Carnegie Mellon University
Research Showcase @ CMU

Parallel Data Laboratory | Research Centers and Institutes

5-2006

Eliminating Cross-server Operations in Scalable File Systems (CMU-PDL-06-105)

James Hendricks
Carnegie Mellon University

Shafeeq Sinnamohideen
Carnegie Mellon University

Raja R. Sambasivan
Carnegie Mellon University

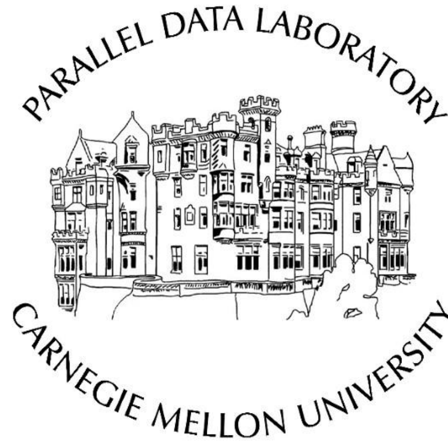
Gregory R. Ganger
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/pdl

Recommended Citation

.

This Technical Report is brought to you for free and open access by the Research Centers and Institutes at Research Showcase @ CMU. It has been accepted for inclusion in Parallel Data Laboratory by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.



Eliminating cross-server operations in scalable file systems

James Hendricks, Shafeeq Sinnamohideen, Raja R. Sambasivan, Gregory R. Ganger

CMU-PDL-06-105

May 2006

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Distributed file systems that scale by partitioning files and directories among a collection of servers inevitably encounter cross-server operations. A common example is a RENAME *that moves a file from a directory managed by one server to a directory managed by another. Systems that provide the same semantics for cross-server operations as for those that do not span servers traditionally implement dedicated protocols for these rare operations. This paper suggests an alternate approach that exploits the existence of dynamic redistribution functionality (e.g., for load balancing, incorporation of new servers, and so on). When a client request would involve files on multiple servers, the system can redistribute those files onto one server and have it service the request. Although such redistribution is more expensive than a dedicated cross-server protocol, the rareness of such operations makes the overall performance impact minimal. Analysis of NFS traces indicates that cross-server operations make up fewer than 0.001% of client requests, and experiments with a prototype implementation show that the performance impact is negligible when such operations make up as much as 0.01% of operations. Thus, when dynamic redistribution functionality exists in the system, cross-server operations can be handled with little additional implementation complexity.*

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Symantec) for their interest, insights, feedback, and support. We also thank Intel, IBM, Network Appliances, Seagate, and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by the Army Research Office, under agreement number DAAD19–02–1–0389. James Hendricks is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense.

Keywords: Object-based storage, OSD, metadata scalability, cross-server operations, dynamic redistribution, trace analysis, failure recovery, atomicity in distributed systems, mitigating complexity

1 Introduction

Distributed file services should be transparently scalable. That is, it should be possible to increase both capacity and throughput by adding servers and spreading data and work among them. Further, users and client applications should not have to be aware of which servers host which data—the visible semantics should be consistent. This design goal has been demonstrated by several systems (e.g., [2, 3, 28]).

Most designs scale by partitioning the set of files across the set of servers such that each file is managed by one server. Most operations access a single file, and accesses to files managed by distinct servers can proceed in parallel.[1] But, a few operations require access to more than one file and, occasionally, the files will be managed by distinct servers. For example, a RENAME operation can move a file from one directory to another, thus involving both directories. Some systems don't guarantee the same semantics for operations on multiple files that are managed by distinct servers, but "transparent" scalability dictates that a distributed file system should. Continuing with the example, many applications rely on RENAME operations to provide specific consistency semantics (e.g., atomicity).

The common approach to upholding the semantics uses additional protocols to complete cross-server operations. Doing so provides the expected semantics, but it introduces significant complexity to support a very uncommon case—for example, cross-directory RENAMEs represent less than 0.001% of operations in analyzed NFS traces, and little of that 0.001% would involve more than one server in most deployments. This rareness also means that cross-server operations don't naturally show up in benchmark-based testing, aggravating the cost of the added complexity (by requiring crafted test scenarios) or resulting in robustness holes.

This paper promotes an alternate approach: avoid cross-server operations altogether. When a client request would involve files managed by more than one server, the system redistributes management responsibilities such that the affected files are managed by the same server. The ability to redistribute files is an important part of balancing load among servers, allowing a system to address hot spots, capacity exhaustion, and server addition or removal. If it is present for these other reasons, dynamic redistribution can be used to eliminate cross-server operations for "free," avoiding the need for protocols to support them.

This paper describes a prototype metadata service that supports dynamic redistribution and uses it to avoid cross-server operations. Each metadata server manages an independent set of files and stores its state as database tables in shared storage-nodes. Moving management of some files from one server to another involves two steps: checkpointing the relevant database table(s) and passing ownership of them to the new server. When the system needs to move management of only a subset of the files, splitting a table can reduce the overall impact. But, moving entire tables will work, and of course they can be moved back, if appropriate, after the movement-inducing operation is complete. Clients contact the server most recently known to manage a file and are redirected if necessary.

Experiments with the system and trace analyses illustrate the efficacy of using dynamic redistribution to eliminate cross-server operations. As expected, we observe linear throughput increases for balanced workloads when increasing the number of servers. Also as expected, operations that require redistribution to avoid being cross-server are significantly slower (e.g., 7100μs vs. 1000μs for RENAME) than like operations that do not. Analysis of long-term NFS traces, however, show that such operations are very rare—fewer than 0.001% of operations could be cross-server (e.g., cross-directory RENAMEs), and their locality makes most of them unlikely to be cross-server in practice (e.g., because they rename a file to the immediate parent directory). Thus, although cross-server operations are much slower, the impact on overall performance is minimal. Balanced against the simplicity of avoiding them, we believe this approach is right for many distributed file systems and perhaps other distributed systems as well.

[1] We talk in terms of one server per file, for clarity. Each "server" in this discussion could be a fault-tolerant group of replicas (i.e., a replicated state machine [26]), as in Farsite [2] for example, with no change.

The remainder of this paper is organized as follows. Section 2 reviews scalable file services, cross-server operations, and related work. Section 3 discusses design issues for dynamic redistribution support and its use in avoiding cross-server operations. Section 4 describes our implementation of such support in a prototype distributed storage system. Section 5 evaluates this approach to handling potential cross-server operations with experiments and trace analyses.

2 Background and related work

This section discusses scalability and transparency in distributed file systems, cross-server operations, and approaches to supporting them.

2.1 Scalability in distributed file systems

There have been many distributed file systems proposed and implemented over the years. To provide context for this work, we categorize them into three groups based on what they can scale transparently. We use "scaling" to refer to increasing storage capacity and throughput by adding more servers, as contrasted with using techniques like client-side caching to increase the number of clients that a given server can support [16]. Thus, scaling happens by increasing the number of servers and partitioning data and responsibilities across them. "Transparent" scaling implies scaling without client applications having to be aware of how data is spread across servers; a distributed file system is not transparently scalable if client applications must be aware of capacity exhaustion of a single server or different semantics depending on which server(s) hold accessed files.

No transparent scalability: Many distributed file systems, including those most widely deployed, do not scale transparently. NFS, CIFS, and AFS all have the property that file servers can be added but that each serves independent file systems (or volumes, in the case of AFS). A client can mount file systems from multiple file servers, but must cope with each server's limited capacity and the fact that certain operations (e.g., RENAME) are not atomic across servers.

Our focus, in this paper, is on distributed file systems that provide fairly strong consistency semantics. But, file systems that provide weaker consistency, such as eventual consistency with posthoc conflict detection and application/user-assisted resolution (e.g., Bayou [27], Pangaea [24], and Ivy [20]), could also go into this "no transparent scalability" category.

Transparent data scalability: An increasingly popular design principle is to separate metadata management (e.g., directories, quotas, data locations) from data storage [3, 13, 17, 28]. The latter can be transparently scaled relatively easily, assuming all multi-object operations are handled by the metadata server(s), since each data access is independent of the others. Clients interact with the metadata server(s) for metadata activity and to discover the locations of data. They then access data directly at the appropriate data server(s). Metadata semantics and policy management stay with the metadata server(s), permitting simple centralized solutions. The metadata server(s) can limit throughput, of course, but offloading data accesses pushes the overall system's limit much higher [14]. Scaling of the metadata service is needed to go beyond this limit, and many existing systems do not offer a transparent metadata scaling solution (e.g., Lustre [18] and most SAN file systems).

Transparent scalability: A few distributed file systems offer full transparent scalability, including Farsite [2], GPFS [25], and Frangipani [28]. Most use the data scaling architecture above, separating data storage from metadata management. Then, they add protocols for handling metadata operations that span metadata servers. Section 2.3 discusses these further.

2.2 Multi-file operations

There are a variety of operations that manipulate multiple files, thus creating a consistency challenge when the files are not all on the same server. Of course, every file create and delete involves two files: the parent directory and the file being created or deleted. But, most systems assign a file to the server that owns its parent directory. At some points in the namespace, of course, a directory must be assigned somewhere other than the home of its parent, or else all metadata will be managed by a single metadata server. So, the create and delete of that directory will involve more than one server, but none of the other operations on it will do so. This section describes three more significant sources of multi-file operations: non-trivial namespace manipulations, transactions, and snapshots.

Namespace manipulations: The most commonly noted multi-file operation is RENAME, which changes the name of a file. The new name can be in a different directory, which would make the RENAME operation involve both the source and destination parent directories. Also, a RENAME operation can involve additional files if the destination exists (and, thus, should be deleted) or if the file being renamed is a directory (in which case, the '..' entry must be modified). Application programming is simplest when the RENAME operation is atomic, and both the POSIX and the NFSv3 specifications call for atomicity.[2] Many applications assume atomic RENAME, or at least that the destination will be its before or after version, as a building block. For example, many document editing programs implement atomic updates by writing the new document version into a temporary file and then using RENAME to move it to the user-assigned name. Without atomicity, applications and users can see strange intermediate states, such as two identical files (one with each name) existing or one file with both names as hard links.

Creation and deletion of hard links are also multi-file operations. Links created in the same directory rarely result in cross-server operations. But, for links created in other directories, it is possible for the two directories with names for a file to be on different servers. Thus, the create and subsequent delete of that file would involve both servers. The same can happen with a RENAME that moves a file into a directory managed by a different server than the directory in which it was originally created.

Transactions: Transactions are a very useful building block. Modern file systems, such as NTFS [21] and Reiser4 [23], are adding support for multi-request transactions. So, for example, an application could update a set of files atomically, rather than one at a time, and thereby preclude others seeing intermediate forms of the set. This is particularly useful for program installation and upgrade.

Snapshot: Point-in-time snapshots [5, 15, 19, 22] have become a mandatory feature of most storage systems, as a tool for on-line and consistent off-line back-ups. Snapshots also offer a building block for on-line integrity checking [19] and remote mirroring of data [22]. Snapshot is usually supported only for entire file system volumes, but some systems allow snapshot of particular subtrees of the directory hierarchy. In any case, it is clearly a substantial multi-file operation, with the expectation that the snapshot captures all covered files at a single point in time.

2.3 Handling cross-server operations

As discussed in Section 2.1, few distributed file systems support cross-server operations transparently. Those that do use one of two approaches: client-driven protocols and inter-server protocols.

In the first approach, illustrated by GPFS [25] and Frangipani [28], the client requesting the cross-server operation implements it. This approach is generally used for shared disk (or logical disk) systems. The client acquires locks on the relevant files and updates them in the shared disk system, using write-ahead logging into shared journals to update them atomically. This approach offloads work from servers, which

[2] Each specification indicates one or more corner cases where atomicity is not necessarily required. For example, POSIX requires that, if the destination currently exists, the destination name must continue to exist and point to either the old file or the new file throughout the operation.

is generally a good thing, but also trusts clients to participate correctly in metadata updates and introduces interesting failure scenarios when clients fail.

In the second approach, illustrated by Farsite [2],[3] operations that span servers are handled by an inter-server protocol. The protocol usually consists of some form of two-phase commit, with a lead server acting as the initiator. Because this second approach does not require modification of the client-server protocol, it is the one most often identified as a "planned extension" for systems that do not yet scale to more than one metadata server (e.g., [1, 18]). Such extensions often stay "planned" for a long time, however, because of complexity and performance overhead concerns (e.g., [7, 8]).

Because cross-server operations are rare, protocols implemented just to support them usually do not receive the same attention as the more common cases. They tend to be exercised and tested less thoroughly, becoming a breeding ground for long-lived bugs that will arise when this code is finally relied upon. We promote an approach based on eliminating cross-server operations, via heavy-weight redistribution that serves multiple functions, rather than supporting them with their own protocols. Doing so can simplify scalable distributed file systems with minimal impact on performance, so long as cross-server operations are truly rare.

3 Dynamic redistribution

The ability to dynamically redistribute responsibility for storing and managing files is a powerful building block of scalable file systems. Dynamic redistribution allows a system to balance load when hot spots form or one server's capacity is exhausted. It also allows the system to reconfigure responsibilities when servers are added or removed, so as to appropriately utilize available resources. This section reviews how dynamic redistribution can be realized and describes how it can be used to eliminate cross-server operations.

3.1 Supporting dynamic redistribution

Dynamic redistribution has been a part of many previous system designs. For example, AFS [16] allows volumes to be migrated freely among servers in an AFS cell. xFS's design [3], which splits metadata management from data storage, allows for migration of each (for subsets of data) among participating systems.

Although not trivial, dynamic redistribution is not overly complex. It involves four primary components: a level of indirection between IDs and locations, a mechanism for changing that mapping, a mechanism for moving the content from previous "owner" to new, and a mechanism for deciding what should be where.

Mapping objects to servers: A level of indirection is a common tool for gaining flexibility in mapping virtual identities to physical resources. Dynamic redistribution exploits such a level of indirection, between a data or metadata ID and the server that manages that object, which can be provided by a mapping server (e.g., the volume location database in AFS) or a read-shared table (e.g., the manager map in xFS).

Changing the mapping: When the mapping is changed, relevant clients and servers must know about it. The previous and current servers must know, since the former should stop servicing requests and the latter should start. For the servers involved, at least, there must be agreement on who "owns" a given object at every point in time. Also, clients obviously must be able to learn the latest server for a given object in order to be able to access it.

A change to the mapping usually includes two parts: updating the mapping and rerouting clients with out-of-date mapping information. The mapping can be updated locally, in the mapping server case, or

[3] Recall that "server" in our discussion could be replaced with "replicated state machine," with no other changes. Farsite uses BFT-based [4] Byzantine fault-tolerant state machine replication to form each "directory group" that manages a subset of the namespace. For Farsite, "cross-server operations" means "cross-directory group operations."

updated and pushed out in the read-shared table case. Clients can easily end up going to a previous server with a request—for example, if it requested the mapping just before the change and sent the request just after—so servers must reroute such client requests to the appropriate server. They can do so by informing the client of the change or forwarding the request directly.

Moving the content: In addition to shifting responsibility, the actual content being served needs to be put under the control of the new server. There are two cases to consider. First, when the object metadata being redistributed is stored in a storage infrastructure shared by the servers (e.g, in a SAN to which they are all attached), it is sufficient for the previous server to checkpoint any relevant dirty cache state and stop accessing it. The new server can then take over responsibility for that metadata and access it directly on the same storage device. This "shared storage" architecture is increasingly common in scalable file systems [2, 3, 28]. Second, when objects are stored on each server's locally-attached storage devices, they must actually be copied from one server to another. Clearly, such copying is much more expensive than the first case, and thus redistribution in such an environment must be more carefully used.

While moving the contents from one server to another, most systems will block write requests for those contents. In such systems, moving the contents usually cannot be a background activity, and redistribution can create a noticeable hiccup in performance.

Coordinating changes: A system that supports dynamic redistribution must have policies for deciding which changes to make and when to make them, as well as a mechanism for invoking and coordinating changes. Redistribution is usually used to improve performance or address a capacity constraint. At the same time, it comes with overheads (e.g., locking and data copying). Thus, decisions to redistribute should balance these issues. In many systems, redistribution is driven by explicit administrator tools, and approaches for automated decision-making continue to be refined.

Changes that are desired must be coordinated to handle possibly concurrent redistributions. One approach is to have a central "coordination server" that decides what should be where and enacts one change at a time to move towards that state. Alternately, a system can use agreement protocols among the relevant servers to decide on a change to the mapping table and then enact it. Each change involves locking the relevant objects, moving them, changing the mapping information, and unlocking the objects. Changes for independent sets of objects can certainly proceed in parallel, though the rate of changes should not be high enough to make such concurrency important.

3.2 Eliminating cross-server operations

In addition to load and capacity balancing, dynamic redistribution can also be used to eliminate cross-server operations; Figure 1 illustrates our method for accomplishing this goal. The concept is straightforward: when a multi-object request arrives that requires access to objects "owned" by distinct servers, the system can redistribute those objects and then service the request on the server that owns them all. If desired, the objects can be redistributed to their pre-request servers immediately after the request is completed, or they can remain on the new server until the load-balancing policy dictates that they be moved. Although such dynamic redistribution may be significantly less efficient than an explicit cross-server update protocol, the impact on overall performance will be small if cross-server operations are rare.

Within this basic concept, there are many options. For example, the most efficient approach to handling a single multi-object request is to redistribute only the objects involved, particularly if the server state is not kept in shared storage. But, limitations of the data structures used for storing metadata and mapping locations may not allow redistribution of just a few objects; they may dictate that large collections be redistributed together, particularly if they were designed with only load balancing in mind. Perhaps the simplest approach is to collapse all objects owned by two servers into one, splitting them again afterwards if imbalance concerns dictate doing so. If the need for redistribution is infrequent, these choices can be made largely based on implementation simplicity and conformance to other uses of the dynamic redistribution

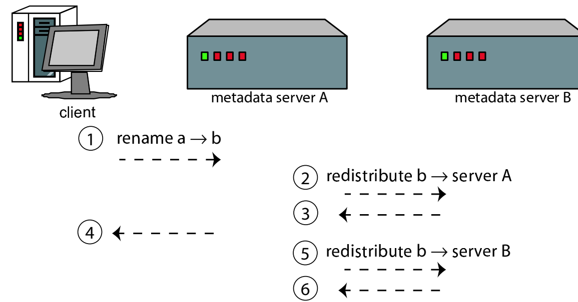


Figure 1: **Design for eliminating cross-server operations**. The sequence of operations required to handle RENAME a to b is shown. Returning to the original state is similar.

mechanism.

4 Implementation

We implemented dynamic redistribution and cross-server RENAME on our prototype distributed storage system, Ursa Minor. A high-level overview of our implementation is provided in Section 4.1 and the details are described in Section 4.2. Further information about Ursa Minor can be found in Abd-el-Malek et al [1].

4.1 Overview

Ursa Minor's architecture is based on the NASD/OSD storage model [13, 29] (see Figure 2). In this model, clients query a metadata server to map object IDs to the storage nodes in which that object's data is stored. The metadata server stores its metadata in tables, which are stored as objects in the storage node cluster. It accesses these objects as any other client would. Under high load, the centralized metadata server can become a bottleneck. Though it could be replicated for fault tolerance, our metadata server is not; however, it does provides a level of failure recovery by synchronously writing metadata operations through to the underlying storage nodes. A crashed metadata server can be recovered by pointing a new metadata server to the crashed server's metadata table. This server is implemented in about 20,000 lines of code (see Table 1).

Implementation of dynamic redistribution involved modifying a centralized metadata server capable of crash recovery. Additionally, a special *root metadata server* was created; in our implementation, this server is responsible for coordinating redistribution. Since the granularity of redistribution is a single table in our implementation, the metadata servers distribute object metadata over multiple tables in order to minimize the number of objects that must be moved for a single a cross-server operation. To dynamically redistribute a portion of a metadata server's load, the associated table is "crashed" at the source and recovered at the destination. The root metadata server coordinates the distribution process. Approximately 3000 lines of code were modified to support multiple tables and the special features of the root metadata server.

Once dynamic load redistribution was in place, adding cross-server operations required only 300 lines of code. Cross-server operations are built entirely on top of the dynamic redistribution mechanism by redistributing management of all involved objects to a single server and then executing the standard command code path. This is described in more detail in Section 4.2.2.

Our implementation is detailed below to show how little complexity is involved in our solution and to demonstrate how a legacy system can be retrofitted for dynamic redistribution and cross-server operations. The primary design dictum was to do whatever is simplest with little consideration for performance. Our experimental results show that an implementation need not be elegant or efficient to scale well.

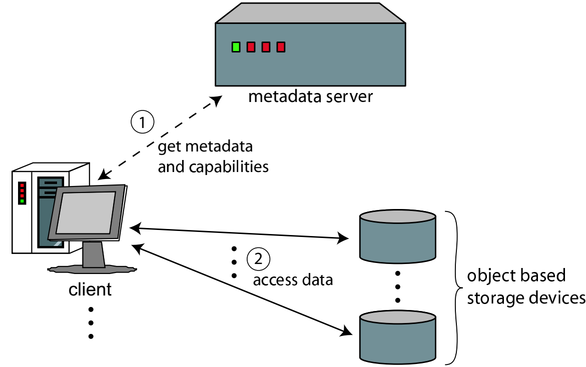


Figure 2: **Direct client access**. First, a client interacts with the metadata server to obtain mapping information (e.g., which object IDs to access and on which storage devices) and capabilities (i.e., evidence of access rights). Second, the client interacts with the appropriate storage device(s) to read and write data, providing a capability with each request.

Component	Lines of Code
Metadata Server	19694
Dynamic redistribution	2959
Cross-Server Rename	300

Table 1: **Complexity in terms of Lines of Code**. If facilities for dynamic redistribution are in place, adding cross-server rename is relatively simple.

4.2 Implementation in Ursa Minor

In Ursa Minor, clients access data on storage nodes directly once they have retrieved the data location, encoding, and permissions from the metadata server. Clients access storage nodes using the PASIS read/write protocols [30] which provide fault tolerance and consistency for block reads and writes. Storage nodes use non-volatile RAM to allow writes to return immediately as long as there is room in the write-back buffer cache.

The metadata server is responsible for granting permissions and storing metadata safely. Because metadata is small and RAM is plentiful, most read-only operations, such as lookup, are expected to hit in cache and return almost immediately. Operations that require modifications, however, are far more expensive. To durably store an operation that creates metadata (e.g: CREATE) or an operation that changes metadata (e.g: RENAME or a change of object size) our metadata server writes the modified metadata to a storage node synchronously. This limitation is aggravated because our naïve implementation limits update concurrency.

Metadata servers store metadata records in a B-tree table. The B-tree is updated atomically using the following simple shadow paging scheme. Each page contains a list of all other pages updated by that transaction, the transaction ID, and the new and previous versions of that page's data. The PASIS R/W protocols guarantee that page writes are atomic. Only one transaction is allowed to commit at a time in the current implementation.

This policy guarantees that if transaction N completed, any transaction numbered lower than N has also completed. Otherwise, a transaction has completed if all associated pages have been written. Whether a page is part of a partially completed transaction is determined by verifying that all pages involved in that transaction were written by that transaction or a subsequent one. Given that a transaction N has completed, only pages marked with a higher transaction ID need be examined. If a transaction is incomplete, it can be rolled back using the previous data included in the page. The detection and rollback process can be performed by a `fsck`-like tool or online.

The primary benefit of this mechanism is that it is simple. A failed metadata server can easily be restarted on another node. Similarly, an unresponsive metadata server can be forced down and restarted on another node. It also works when using multiple tables, though the recovering server must have access to all tables involved in the last transaction. This can be avoided during a clean shutdown if the server performs a final transaction on each table such that its last transaction involves no other table.

4.2.1 Dynamic redistribution

Dynamic redistribution was added in two steps. First, metadata was partitioned into multiple B-tree tables, each holding metadata for a set of object IDs. All distribution is done at the granularity of these tables and each table is managed by a single metadata server at any time. Though tables cover a fixed range of object IDs in our prototype, we are currently implementing support for dynamic resizing of tables.

The second step was to implement the actual redistribution routines. The root metadata server can request that a metadata server assume or relinquish responsibility for a metadata table. A metadata server that does not respond in a timely manner is assumed faulty and is recovered as described below. Hence, the root metadata server has the following three responsibilities. First, it keeps a delegation list describing which metadata server manages each object. Second, it coordinates redistribution. Third, it durably stores the metadata for all metadata tables in a meta-metadata table that it manages. This latter table is required to bootstrap the root and other metadata servers as well as to recover from crashes. Each metadata server queries the root, on startup, for the delegation list and metadata to access the tables it is assigned to manage.

A metadata server must execute commands issued by the root. Upon assuming management of a table, a metadata server verifies its consistency and begins serving queries for that table. To relinquish responsibility for a table, a metadata server completes executing queries and queues further queries. It then flushes its cache of the table to the storage nodes and commits a null transaction to cleanly close the table. While management responsibilities are being transferred, there is a short period when no metadata server is managing the table. If the client tries to issue a request, it will discover this and will re-fetch the delegation list. Hence, queries will queue up at the client during this time. A metadata server can request more or fewer tables and it may request adding or dropping management responsibilities for a particular object. If the root agrees to such requests, it will coordinate the sequence change in management responsibilities.

Clients are unchanged except that they must request an updated delegation list from the root if the metadata for an object is not where expected.[4] This can happen at startup or after redistribution. Of course, a client must issue requests to the metadata server specified in the latest delegation list that it has cached.

Recovery is a more general variant of the centralized case. Instead of restarting a failed server on a new host, the metadata tables from a failed metadata server are redistributed to the remaining metadata servers. If a new host is booted for the purpose of recovery, this server is a natural candidate for managing such orphaned tables. Our current implementation suffers from the simple failure recovery mechanism outlined above. Specifically, all tables involved in a transaction are required to recover a table of questionable consistency. Therefore, the consistency of tables with overlapping operations must be checked from a single server. Fortunately, few operations require more than one table. A `fsck`-like tool can be used in the background to certify a table as consistent. To mitigate this expense, the recovering metadata server can accept all of a failed server's tables and then redistribute its original tables to shed load.

4.2.2 Cross-server operations

We implemented two atomic cross-server operations, RENAME and ENUMERATE. ENUMERATE provides an atomic view of the metadata for a subset of objects in the system. We need ENUMERATE only for bootstrapping, so it need not be fast.

[4] Delegation list queries can be served by other metadata servers, as well.

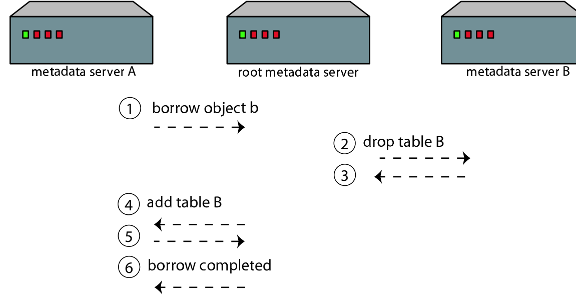


Figure 3: **Our implementation of cross-server operations**. The sequence of operations required to handle RENAME a to b is shown. Returning to the original state is similar.

Implementing cross-server operations on top of dynamic redistribution was simple. Here, we outline RENAME; ENUMERATE is similar. To move an object from a directory managed by one metadata server to one managed by another, the metadata server with the source directory borrows management responsibility for the destination directory. The root metadata server coordinates this redistribution, and the metadata server executes a same-server RENAME. When the operation completes, the metadata server requests that the root redistribute the destination directory back to its original server. Figure 3 illustrates the first half of this sequence.

For simplicity, in our implementation, the server performing a cross-server RENAME handles no other queries during the RENAME operation. As a result, if each metadata server manages N tables, $N+1$ tables are essentially locked for the duration of the RENAME. Additionally, the root metadata server avoids deadlock problems (e.g., a case where one server holds a lock for table `A` and requests another for table `B` while another holds the lock for table `B` and wants the lock for table `A`) by allowing only one redistribution in the system at a time. Though there are higher performance solutions to this potential deadlock problem, our simple solution is sufficient; it does not degrade performance in practice due to the rareness of cross-server operations.

5 Evaluation

We evaluated the scalability of our prototype on a cluster of twenty well-connected server nodes (the hardware setup is in Table 2). Of these twenty machines, there were six each of metadata servers, storage nodes, and load generators. The other two were used for the root metadata server and its storage node.

Metadata tables were distributed equally across the storage nodes, and each metadata server was initially given management responsibility for all tables at one storage node. Each storage node used a 20 GB partition on its disk for the metadata tables assigned to it.

Metadata servers used a 640 MB write-through cache, and storage nodes used a 640 MB write-back cache stored in non-volatile RAM.

Though storage clusters with hundreds or even thousands of disks are becoming more common [6, 9], our resources are more limited. To fully load our metadata servers with a standard filesystem benchmark would require more storage nodes than we have available. Furthermore, it would not allow us as much control over our experiments. We implemented a distributed workload generator that replays operations from the traces described in Section 5.2. The workload generator bypasses our prototype's filesystem and instead queries the metadata servers directly. To ensure that all available metadata server throughput was utilized, we used as many machines for the load generator as for metadata servers. Operations are played in parallel, as fast as possible, so long as the operation involves only objects that have already been successfully

Servers	20 × Dell PE650
Memory	1 GB RAM
Processor	2.66 Ghz Pentium 4
Disk	Seagate ST336607LW 36 GB
NIC	Intel e1000
Switch	Dell PowerConnect 5224

Table 2: **Hardware setup**. Metadata Server, Storage Node, and Load Generator configuration.

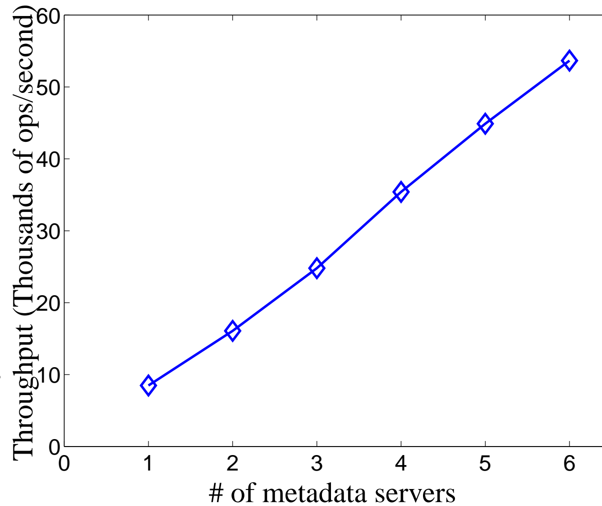


Figure 4: **Metadata Servers vs Throughput**. Throughput using the Harvard EECS workload scales linearly with respect to the number of Metadata Servers.

created.

Objects seen in the trace are randomly distributed among the available metadata tables. Operations within a directory are always executed on a single server. Directory locality is lost in the trace replay, so directories are similarly distributed randomly amongst metadata tables, creating more cross-server operations than would truly be present. Operations that span directories (cross-directory renames) require both directories; because directories are randomly distributed, such operations will be executed on two random servers. So, for any N metadata servers, there is a $1/N$ chance that such operations will operate on a single server.

5.1 Performance

Figure 4 shows that our prototype scales linearly for an average day of the Harvard EECS workload described below. (Operation counts for the three workloads described below are given in Table 4.) In this sense, our simplistic implementation exhibits optimal performance. As expected, when cross-server operations are rare, cross-server operation performance is largely unimportant.

Table 3 shows average latency for a number of individual operations. Clients get a response to metadata update operations in 1000 μs because the metadata server must durably write the operation to a storage node. The response time goes down to 260 μs—almost the network roundtrip time of 230 μs—if the operation can be serviced from the Metadata Server cache. Atomic cross-server renames exhibit a latency of about 7000 μs, which is tolerable for interactive operations and, because cross-server renames are so rare, does not affect overall performance.

Operation	Latency at Client
CREATE	1.20ms
LOOKUP	0.26ms
UPDATE	1.00ms
Same-server RENAME	1.00ms
Cross-server RENAME	7.10ms
Ping	0.23ms

Table 3: **Microbenchmark**. Lookup is fast. Modifying operations are slower, and cross-server rename is slowest.

The cost of cross-server rename is not entirely reflected in the individual operation latencies because moving a table for a rename affects more than just the rename operation. The most expensive cost of cross-server operations in our system is that the issuing metadata server must lock its tables while it borrows and returns the destination table. This cost is entirely an artifact of our implementation running on top of a naïve crash-recovery mechanism and can be avoided in systems where it affects overall performance. A second penalty is that a RENAME flushes the destination metadata server's cached copy of a table. This could be partially resolved by subdividing each table further and further resolved by reloading previously cached entries when a table is returned. Additonally, if the borrower promises to modify only one object, the lender only needs to flush that one. If the issuing metadata server did not stall, a third cost would be that this server would now need to service queries to its allocated tables plus any queries to any tables that it is currently borrowing. This can be limited by using more fine-grained redistribution, either by using more tables or by dynamically creating tables. We plan to support more fine-grained redistribution by dynamically splitting and merging tables; cross-server operation performance will improve as a result for. For overall performance, however, none of these costs are significant given the rarity of cross-server operations.

Figure 5 considers the impact of the cross-server operation frequency on throughput. The proportion of cross-directory RENAMES is gradually increased in a workload otherwise identical to the Harvard EECS distribution accessing six metadata servers. Even when cross-directory renames are 0.1% of the workload, which is orders of magnitude larger than in the traces, our unoptimized implementation achieves 80% of its maximal throughput.

To understand this data, consider the state of each metadata server. Recall that our prototype allows only one cross-server rename to proceed at a time, and the metadata server handling that rename handles no other requests. Hence, for six metadata servers, when there is always an outstanding cross-server RENAME, the expected throughput is just under $\frac{5}{6}$, or about 80%. The system does not scale past this boundary; when more than one RENAME is outstanding, that RENAME must wait until the first completes. So, we expect our prototype to scale well until cross-server RENAMEs conflict. Indeed, if 5 metadata servers respond to 45,000 operations per second (from Figure 4) and a cross-server RENAME takes 0.007 seconds, then all six metadata servers can respond to one cross-server RENAME and 315 operations in 0.007 seconds. Hence, our prototype will not scale when cross-server RENAMEs are $\frac{1}{316} = 0.3\%$ of operations (as shown in Figure 5). If our implementation allowed concurrent redistribution and concurrent queries during RENAME, performance would be bound only by the overhead of dynamic redistribution. We estimate that such a system could achieve over 90% of its maximum throughput even if cross-server operations represented 2% of the workload, and over 50% if cross-server operations were 10%. Dynamic resizing of tables would provide further benefit.

The experimental results from our prototype implementation demonstrate that scalability is linear if cross-server operations are rare. Fortunately, cross-server operations are extremely rare in practice. The following section quantifies just how rare these operations are.

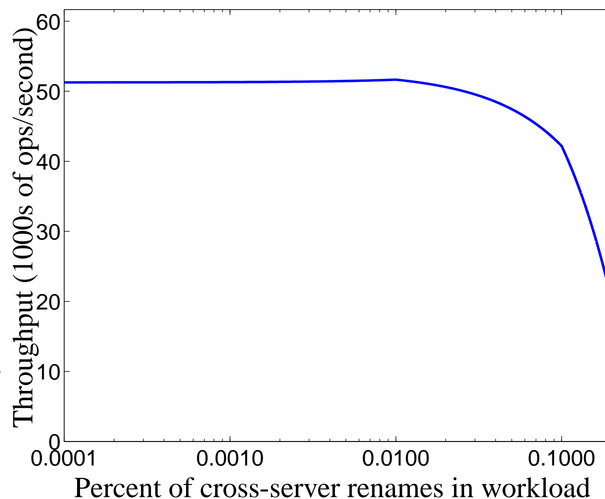


Figure 5: **Throughput vs Percent of cross-server renames**. This graph shows how throughput varies as cross-directory renames are added to a workload.

5.2 Analysis of RENAMEs in real-world workloads

We analyzed three NFS traces in order to determine the frequency and distribution of RENAME operations in real workloads. Our results show that while RENAMEs are rare, cross-directory RENAMEs are even rarer—they comprise a maximum of only 0.006% of all operations in the traces. We also find that cross-directory renames arrive in quick bursts, indicating that it might be possible to amortize the overhead of initiating a cross-server RENAME over multiple RENAME operations.

5.2.1 Traces used

We used three NFS traces [10] from Harvard University for our trace-based evaluation. We describe each trace and its constituent workload below.

EECS03: The EECS03 trace captures NFS traffic observed at a Network Appliance filer between February 8nd–9th, 2003. This filer serves home directories for the Electrical Engineering and Computer Science Department. It sees an engineering workload of research, software development, course work, and WWW traffic. Detailed characterization of this environment can be found in [12].

DEAS03: The DEAS03 trace captures NFS traffic observed at another Network Appliance filer between February 8th–9th, 2003. This filer serves the home directories of the Department of Engineering and Applied Sciences. It sees a heterogenous workload of research and development *combined with* e-mail and a small amount of WWW traffic. The workload seen in the DEAS03 environment can be best described as a combination of that seen in the EECS03 environment and e-mail traffic. Detailed characterization of this environment can be found in [11] and [12].

CAMPUS: The CAMPUS trace captures a subset of the NFS traffic observed by the CAMPUS storage system between October 15th–28th, 2001. The CAMPUS storage system provides storage for the e-mail, web, and computing activities of 10,000 students, staff, and faculty and is comprised of fourteen 53 GB storage disk arrays. The subset of activity captured in the CAMPUS trace includes only the traffic between one of the disk arrays (home02) and the general e-mail and login servers. NFS traffic generated by serving web pages and by students working on CS assignments is not included. However,

	EECS		DEAS		CAMPUS	
Total operations	176,933,945		577,751,752		655,637,109	
LOOKUP	165,511,605	92.979%	567,130,767	98.162%	638,424,369	97.375%
CREATE	1,241,034	0.758%	1,856,980	0.321%	1,917,930	0.293%
UPDATE	9,469,619	6.207%	8,593,249	1.487%	152,288,896	2.332%
Same-directory RENAME	96,086	0.0543%	134,680	0.023%	5,666	ε
Cross-directory RENAME	2,186	0.0012%	36,076	0.006%	248	ε

Table 4: **Metadata operation breakdowns for the EECS, DEAS, and CAMPUS month-long traces**. The equivalent metadata operations necessary to perform each operation in the traces are counted and shown in this table. Note that RENAMEs are very rare. NFS `GETATTR` operations are not counted in this table as they are an artifact of the NFS caching model and, as such, clients of an object-based storage system would not issue these requests as the storage system would likely support more a more advanced caching model (e.g., leases). Note that if `GETATTR`s were counted, the contribution to total operations by RENAMEs would decrease even further.

despite the these exclusions, the CAMPUS trace contains more operations per day (on average) than either the EECS03 or DEAS03 trace. Detailed characterization of this environment can be found in [10] and [11].

5.2.2 Trace analysis

Real-world workloads contain orders of magnitude fewer cross-server operations than can be supported by our rename distribution method without loss of performance. Table 4 shows the breakdown of operation counts for both traces. RENAMEs account for only 0.054% of all operations (96,086 operations) in the EECS trace, 0.023% of all operations (134,680 operations) in the DEAS trace, and are almost non-existant in the CAMPUS trace. Additionally, most of these RENAMEs will not require a multiple server interaction, as they move files within the same directory (i.e., the source directory and destination directory are the same). Cross-directory RENAMEs, which may require a multiple server interaction, account for only 0.001% of all operations (2,186 operations) in EECS and 0.006% of all operations (34,076 operations) in DEAS. Many of these would not be cross-server, because they are among directories nearby in the heirarchy and likely to be managed by the same server.

Figure 6 shows that cross-directory RENAMEs in real-world workloads are bursty. Specifically, in the EECS trace, there is a 70% chance that a second RENAME will be seen within one second of the first; this probability is 93% in the DEAS trace. Such burstiness suggests that the overhead of dynamic distribution can be amortized across multiple cross-server RENAMEs as long as all of them involve the same two metadata servers. Cross-directory RENAMEs are not as bursty in the CAMPUS trace as in the EECS and DEAS traces, but the overhead of dynamic distribution in a workload resembling CAMPUS is not likely to be significant as RENAMEs are extremely rare in this trace.

6 Future work

We plan to improve this work in several ways. We plan to evaluate the benefits of multiple concurrent redistributions, dynamic resizing of tables, and concurrent queries during cross-server operations. Because cross-server operations are rare, these features are not needed for average-case performance but rather for to broaden the scope of applicability—we would like to demonstrate in a graph similar to Figure 5 that this approach works well when cross-server operations are over 1% or more of load. We plan to implement and measure other cross-server operations, such as hourly snapshot, and we would like to compare directly the performance and complexity of two-phase commit and other mechanisms. We plan to demonstrate

PSfrag replacements

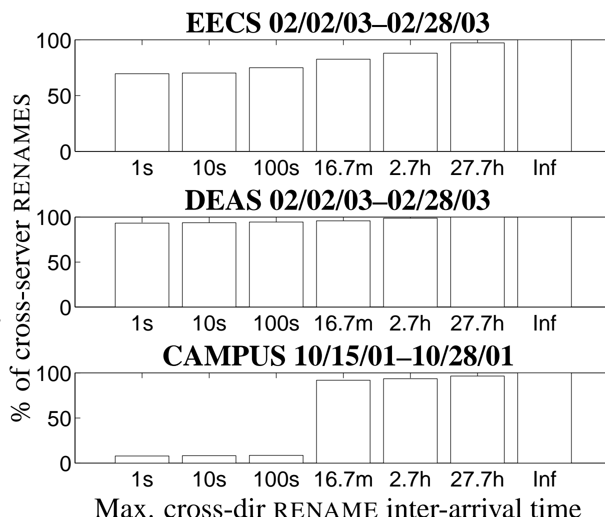


Figure 6: **These graphs show the percentage of cross-directory RENAME operations vs. maximum inter-arrival time**. Most cross-directory RENAMEs occur in bursts – in the EECS and DEAS trace, 70% and 93% of all cross-directory RENAMEs occur within one second of each other. The inter-arrival times of cross-directory renames is larger in the CAMPUS trace, but renames are also much less frequent in this trace than the EECS and DEAS trace.

how to efficiently and expeditiously revoke the authority of an errant metadata server. Though this work is orthogonal to distributed workload redistribution, we would like to distribute the root metadata server's functions. Ideally, we would like to demonstrate the simplicity of our implementation on a third-party object-based storage system or similar distributed system. We would also like to evaluate trace data from an environment where many clients access many objects in parallel.

7 Summary

Cross-server operations are a long-standing source of headaches for implementers of scalable distributed file systems. Dynamic redistribution can be used to eliminate cross-server operations, eliminating the need for protocols dedicated to handling them. Although it is much more heavy-weight than a dedicated cross-server update protocol, the use of dynamic redistribution can simplify the system with minimal impact of performance. Experiments with our prototype implementation show that overall performance is not affected when fewer than 0.01% of operations are (initially) cross-server. Analysis of NFS traces shows that cross-server operations are much rarer than that—fewer than 0.006% of client requests could possibly be cross-server in the traces analyzed. We believe that this approach to handling infrequent cross-server operations is very promising for distributed file systems and, perhaps, for other scalable distributed systems as well.

Acknowledgements

We would like to thank Craig Soules for his comments and feedback. We also thank Dan Ellard and Margo Seltzer for sharing the NFS traces.

References

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002), pages 1–15. USENIX Association, 2002.

[3] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5):109–126, 1995.

[4] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 173–186. ACM, 1998.

[5] Ann L. Chervenak, Vivekanand Vellanki, and Zachary Kurmas. Protecting file systems: a survey of backup techniques. *Joint NASA and IEEE Mass Storage Conference* (March 1998), 1998.

[6] Don Clark. Los Alamos Lab Picks Panasas for Data Storage. *Wall Street Journal Online*, 20 Oct 2003.

[7] Clustered MDS, Apr 2006. https://mail.clusterfs.com/wikis/lustre/DesignChanges.

[8] The Lustre file system roadmap, May 2006. http://www.clusterfs.com/roadmap.html.

[9] Selecting a Scalable Cluster File System, Nov 2005. Cluster File Systems, Inc.

[10] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003), pages 203–216. USENIX Association, 2003.

[11] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer. *The utility of file names*. Technical report TR-05-03. Harvard University, March 2003.

[12] Daniel Ellard and Margo Seltzer. New NFS tracing tools and techniques for system analysis. *Systems Administration Conference* (San Diego, CA), pages 73–85. Usenix Association, 26–31 October 2003.

[13] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[14] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Seattle, WA, 15–18 June 1997). Published as *Performance Evaluation Review*, **25**(1):272–284. ACM, June 1997.

[15] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994.

[16] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988.

[17] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[18] Lustre, Apr 2006. http://www.lustre.org/.

[19] Marshall Kirk McKusick. Running 'fsck' in the background. *BSDCon Conference* (San Francisco, CA, 11–14 February 2002), 2002.

[20] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002). USENIX Association, 2002.

[21] When to Use Transactional NTFS, Apr 2006. http://msdn.microsoft.com/library/en−us/fileio/fs/when_to_use_transactional_ntfs.asp.

[22] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 117–129. USENIX Association, 2002.

[23] Reiser4 Transaction Design Document, Apr 2006. http://www.namesys.com/txn-doc.html/.

[24] Yasushi Saito and Christos Karamanolis. *Name space consistency in the Pangaea wide-area file system*. HP Laboratories SSP Technical Report HPL–SSP–2002–12. HP Labs, December 2002.

[25] Frank Schmuck and Roger Haskin. GPFS: a shared-disk file system for large computing clusters. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 231–244. USENIX Association, 2002.

[26] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, **22**(4):299–319, December 1990.

[27] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5), 1995.

[28] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997.

[29] R. O. Weber. Information technology - SCSI Object-Based Storage Device Commands (OSD). Technical Council Proposal Document T10/1355-D, ANSI Technical Committee T10, July 2004.

[30] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.