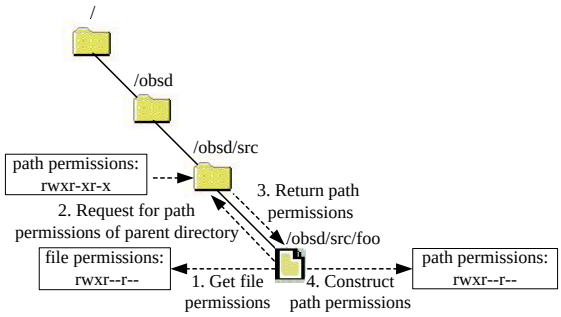

Figure 2. Example of creating ACLs.

To avoid this overhead, LH uses a unique dual-entry access control list (ACL) structure for managing permissions. Each file or directory has two ACLs representing the *file permissions* and the *path permissions* for the file or directory. The file permissions are the ACL for the object itself. The path permissions are essentially the intersection of the file permissions and the parent directory's path permissions – yielding an ACL that represents the permissions that would be obtained by first visiting the parent directory, and then visiting the file itself. As the parent directory's path permissions were similarly constructed, the path permissions represent the permissions that would be obtained by following the entire directory path down to the file. Constructing the path permissions consists of a straightforward intersection of the two permissions while accounting for the slight differences in semantics between file and directory permissions. Path permissions are created at the time that the file is created, and updated whenever there is a change to the file permissions of a directory in its path. Figure 2 shows an example of constructing the ACLs for */obsd/src/foo*.

3.3. LH Efficiency

As a result of the file pathname hashing, MLT indirection, and path permissions, LH allows for extremely efficient metadata service. Most metadata requests will require a single message to a single metadata server. In general, neither the client nor the metadata server cluster need traverse the directory hierarchy to locate the file metadata or determine the access permissions. With appropriate data management algorithms in each metadata server, this can result in extremely efficient metadata access. Table 1 shows the distribution of a 16-bit namespace on a metadata server cluster with 4 metadata servers (real systems will use larger IDs). An illustration of metadata access under normal operations is shown in Figure 3. Four steps are required for each

Table 1. Example metadata server lookup table (MLT)

Range of Hash Values	metadata server ID
0–3FFE	0
3FFF–7FFD	1
7FFE–BFFC	2
BFFD–FFFB	3

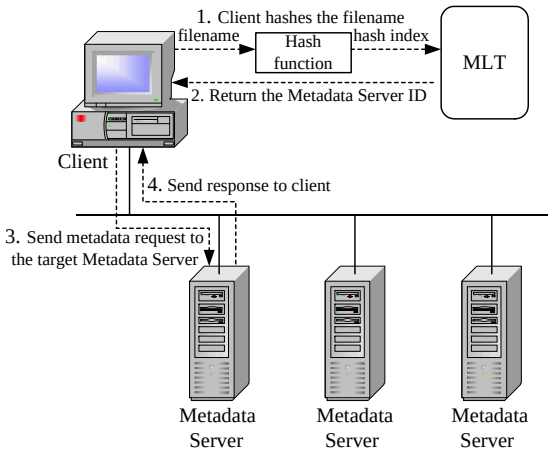

Figure 3. Accessing metadata using LH

access. First, the client hashes the filename. The resulting hash is used as an index into the MLT to find the metadata server containing the file metadata. The client contacts the specified metadata server with its request, and the metadata server responds as appropriate.

Because the hierarchy is never traversed in normal operation, and because a hash of full pathnames is used to locate the metadata servers, certain operations can incur significant overhead. A discussion of these operations and our techniques for mitigating them are the subject of the next section.

4. Lazy Policies

Because each file is located on a metadata server indicated by a hash of its full pathname, files are widely distributed among the metadata servers. As a result, most files will not be located on the same metadata server as their parent directory. Nevertheless, LH maintains hierarchical directories to support the standard directory semantics of general-purpose storage systems. A consequence of this design choice is that certain operations can incur significant overhead. We have identified four operations that can be particularly expensive in LH: changing permissions on a directory, changing the name of a directory, removing a directory, and changing the MLT to indicate the addition or removal of metadata servers in the cluster. These operation may incur overhead in two different ways. First, a large number of messages may need to be sent to the metadata servers to distribute the changes to each of the files in the directory hierarchy rooted at the changed directory. Second, a large amount of metadata may need to be moved.

When the file permissions are changed on a directory, the path permissions of all of the files and directories in the directory subtree rooted at that point will need to be updated. Because most of the files and directories will reside on different metadata servers, updating the path permissions may require a large number of messages to other metadata servers. In general, if there are n metadata servers and m files or directories in the modified subtree and given a perfect hash function, $m \times (n-1)/n$ messages will be required. Assuming 50 ms per update (a goal for our individual metadata servers), a subtree of 200 files will result in a delay of up to 1 second, and a subtree of 12,000 files will result in a delay of up to 1 minute.

When the name of a file changes, the hash of the new pathname may indicate that the file resides on a different metadata server than did the hash of the old pathname. Thus, the metadata must be relocated to the indicated metadata server so that clients can still access it directly via the hash. The overhead for moving the metadata of a single file is relatively small. However, the problem is compounded when the name of a directory changes. In that case, again because the hash is based on the full pathname, the metadata for most of the files in the subtree rooted at that point will need to be relocated. As with the permission updates, this can be prohibitively expensive to do synchronously. Removing a directory incurs similar overhead, except that the metadata is removed instead of being relocated.

When a metadata server is added to or removed from the cluster, a significant amount of metadata may need to be moved. When a metadata server is removed, the metadata stored on that server will have to be redistributed to other servers in the system. This can be accomplished by modifying the MLT, removing all references to the server in question, filling in the entries with other servers, then moving the metadata to the appropriate servers. When a new metadata server is added to the cluster, which is expected to be the more common operation by far, metadata will be moved to the server to redistribute the metadata, restoring the balance in the system and allowing the performance to scale with the number of servers. Again, the MLT is updated to move some portion of the hashed metadata values to the new server, and the metadata is moved to the new server.

The amount of latency incurred by each of these operations depends upon the amount of metadata affected, but the worst case for each is prohibitively slow. To address this problem, we employ metadata invalidation and lazy metadata update techniques that defer and distribute the cost of these operations. When one of these operations is executed, the affected metadata is invalidated or updated (but not moved). Later, upon the first access, the metadata is updated or moved by partial traversal of the directory path. The result is that the initial operation is very fast, and a small amount of overhead is incurred the first time each of the modified metadata objects is accessed. This slightly increases the average cost of the accesses, but the operations that cause this occur relatively infrequently, the overhead is incurred at most once for each updated metadata object, and the overhead is relatively small – one or two messages per updated metadata object, depending upon the update. Invalidation and lazy update are discussed in more detail in the following subsections.

4.1. Invalidation

When a directory is removed or its name or permissions are changed, the change affects all of the files in the subtree rooted at that point. If the files are accessed hierarchically, by traversing the path to reach the file, the updates are obtained automatically. However, by hashing directly to the file based on a hash of the full pathname, mistakes can be made if the updated pathname or permissions have not been propogated to all metadata objects in the path. A deleted file (in a deleted directory) could be accessed, a renamed file could be accessed by the old name or an access to the new name could fail, or file access could be allowed or prevented when permissions were changed to prevent or allow the access. Renamed files are particularly difficult because a directory could be renamed and a new (empty) directory could be created with the same name. In that case references to files in the old hierarchy should fail, even though a directory with the same name exists.

To prevent these incorrect accesses, the metadata server where the operation takes place sends an invalidation or update message to each of the other metadata servers, depending on whether the operation was a permission change or a name change. The message contains the pathname of the affected directory, the time of the invalidation or update, and the specific operation that took place. The change is logged on each of the metadata servers. Metadata servers apply logged operations to affected metadata objects in the background. At the same time, until the operation is completed, requested metadata objects can be compared with the logged operations to see if the invalidation should be applied to them before the requested operation takes place.

In this way, logged invalidation operations can be processed in the background as system load permits, while still allowing for correct operation of the system until they have been completed. Thus, once the operations have been logged, the behavior of the system is as if the invalidation operation had been completed.

4.2. Lazy Metadata Update and Relocation

By invalidating the metadata, we can prevent incorrect operations from taking place, but we haven't yet shown how we can guarantee that correct operations will take place. With deferred metadata deletions, applying the deferred operation guarantees correct behavior – the metadata no longer exists. However, with permission changes, pathname changes, and metadata location changes (caused by changes in the MLT), the metadata must still be updated and/or moved.

As was mentioned previously, the hierarchical directory structure is maintained in LH. This allows the correct information to be obtained, either the permissions of the parent directory or the actual location of moved metadata. After invalidation or update, a metadata object can be in one of two states. Either its path permission is invalidated, or the pathname is changed. The resolution of the update depends upon which is the case, but the operation of both is similar. When an object is found to have an invalid path permission, the path permission of the parent directory is obtained by hashing on the filename of the parent directory and contacting the metadata server on which it is located to request its updated path permission. Given the parent directory's path permission, the path permission of the file can be updated accordingly. If the parent's path permission is also invalid, then the process repeats recursively until either the changed directory is reached, or an updated directory in the path between the file and the changed directory is reached. Thus, each update only traverses as much of the path as necessary, and each file or directory path permission can be lazily updated.

When metadata needs to be moved, its location will not match that indicated by a hash of its pathname. This is caused by either a name change, or a change in the MLT configuration. In either case, metadata may not be present on the metadata server contacted by a client as directed by the hash function and MLT. Regardless of the cause, the action taken is the same: the metadata server contacts the metadata server containing the parent directory to see if the file exists or not. If it does not, then the appropriate response is sent to the client. If the file does exist, but on a different metadata server, then the metadata server indicated in the directory is contacted to send the metadata to the server indicated by the hash function, and the re-

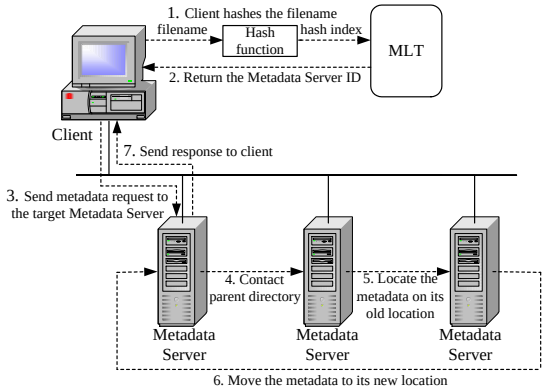

Figure 4. Lazy metadata movement

quest is completed or not, depending on whether or not the client has the appropriate permissions. When metadata is moved, the parent directory is updated to record the new location. Figure 4 shows the communications between metadata servers for a single metadata movement.

It may be the case that the parent directory does not reside on the metadata server indicated by the hash of its pathname, in which case the same process recursively takes place for the parent directory. Once that is resolved, the process continues for the file in question. This recursive procedure guarantees that the least amount of directory hierarchy traversal takes place for each request, and takes place at most once for each file or directory that is located on a metadata server other than is indicated by the hash function. Thus, although files are accessed in a flat, efficient namespace through pure hashing, the hierarchical directory structure is maintained and acts as a stable backup in locating files when direct access in the flat namespace fails.

Unfortunately, lazy data movement doesn't work particularly well for removing metadata servers, in which case it may be desirable to update and move the metadata immediately. If this is desired, the update can proceed either by synchronous traversal of the directory hierarchy, moving each object that is encountered, or by setting the invalidation request to move the metadata as it is updated, with a high priority. Synchronously updating each metadata object in the directory can proceed in parallel, as each metadata server can manage the objects in each subdirectory that it caches, allowing many of the moves to occur in parallel. This will occur infrequently enough that the overhead should be acceptable.

4.3. Efficient Log Operation

Comparing a metadata object to the log requires a simple comparison of the pathnames and times; a match occurs if the path of the logged operation is a substring of the object pathname and the time of the logged operation is newer than the timestamp of the metadata object. The timestamp of metadata objects are set to the time of the newest update they have been compared with, whether the update has been applied to them or not. To make log management as efficient as possible, logs will be sorted by time such that a metadata object need only be compared against those with times newer than the metadata object's timestamp. Sorting the operations in time order also guarantees that subsequent operations to the same file are processed correctly. When a metadata object is compared against the appropriate invalidation requests in the log as determined by the timestamp, substring matching can be accomplished relatively quickly using known techniques to determine which requests actually apply to the object.

Invalidation requests in the log can also have a priority associated with them such that they can be processed in the background at different rates depending upon their priority. In this way, some updates can take place faster than others, the size of the logs can be kept relatively small, and the updates can be guaranteed to complete eventually.

While the logs are compared with every object that is accessed, the background updates do not have to do so – they can proceed via hierarchical directory traversal. Thus, a tradeoff can be made between the low number of metadata accesses of directory traversal (which accesses only those objects in the subtree) but relatively high number of messages (one per metadata object in the subtree), and the relatively low number of messages of logged operation (one per metadata server) but high number of metadata accesses (one per accessed metadata object – but since they are already in memory, this overhead is relatively small). In either case, once the update is complete, the operation is removed from the log.

To maintain consistency of the log files, the broadcast of the related directory operations must be synchronous, processed as a distributed transaction [1]. When a metadata server recovers from a crash, before processing any requests, it should contact other metadata servers to get the most up-to-date log file. Log file recovery requires that old log operations be retained until they have been accomplished by all servers, at which time they will no longer be needed for recovery.

4.4. Links

Because the directory hierarchy is not traversed each time a file is accessed, file system links present something of a problem. In particular, hashing on a pathname that includes a hard link or a symbolic link will likely direct the client to the wrong metadata server. Even if the right meta-

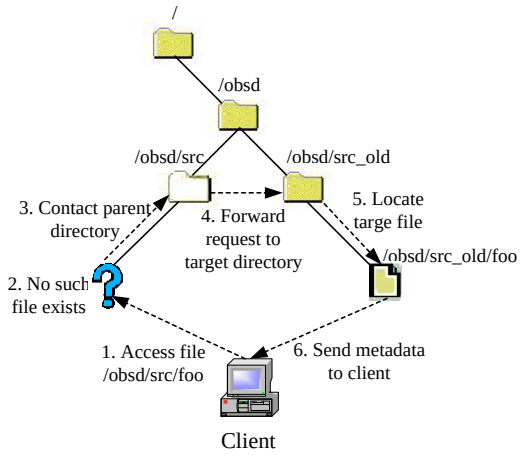

Figure 5. Accessing a directory via a symbolic link

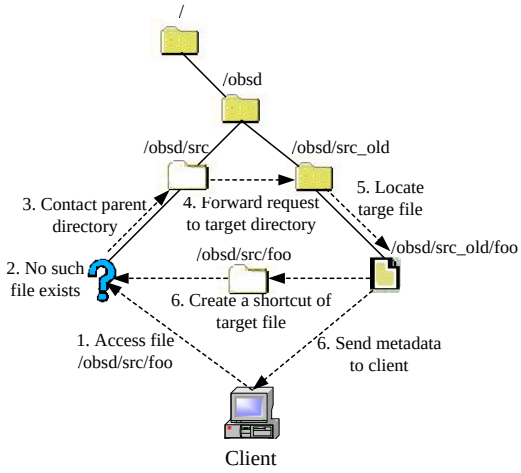

Figure 6. Creating a shortcut under a symbolically linked directory

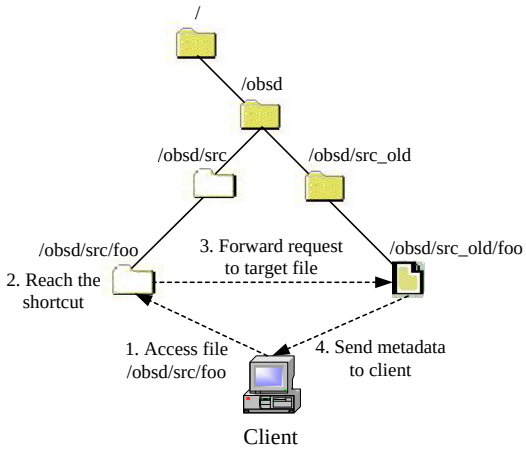

Figure 7. Accessing a file through a symbolic link shortcut

data server is contacted, a match will not be found for the specified pathname.

In LH, the lazy techniques we have created for moving data can be used to handle links as well. The simplest approach is to let the client try to access the file directly, and let the recursive metadata-not-found procedure backtrack up the directory path to the link and back down the actual path to locate the metadata. With a link to a directory, accessing the subtree of the destination directory through the symbolic link always results in a failure because the subtree's location is determined by hashing its real name under the destination directory. This is analogous to the first access of a file in the subtree of a renamed directory; hashing fails since the metadata is not moved to the new location indicated by hash of the new name. Path traversal is then used to locate the metadata. Unfortunately, unlike directory renaming, the overhead of name traversal is incurred each time the subtree is accessed through the symbolic link. If there are recursive symbolic links, the cost is higher due to nested redirections. This is not terribly efficient, but guarantees correct semantics. In the following example, */obsd/src* is a symbolic link to */obsd/src_old*. The access to */obsd/src/foo* requires six steps using the above straightforward approach, as shown in Figure 5.

An alternate solution is to allow link names to propagate through the system, analogous to the movement of metadata in the system. In this case, new metadata links can be created lazily to represent alternate names for each of the files in the directory hierarchy. This allows the system to trade storage space for latency – creating the additional links takes up space, but the end result is that frequently accessed files in the path of a linked directory can be accomplished with almost the same efficiency as regular file metadata look-ups: 2 messages instead of one, since the linked metadata will still have to be found. Figure 6 illustrates the creation of shortcut for */obsd/src/foo* in the above example. Figure 7 shows the operations for future access to */obsd/src/foo* after the shortcut is created. Like lazy metadata movement, symbolic link and shortcut creation takes place on a per-file basis as each file is accessed through a symbolic link path. When directory names or permissions change, these shortcuts may also have to be updated just like other metadata objects.

It is worth noting that although they may incur more overhead in LH than in other techniques, directory renames, links and permission modifications occur very infrequently. An examination of the Coda traces [10] for

one machine (mozart) in a general-purpose environment shows only 117 directory renames, 1851 directory symbolic links and less than 3000 directory permission and ownership changes over the course of two years. No data is available for the frequency of metadata server additions or deletions, but these operations are also expected to be relatively rare in the system. Thus the performance advantage from directly accessing metadata with a single request significantly outweighs any additional overhead incurred by these operations, and our lazy techniques will further reduce the impact of these overheads.

5. Conclusions

We present Lazy Hybrid metadata management, a scalable metadata management mechanism based on pathname hashing with hierarchical directory management. LH avoids the bottlenecks of directory subtree partitioning and pure hashing by combining their best aspects and by propogating expensive directory name and permission changes lazily, improving the performance of the system and distributing the overhead of these potentially costly operations. LH provides high performance by avoiding hot spots in the metadata server cluster and minimizing the overhead of disk access and server communication. We are currently implementing LH and integrating it into our distributed object-based storage system. We plan to use trace-driven and benchmark-driven experiments to compare the performance of LH with the traditional metadata management techniques to show that LH provides significantly better performance and scalability than standard techniques.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[2] P. Braam, M. Callahan, and P. Schwan. The intermezzo file system. In *Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*, Monterey, CA, USA, Aug. 1999.

[3] P. J. Braam. The Lustre storage architecture, 2002.

[4] P. F. Corbett and D. G. Feitelso. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[5] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[6] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), Dec. 1990.

[7] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[8] E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4):419–446, 1997.

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

[10] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software—Practice and Experience (SPE)*, 26(6):705–736, June 1996.

[11] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[12] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)*, pages 15–24, Dec. 1985.

[13] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

[14] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, , D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, Maastricht, Netherlands, May 2000.

[15] G. J. Popek and B. J. Walker. *The LOCUS distributed system architecture*. Massachusetts Institute of Technology, 1986.

[16] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[17] R. O. Weber. Information technology—SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, Aug. 2002.