

# Reusing dynamic redistribution to eliminate cross-server operations and maintain semantics while scaling storage systems

Shafeeq Sinnamohideen  
*Carnegie Mellon University*

December 3, 2007

## **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 thesis explores an alternate approach, with simplicity as a goal, 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, preliminary analysis of NFS traces indicates that such operations are extremely rare in file system workloads. Thus, when dynamic redistribution functionality exists in the system, cross-server operations can be handled with very little additional implementation complexity.

## **1 Introduction**

Transparent scalability is a goal of many distributed systems. That is, it should be possible to increase both capacity and performance by adding servers and spreading data and work among them. Furthermore, it is desirable that client applications and users are presented a consistent set of semantics, regardless of which servers are hosting which data.

In the case of file systems, many designs scale by partitioning the set of files across the set of servers. Each file is managed by a particular server, and accesses to files managed by different servers are completely independent. The vast majority of operations affect only a single file, so this approach works well in the common case. A few operations, such as a cross-directory rename or a snapshot, affect more than one file or directory and so may involve files managed by two distinct servers. A transparently scalable system would provide the same semantics in this case as in the case where all files are on the same server. Providing strong semantics on a single server is

relatively easily done using techniques like local locking and write-ahead logging, but doing so is more difficult when multiple servers are involved.

Many existing systems do not provide identical semantics in this case. Applications, on the other hand, frequently rely on specific consistency semantics (e.g., atomicity of rename) and have no convenient way of knowing whether any given operation will get the same-server or cross-server semantics. Additionally, systems that aim to scalably support legacy protocols, such as NFSv3 [3] or CIFS [8], must maintain the semantics defined by the protocol, regardless of whether any application relies on them.

Systems that do provide transparent scalability use some sort of distributed protocol to handle operations that involve multiple servers. In the case where all objects are accessible from any server, this is often done by having one server acquire locks on all objects involved, update the objects and any logs as necessary, and then release the locks. The underlying distributed lock manager will trigger the appropriate cache invalidations to maintain consistency between servers. GPFS uses such an approach. An alternative, frequently used when servers do not share a common storage pool, is to have each server execute only the portion of the operation that pertains to it. All objects are then updated to their final state using a multi-phase commit protocol to ensure atomicity.

These solutions, while effective, are also complex to implement, debug, and verify, particularly for the cases involving failures. Furthermore, in many workloads, cross-server operations occur very rarely. For example, the Farsite synthetic workload intended to represent common user actions, consisted of 2.1M operations, only 8 involved multiple servers [4]. Thus, the programmer effort traditionally involved in supporting transparent cross-server operations is out of proportion with its utilization in most workloads.

## 2 Thesis statement

This dissertation work will develop an alternate approach to handling cross-server operations. Specifically:

**Reusing dynamic redistribution to eliminate cross-server operations is an effective approach to transparently scaling decentralized storage systems**

Most scalable systems have mechanisms to dynamically redistribute objects across servers. This capability is used to ensure that server resources are utilized efficiently. For example, when a server is overloaded, some of the objects it's responsible for should be moved to other servers, moving the load associated with them. Similarly, when a server's disks are nearly full, some of its objects should be moved to other servers with more free space. There has been much work on load balancing policies, and the underlying mechanisms used for transferring objects are fairly straightforward.

I propose to use this same redistribution mechanism to support multi-server operations, instead of a dedicated protocol. If servers can only process requests for objects they are responsible for, all the objects involved must be the responsibility of the same server. If this precondition is not

satisfied, the system will redistribute the required objects until it is satisfied, and only then execute the operation. Objects that were moved may be returned to their original servers immediately, or they can be left until load-balancing policy dictates that they be moved again. All of the inter-server communication is encapsulated in the dynamic redistribution mechanism, which already exists for other purposes, saving implementation and debugging time.

This approach will likely involve more overhead than a dedicated cross-server protocol. For example, the data structures for storing contents and mappings may not permit the redistribution of a single object; a collection of objects may have to be moved together. The reduced performance, however, may be an acceptable tradeoff for the reduced implementation complexity of reusing existing code. This is especially true if cross-server operations are expected to be relatively infrequent, as is the case in distributed file systems. If cross-server performance is important, the dynamic redistribution mechanism can be optimized in various ways to tune it for its new use, rather than just load-balancing. Taken to the logical limit, these optimizations can become as complex as a dedicated cross-server protocol. However, this affords a system designer a spectrum of performance versus complexity options. And unlike a dedicated protocol, it allows for incrementally improving the system from “simple but slow” to “complex and fast” while leaving the rest of the system unaffected.

### 3 Argument

The efficacy of using dynamic redistribution to implement cross-server operations depends on two factors :

- The performance of using redistribution is reasonable compared to using a dedicated protocol.
- Supporting cross-server operations using redistribution requires less implementation effort than a dedicated protocol for the same purpose.

We expect some performance penalty, in many cases, we expect this penalty will be small enough to be outweighed by the savings in implementation cost. System architects make many such tradeoffs in the course of designing a system. If the architect’s goal is absolute performance at any cost, then any performance penalty is unacceptable. Frequently, however, the goal is to maximize “performance for a fixed cost”(e.g., a deadline) or “performance per unit of effort”. System builders often have a list of potential performance enhancements they have not yet had time to pursue. The performance penalty will be acceptable as long as it is inconsequential, or if the effort saved in supporting cross-server ops can then used to optimize other aspects of the system to regain more performance than was lost. Amdahl’s law suggests that this will be likely — the vast majority of operations are single-server, so a small improvement on each can overcome a large slowdown on the rare cross-server operation.

### 3.1 On the performance penalty

The performance penalty is expected to be small in many common environments. The size of the penalty is governed by how frequent cross-server operations are and how expensive each one is. In most file system scenarios, the frequency of cross-server operations is very low. Thus, even though each cross-server operation may be expensive, the effect on overall throughput will be negligible. The latency of redistribution, however, will be added directly to the latency of the cross-server operation.

#### 3.1.1 On the frequency of multi-file operations

The frequency of multi-file operations is a property of a given workload. These operations exhibit the well-known properties of spatial and temporal locality, just as single-file operations do. Thus, traditional techniques to improve overall performance, such as grouping files by directory, subtree, or access pattern and placing an entire group on the same server, result in most multi-file operations affecting multiple files on the same server. The only times an actual multi-server operation would occur in practice are in the rare case of a multi-file operation that affects files far enough apart in the directory tree to be on separate servers.

Examples of multi-file operations include `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.<sup>1</sup>

Many applications assume atomic `RENAME`, or at least that the destination will be either 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.

When the object being renamed is a directory, the file system must ensure that the directory is not renamed into one of its own subdirectories, which would create a disconnected loop in the namespace. Detecting this requires traversing the directory tree upwards from the destination to determine if it includes the target of the rename.

Creation and deletion of hard links are also multi-file operations that affect the directory being linked into, and the file being linked. Links created in the same directory should rarely result

---

<sup>1</sup>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.

in cross-server operations. But, for links created in other directories, it is possible for the file and the directories with names for it 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. Additionally, if the contents of a directory are split across servers, an impossibility in many systems, creating a new link may require atomically searching all of the directory to ensure it does not already exist before creating it.

Finally, modern file systems, such as NTFS [10] and Reiser4 [12], 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 application installation, application upgrades, and source control applications. The compound operations in NFSv4 and CIFS are a special case where an intermediate state corresponding to the partial sequential execution of the compounded operations is valid.

### 3.1.2 On the cost of cross-server operations

The cost of a cross-server operation depends on the cost of performing a redistribution, which in turn depends on how much data needs to be moved and the mechanisms for moving it.

For the purposes of supporting cross-server ops, usually only a single item needs to be redistributed. Many systems, however, cannot redistribute at the granularity of a single item. Single item redistribution requires that the mechanism used to map items to servers must handle the case where every item is potentially on a different server, and the servers must store items such that individual ones can be transferred separately. Additionally, because each server could be responsible for a sparse set of items, the simple task of determining what the “next” item is is complicated by the fact the next item could always be on another server. All of these add to the basic complexity of the system by requiring more complex data structures, and increasing overheads even when no redistributions are in progress.

The approach most systems take is to store items in the most convenient structure, and redistribute at the granularity supported by that structure. If items are stored in a B-tree, redistributing at a page granularity makes sense, since the tree can already support adding, removing, reading, and writing pages. Similarly, systems that maintain hierarchical directories may choose to store some part of an item within the structure that used for creating the hierarchy. For instance, Ceph [13] stores file inodes in the directory file rather than the traditional separate inode table. If directories are indivisible, it implies that the directory and all items in it must be redistributed together. Again, because all the items are stored together, moving them together is logical. Taken further, systems like AFS [6] and Coda [7] store both the names and contents of a subtree of items together. These systems can only redistribute at a volume granularity. These approaches have the added benefit that spatial locality suggests that items nearby in the hierarchy are likely to be used

together. On the other hand, because distribution is tied to the namespace, and the namespace is controlled by users, users can indirectly constrain redistribution decisions.

Depending on the system architecture, redistributing files either involves copying them over the network from one server to another, or relies on storing all files in a common storage pool, in which case redistribution merely involves a logical hand-off of responsibility. If shared storage is used, the latency for the logical hand-off can be expected to be on the order of that for a distributed transaction (since the hand-off is a simple distributed transaction itself), and thus have negligible effect on throughput. If shared storage is not available, then the latency will be heavily dependent on the size of the unit of redistribution. If only metadata needs to be transferred, and the unit is reasonably sized (a directory or a small number of directories), the transfer latency can also be small enough. Fortunately, it seems likely that one or both of these conditions will be true in most future storage systems.

In particular, consider the typical system that uses redistribution for load-balancing. The load controller in this system has the goal of ensuring that every server's load is within a certain threshold of average, say 10%. If a particular server's load is beyond this threshold, the load balancer would need to redistribute some of that server's load to another, less loaded, server. Assuming load is uniformly distributed across objects, the redistribution mechanism would need to support a granularity no coarser than 20% of the objects on the overloaded server. Removing this 20% will result in a server underloaded by at most 10%. However, if there is no destination server that can accept all of these objects without becoming overloaded itself, the objects will have to be split across many lightly underloaded servers, requiring a finer granularity. Additionally, if groups of objects (such as a subtree) are constrained to be moved together, or if load is not uniform across objects, even finer granularities may be necessary because a single group may be responsible for a greater fraction of the load than its numerical proportion. xFS, for example, recommends a granularity of 10% or finer in order to support even coarse-grained load-balancing [2] over a relatively small number of servers. As future systems consider scale to larger numbers of servers, the required granularity gets finer (re-balancing after adding 1 node to a 100 node system requires redistributing 1% from each server). Thus, it seems reasonable to expect that a typical system would support a granularity at least as fine as 1% of a server's contents. Since it is unlikely more than a handful of cross-server operations will be in progress concurrently, the maximum load imbalance caused by redistributing to service multi-object operations is only a few percent.

At the same time, the availability of standard, commodity, object-based storage devices, are making direct-access storage architectures popular for large-scale storage systems. Data in these systems resides on a large number of network-attached storage bricks, each of which is accessible to all (or perhaps only a subset of) servers. Thus, in such a system, it would be logical to implement redistribution as a logical hand-off of control over portions of the shared storage. Note that some soft server state, such as callback or active filehandle state, may still need to be copied from server to server. Redistribution can, in these cases, be expected to be fast.

## 3.2 On the reduction in complexity

Both redistribution and a dedicated cross-server operation mechanism require distributed protocols. Redistribution, however, is simpler to implement than generalized distributed transactions, particularly if the system was not designed from the ground up to support either. Furthermore, the vast majority of systems include redistribution as a part of their basic functionality. This subsection will examine the reasons in more detail.

### 3.2.1 Why are distributed transactions hard?

Traditionally, cross-server operations are implemented using a distributed transaction protocol, such as a two-phase commit [5]. Since each server already must implement atomic single-server operations, usually by using write-ahead logging and rollback, the distributed transaction system can be built on top of the local transaction system. A transaction affecting two servers would first add a “prepare” entry to both of the logs, covering the update of the respective data items. If both servers successfully “prepared”, the transaction is finalized with a “commit” entry to both logs. If the “prepare” did not succeed on all servers, each server must examine its log and roll back its state to that of the beginning of the transaction. Crash recovery, however, is now much more complicated: with single-server transactions, it is sufficient to examine the log and undo any incomplete transactions. With more than one server, it is possible for some servers to crash and others survive. If one crashed while “preparing”, all servers must rollback that transaction, as mentioned before. If one crashed between the “prepare” and the “commit”, the recovering server does not know if the “prepare” succeeded on all other servers. By examining the logs of the other servers, the recovering server can determine if the “commit” appears in any log, in which case it should be added to the recovering log, or if it did not commit and should be undone locally. Any step involving communication with other servers may fail, and if other servers have crashed, it may not be possible to proceed until they are online as well.

Distributed transactions may complicate other aspects of the system as well. Consider a simple operation that reads and updates two items, each on different servers. In order to prevent a single-server operation on either server from modifying one of the items between the read and write phase of the cross-server operation, the server executing the transaction must lock both items for the duration of the transaction. This must also be done for a single-server transaction, but all potential contention is local to a single server. A lock held by a different server introduces the potential for the lock holder to crash independently of the server managing the lock. While there are many existing techniques, such as leases, to handle this situation, a lock recovery scheme is simply not needed when locks can only be local to a server. Handling non-crash faults, such as intermittent networks or Byzantine servers, adds far more complexity to any distributed protocol [4].

As can be seen, most of the additional complexity is in the recovery path. Not only must the recovery path handle recovery from a wide variety of errors or crashes, it must also handle errors during

recovery. This leads to a large number of cases that must be detected and handled correctly. Since errors in general are rare, any particular error is even rarer, which means bugs in the fault-handling path may be triggered rarely and be even harder to reproduce. This places more reliance on test harnesses, which must be crafted to exercise each of the many error conditions and combinations thereof.

### 3.2.2 Why is reusing redistribution simpler ?

Redistribution involves 3 main steps :

- The source server must stop serving the items to be moved
- The responsibility for the items must be moved to the destination
- The destination server must start serving the items

The complete redistribution process must be atomic - in the face of any crash failure, either the source or destination must be completely responsible for and able to serve the items in question, and, at all times, all parties agree on which server is responsible for which item. A non-atomic redistribution raises the possibility of items disappearing forever or different versions of the same item being available to clients. As this is clearly undesirable, most systems implement redistribution using a simplified form of distributed transaction [4, 13, 6]. The presence of a authoritative mechanism to maps items to servers simplifies the task because it provides a central point of coordination, instead of having to rely on distributed logs.

The first step is relatively straightforward — the source server must either complete or abort any in-progress operations and stop accepting any new ones. The mechanism that maps items to servers must be updated so that clients know that the items in question are being moved. Clients can use this knowledge to either hold requests until the move is complete or, if the destination is known, to direct them to the new server. This phase can be viewed as analogous to acquiring locks for a distributed transaction, except that the lock management is centralized.

Moving responsibility for the items is the most complex part of redistribution. If all servers share a common storage pool, transferring responsibility may simply involve updating each server's idea of which items it is responsible for. If the destination server later receives a request for one of the moved items, it can simply read it from the shared storage. If servers do not share storage, the contents of the items being moved must be packaged up and transferred over the network before being written to the destination server's local storage. In either case, this transfer of responsibility must be atomic.

Finally, once the destination server has received all that it needs to serve the items, the mapping of items to servers is updated so that client requests are directed to the new server. That server can then serve the transferred items as it would any other. Since the server has not served these items before, accesses to those items will suffer all the penalties, such as cache misses, common to

all not-previously-accessed items. This step can be thought of as being equivalent to the commit phase of a distributed transaction. Because the mapping mechanism is a single logical entity, the “commit” requires only a single update. This contrasts with a general distributed transaction, in which case logs on each server would need to be updated. This simplifies both the common case, and the recovery case, because only the single, authoritative, mapping service needs to be consulted to determine the most recent state.

In contrast to a system that uses a dedicated protocol for cross server operations, redistribution has both a simpler recovery case, and redistribution is already implemented as part of basic system functionality. Thus when considering how to add cross-server operation support to a system, adding a dedicated protocol requires the additional code to implement all of it, including the distributed transaction and all its recovery cases. Reusing redistribution, on the other hand, does not require any changes to the existing redistribution-only mechanism that is included in the basic functionality of the system. It only requires relatively simple code to detect the conditions in which redistribution will be required and trigger the existing mechanism. Failures during redistribution will be handled by the existing failure paths. The only additional failure case is that a cross-server op can now fail because the redistribution it required failed, which is trivial to handle.

Additionally, because redistribution is primarily a coarse-grained internal operation between two servers in the system, the architect has some freedom to design the redistribution protocol and its semantics, persistent data structures, and server structures to best suit their combined requirements. Thus, it is possible to make a small change in one (e.g., the semantics of redistribution) in order to simplify the others. With a dedicated protocol for cross-server operations, the choices are constrained by the semantics required by the external, client-initiated, operations.

One could implement a generic distributed transaction mechanism and use it to support both redistribution and cross-server operations. Such a mechanism would have to efficiently support both the small transactions typical of cross-server operations and the very large transactions typical of redistribution. Compared to a system tailored for each specific use, a generic system would either involve compromises that reduced its performance, or provide optimizations for the two extremes, which would improve performance at the cost of extra complexity. Thus, it stands to reason that such a system would require more effort to implement than one tailored specifically for only one purpose, but less than implementing two separate dedicated systems.

While either option is more work overall than having a generic mechanism from the beginning, it may be a reasonable tradeoff to reduce time to market by implementing on only the required functionality first even if it means more work later. Such an approach will certainly be simpler than the double effort of implementing separate mechanisms. It may even be simpler to build a redistribution-only mechanism with cross-server ops on top of it than a generic mechanism. Thus we expect this approach to be very suitable for retrofitting cross-server ops into legacy systems, as well as a viable option for clean-sheet designs.

### 3.2.3 Why does redistribution always exist ?

The purpose of redistribution is to move objects from one server to another. The ability to do so is required in order to provide several desired system capabilities. For instance, moving objects are required as a part of : decommissioning a server, populating a newly commissioned server, addressing capacity limitations o a single server, matching workloads to the server best suited for them, ensuring that data of a certain security level or “owned” by a particular entity remains on certain servers. Additionally, automated management features can perform some of these functions in current systems [11, 9], and will increasingly do so in the future [1].

Managing large-scale storage systems would be very difficult without redistribution - at the very least, hardware replacement and consolidation as individual device capacity grows, must be accounted for. Almost every storage system has a way of performing redistribution, in the worst case by backing up data on the original server, deleting it, and restoring on the destination server.

Such offline redistribution, however, is obtrusive to clients, which will notice periods of data unavailability. If the need for redistribution is rare, it can be scheduled to happen during announced maintenance periods. As a system gets larger, the need for redistribution increases, while the tolerance for outages decreases. For this reason, many systems can perform redistribution dynamically [11, 1, 2, 13, 6], leaving clients unaffected except for brief periods of unavailability.

Indeed, most of these systems consider redistribution to be a critical feature (the system will not fulfill its intended purpose without it) and cross-server operations to be a desired enhancement (the lack of it will only rarely become apparent in current workloads and system sizes). Thus, systems such as AFS and Lustre implement simplified distributed transactions for supporting redistribution alone and do not implement cross-server operations in any way. As these systems increase in scale, the lack of cross-server operations will become more apparent, and architects will have to choose a means of addressing it.

## 4 Validation

I will validate my thesis by demonstrating that it is a good choice for a variety of real system scenarios. I will do so by :

- Quantifying the frequency of potentially multi-server operations seen in traces of actual file system activity and common benchmarks.
- Quantifying the performance of this approach as a function of system architecture and frequency of multi-server operations.
- Demonstrating the simplicity of this approach through a comparison, in terms of lines of code, against an implementation of cross-server operations using 2-phase commit.

- Highlighting the complexity of 2-phase commit by examining code or literature on systems that implement it, and soliciting anecdotal evidence from the implementors.

Each trace analysis will be performed by simply counting the number of multi-file operations in the trace. A more accurate estimation of whether a multi-file operation (RENAME or COPY) will involve multiple servers requires determining the distance in the directory heirarchy between the source and destination directories. In the case of CIFS traces, this information can be determined from the request itself, but in the case of NFS traces, the full tree is not known, and must be reconstructed. Once the full tree is known, the effect of various namespace partitioning schemes on the number of CREATE and DELETE operations that cross server boundaries can be explored.

The performance comparison will be done by implementing this approach and a dedicated protocol in the Ursa Minor Metadata Server. The server will support varying architectural parameters, such as the method of redistribution and granularity of redistribution, allowing the influence of these parameters to be explored. A prototype implementation will also illustrate the interactions with loadbalancing and other workloads.

The implementation of either approach in Ursa Minor will provide a comparison for the difficulty of implementing each in a research system. The results should be a conservative approximation of the difficulty of doing so in a commercial system. In both cases, the major complexity is in the error-handling and recovery paths. When reusing redistribution, most of the errors will be incorporated in the redistribution code, which (as discussed earlier) I assume the system already includes. When using 2-phase commit, the error handling and recovery are part of the additional code needed to handle cross-server operations. Thus, if the main difference between the research and commercial systems is in more robust error handling and code quality, reusing redistribution will have an even greater advantage in the commercial system.

## References

- [1] M. Abd-El-Malek, G. R. Goodson, G. R. Ganger, M. K. Reiter, and J. J. Wylie. Lazy verification in fault-tolerant distributed storage systems. Symposium on Reliable Distributed Systems. IEEE, 2005.
- [2] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 29(5):109–126, 1995.
- [3] B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813 - NFS version 3 protocol specification*. RFC-1813. Network Working Group, June 1995.
- [4] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006.
- [5] J. N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.
- [6] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81. ACM, February 1988.
- [7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25. ACM Press, February 1992.
- [8] P. J. Leach. *A Common Internet File System (CIFS/1.0) Protocol (Working Draft)*. Technical report. Internet Engineering Task Force, December 1997.

- [9] Lustre, Apr 2006. <http://www.lustre.org/>.
- [10] When to Use Transactional NTFS, Apr 2006. [http://msdn.microsoft.com/library/en-us/fileio/fs/when\\_to\\_use\\_transactional\\_ntfs.asp](http://msdn.microsoft.com/library/en-us/fileio/fs/when_to_use_transactional_ntfs.asp).
- [11] Panasas, Inc., May 2004. <http://www.panasas.com/>.
- [12] Reiser4 Transaction Design Document, Apr 2006. <http://www.namesys.com/txn-doc.html/>.
- [13] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. Long. Ceph: A scalable, high-performance distributed file system. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006.