Hierarchical Replication Control in a Global File System

Jiaying Zhang and Peter Honeyman
Center for Information Technology Integration
University of Michigan at Ann Arbor
jiayingz@eecs.umich.edu honey@citi.umich.edu

Abstract

We develop a consistent mutable replication extension for NFSv4 tuned to meet the rigorous demands of large-scale data sharing in global collaborations. The system uses a hierarchical replication control protocol that dynamically elects a primary server at various granularities. Experimental evaluation indicates a substantial performance advantage over a single server system. With the introduction of the hierarchical replication control, the overhead of replication is negligible even when applications mostly write and replication servers are widely distributed.

1. Introduction

Grid-based scientific collaborations are characterized by geographically distributed institutions sharing computing, storage, and instruments in dynamic virtual organizations [1, 2]. By aggregating globally distributed resources, Grid middleware provides an infrastructure for computations far beyond the scope of a single organization.

Grid computations feature high performance computing clusters connected with long fat pipes, a significant departure from the traditional high-end setting of a collection of nodes sited at one location connected by a fast local area network. This difference introduces new challenges in storage management, job scheduling, security provision, etc., stimulating growing research in these areas. In particular, the need for flexible and coordinated resource sharing among geographically distributed organizations demands efficient, reliable, and convenient data access and movement schemes to ease users' efforts for using Grid data.

The state of the art in Grid data access is characterized by parallel FTP driven manually or by scripts [3]. FTP has the advantage of following a strict and simple standard and widespread vendor support. However, FTP has some shortcomings.

- Applications must explicitly transfer a remote file in its entirety to view or access even a small piece of it, then transfer it back if the file is modified.
- Consistent sharing for distributed applications is not supported.
- The distribution model also leads to long first-byte latency.

To overcome these problems, we developed an alternative for distributed filing on the Grid that allows users and applications to access widely distributed data as simply and efficiently as they access them locally.

Recent advances in Internet middleware infrastructure — notably, broad support for NFSv4 [4, 5] — offer remarkable opportunities for virtual organizations to share data through a unified global file system. Designed with Internet data management in mind, NFSv4 has the potential to meet the requirements of widely distributed collaborations. As a distributed file system protocol, NFSv4 allows users to access data with traditional file system semantics. NFSv4 guarantees "close-to-open" consistency, i.e., an application opening a file is guaranteed to see the data written by the last application that writes and closes the file. This model, which proves adequate for most applications and users [6], can also serve as an enabling feature for re-using existing software on the Grid.

In spite of these advantages, extending NFSv4 access to a global scale introduces performance challenges. Our evaluation indicates that conventional NFS distribution — multiple clients connected to storage elements through a common server — cannot meet Grid performance requirements when computational elements are widely distributed [7]. To overcome this problem, we developed a replication protocol for NFSv4 that allows placement of replication servers near compute nodes [8]. The protocol supports NFSv4 semantics exactly and requires no client-side extensions, which simplifies deployment in wide area networks.

Our replication extension to NFSv4 coordinates concurrent writes by dynamically electing a primary server for client updates. When no writers are active, our system has the performance profile of systems that support

read-only replication. Unlike read-only systems, though, we also support concurrent write access without compromising NFSv4 consistency guarantees. Security of the protocol follows from the use of secure RPC channels, mandatory in NFSv4, for server-to-server communication. Furthermore, the system can automatically recover from minority server failures, offering higher availability than single server systems.

Our earlier replication protocol breaks new ground in performance and availability for read-dominant applications, yet further analysis exposes a considerable performance penalty for large synchronous writes, bursty directory updates, and widely separated replication servers — data access patterns characteristic of Grid computing. The observed performance penalty is mainly due to the cost of guaranteeing durability and the cost of synchronization. Specifically, the durability requirement delays the response to a client update request until a majority of the replication servers have acknowledged the update. This provides a simple recovery mechanism for server failure but synchronous writes and directory updates suffer when replication servers are far away. The synchronization requirement, which amounts to an election for consensus gathering, also delays applications — especially when they emit a burst of metadata updates — while waiting for distant replication servers to vote.

We assume (and observe) that failures are rare in practice. Furthermore, the computation results by Grid applications can usually be reproduced by simply re-executing programs or restarting from a recent checkpoint. This suggests that we may relax the durability requirement to improve performance for synchronous updates. Instead of automatically guaranteeing durability to a client, we may elect to report failure to the application immediately by making the data under modification inaccessible. The application can then decide whether to wait for server recovery or to regenerate the computation results.

To reduce the cost of synchronization, we introduce a hierarchical replication control protocol that allows a primary server to assert control at granularities coarser than a single file or directory, allowing control over an entire subtree rooted at a directory. This amortizes the cost of synchronization over multiple update requests.

In this paper, we describe these extensions in detail. In particular, we focus on the design, implementation, and evaluation of the hierarchical replication control protocol we developed. The evaluation for using the described replicated file system to support Grid computing and the performance comparisons with GridFTP are presented in a companion paper [7].

The remainder of the paper proceeds as follows. Section 2 describes our earlier work in developing a replication control protocol that coordinates concurrent writes by electing a primary server at the granularity of a single file or directory and the extensions we made to reduce the cost of guaranteeing durability. We refer to it as the *fine-grained replication control protocol* in the following discussion. In Section 3, we introduce a *hierarchical replication control protocol* that allows a primary server to assert control at various granularities to amortize the performance cost of primary server election over more update requests. In Section 4, we examine the performance of these protocols with a file system benchmark. In Sections 5 and 6, we discuss related work and conclude.

2. Fine-grained Replication Control

This section reviews the design of a mutable replication protocol for NFSv4 that guarantees close-to-open semantics by electing a primary server for client updates at the granularity of a single file or directory [8].

Briefly, the system works as follows. When a client opens a file for writing, the replication server to which it connects invokes a replication control protocol, a server-to-server protocol extension to the NFSv4 standard.[1] First, the server arranges with all other replication servers to acknowledge its primary role. Then, all other replication servers are instructed to forward client read and write requests for that file to the primary server. The primary server distributes (ordered) updates to other servers during file modification. When the file is closed (or has not been modified for a long time) and all replication servers are synchronized, the primary server notifies the other replication servers that it is no longer the primary server for the file.

Directory updates are handled similarly, except for the handling of concurrent writes. Directory updates complete quickly, so a replication server simply waits for the primary server to relinquish its role if it needs to modify a directory undergoing change. For directory updates that involve multiple objects, e.g., renaming, a server must become the primary server for all objects. To prevent deadlock, we group these update requests and process them together.

Two requirements are necessary to guarantee close-to-open semantics. First, a server becomes the primary server for an object only after it collects acknowledgements from a majority of the replication servers. To guarantee this, we implement a leader election algorithm that achieves the lower time bound of fast Consensus [9]. The pseudo code of the implementation and the failure recovery mechanisms are provided in our technical report [10]. Second, a primary server must ensure that *all* working replication servers have acknowledged its role when a written file is closed, so that subsequent reads on any server reflect the contents of a file when it was closed.

[1] An application can open a file in write mode without actually writing any data for a long time, e.g., forever, so the procedure is delayed until the client makes its first write.

The second requirement is satisfied automatically if the client access to the written file lasts longer than the duration of the primary server election. However, an application that writes many small files can suffer non-negligible delays. These files are often temporary files, i.e., files that were just created (and are soon to be deleted), so we allow a new file to inherit the primary server that controls its parent directory for file creation. Since the primary server does not need to propose a new election for writing a newly created file, close-to-open semantics is often automatically guaranteed without additional cost.

A primary server is responsible for distributing updates to other replication servers during file or directory modification. In our earlier version of the protocol, we required that a primary server not process a client update request until it receives update acknowledgements from a majority of the replication servers [8]. With this requirement, as long as a majority of the replication servers is available, a fresh copy can always be recovered from them. Then, by having all active servers synchronize with the most current copy, we guarantee that the data after recovery reflects all acknowledged client updates, and a client needs to reissue only its last pending request after switching to a working server.

The earlier protocol transparently recovers from a minority of server failures and balances performance and availability well for applications that mostly read. However, performance suffers for scientific applications that consist of many synchronous writes or directory updates and replication servers that are far away from each other. Meeting the performance needs of Grid applications requires a different trade-off.

Failures occur in distributed computations, but are rare in practice. Furthermore, the results of most scientific applications can be reproduced by simply re-executing programs or re-starting from the last checkpoint. This suggests a way to relax the costly update distribution requirement so that the system provides higher throughput for synchronous updates at the cost of sacrificing the durability of data undergoing change in the face of failure.

Adopting this strategy, we allow a primary server to respond immediately to a client write request before distributing the written data to other replication servers. Thus, with a single writer, even when replication servers are widely distributed, the client experiences longer delay only for the first write (whose processing time includes the cost of primary server election), while subsequent writes have the same response time as accessing a local server (assuming the client and the chosen server are in the same LAN). Of course, should concurrent writes occur, performance takes a back seat to consistency, so some overhead is imposed on the application whose reads and writes are forwarded to the primary server.

3. Hierarchical Replication Control

Even with an efficient consensus protocol, a server can be delayed waiting for acknowledgments from slow or distant replication servers. This can adversely affect performance, e.g., when an application issues a burst of metadata updates to widely distributed objects. Conventional wisdom holds that such workloads are common in Grid computing, and we have observed them ourselves when installing, building, and upgrading Grid application suites. To address this problem, we introduce a hierarchical replication control protocol that amortizes the cost of primary server election over more requests by allowing a primary server to assert control over an entire subtree rooted at a directory. In this section, we detail the design of this tailored protocol.

The remainder of this section proceeds as follows. Section 3.1 introduces two control types that a primary server can assert. One is limited to a single file or directory, while the other governs an entire subtree rooted at a directory. Section 3.2 discusses revisions to the primary server election needed for hierarchical replication control. Section 3.3 then investigates mechanisms to balance the performance and concurrency trade-off related to the two control types.

3.1 Shallow vs. Deep Control

We introduce nomenclature for two types of control: shallow and deep. A server exercising *shallow control* on an object (file or directory) **L** is the primary server for **L**. A server exercising *deep control* on a directory **D** is the primary server for **D** and all of the files and directories in **D,** and additionally exercises deep control on all the directories in **D**. In other words, deep control on **D** makes the server primary for everything in the subtree rooted at **D**. In the following discussion, when a replication server **P** is elected as the primary server with shallow control for an object **L**, we say that ***P** has shallow control on **L***. Similarly, when a replication server **P** is elected as the primary server with deep control on a directory **D**, we say that ***P** has deep control on **D***. Relinquishing the role of primary server for an object **L** amounts to revoking shallow or deep control on **L**. We say that a replication server **P** *control*s an object **L** if **P** has (shallow or deep) control on **L** or **P** has deep control on an ancestor of **L**.

We introduced deep control to improve performance for a single writer without sacrificing correctness for concurrent updates. Electing a primary server with the granularity of a single file or directory allows high concurrency and fine-grained load balancing, but a coarser granularity is suitable for applications whose updates exhibit high temporal locality and are spread across a directory or a file system. A primary server can process any

Upon receiving a client update request for object L
if **L** is controlled by self then serve the request
if **L** is controlled by another server then forward the request
else *// **L** is uncontrolled*
    if **L** is a file then request shallow control on **L**
    if **L** is a directory then
        if a descendant of **L** is controlled by another server then
            request shallow control on **L**
        else
            request deep control on **L**

Upon receiving a shallow control request for object L from peer server P
grant the request iff **L** is not controlled by a server other than **P**

Upon receiving a deep control request for directory D from peer server P
grant the request iff **D** is not controlled by a server other than **P**, and no descendant of **D** is controlled by a server other than **P**

Figure 1. Using and granting controls.

client update in a deeply controlled directory immediately, so it improves performance for applications that issue a burst of metadata updates.

Introducing deep control complicates consensus during primary server election. To guarantee that an object is under the control of a single primary server, we enforce the rules shown in Figure 1. We assume that the single writer case is more common than concurrent writes, so a replication server attempts to acquire a deep control on a directory whenever it can. On the other hand, we must not allow an object to be controlled by multiple servers. Therefore, a replication server needs to ensure that an object in a (shallow or deep) control request is not already controlled by another server. Furthermore, it must guarantee that a directory in a deep control request has no descendant under the control of another server.

To validate the first condition, a replication server examines each directory along the path from the referred object up to the mount point. If an ancestor of the object has a primary server other than the one that issues the request, the validation fails. Checking the second condition is more complex. Scanning the directory tree during the check is too expensive, so we do some bookkeeping when electing a primary server: each replication server maintains an *ancestry table* for files and directories whose controls are granted to some replication servers. An entry in the ancestry table corresponds to a directory that has one or more descendants whose primary servers are not empty. Figure 2 shows entries in the ancestry table and illustrates how the ancestry table is maintained.

An ancestry entry contains an array of counters, each of which corresponds to a replication server. E.g., if there are three replication servers in the system, an entry in the ancestry table contains three corresponding counters. Whenever a (deep or shallow) control for an object **L** is granted or revoked, each server updates its ancestry table by scanning each directory along the path from **L** to the mount point, adjusting counters for the server that owns the control. A replication server also updates its ancestry table appropriately if a controlled object is moved, linked, or unlinked during directory modifications.

A replication server needs only one lookup in its ancestry table to tell whether a directory subtree holds an object under the control of a different server. First, it finds the mapping entry of the directory from its ancestry table. Then, it examines that entry's counter array. If the counter on any replication server other than the one that issues the deep control request has a non-zero value, the replication server knows that some other server currently controls a descendant of the directory, so the replication server rejects the deep control request.

3.2 Primary Server Election with Deep Control

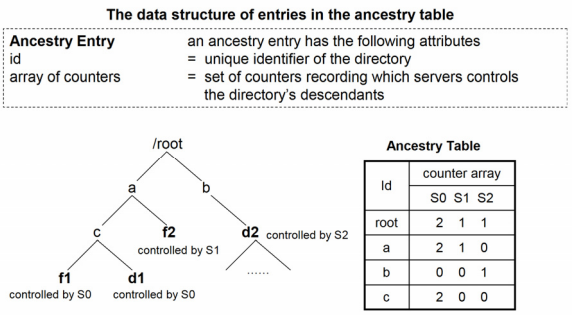
With the introduction of deep control, primary server election requests on two different objects can conflict if one of them wants deep control on a directory, as the example in Figure 3 illustrates. To guarantee progress during conflicts, we extend the consensus algorithm for primary server election as follows. When a replication server receives a deep control request for a directory **D** from a peer server **P** but cannot grant the control according to the rules listed in Figure 1, it replies to **P** with a NACK. A server downgrades a deep control request to shallow if it fails to accumulate acknowledgments from a majority of the replication servers. Proceeding with shallow controls only, the progress of primary server election is then governed by the original consensus algorithm [10].

3.3 Performance and Concurrency Tradeoff

The introduction of deep control introduces a performance and concurrency trade-off. A primary server can process any client update in a deep-controlled directory, which substantially improves performance when an application issues a burst of updates. This argues for holding deep control as long as possible. On the other hand, holding a deep control can introduce conflicts due to false sharing. In this subsection, we strive for balance in the trade-off between performance and concurrency when employing shallow and deep controls.

First, we assume that the longer a server controls an object, the more likely it will receive conflicting updates, so we start a timer on a server when it obtains a deep control. The primary server resets its timer if it receives a subsequent client update under the deep-controlled directory before the timeout. When the timer expires, the primary server relinquishes its role.

Second, recall that in a system with multiple writers, we increase concurrency by issuing a revoke request from



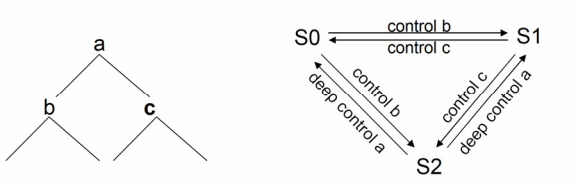
Consider three replication servers: S0, S1, and S2. Currently, S0 is the primary server of file f1 and directory d1, S1 is the primary server of file f2, and S2 is the primary server of directory d2. The right table shows the content of the ancestry table maintained on each replication server.

Figure 2. Maintenance of the ancestry table.

one server to another if the former server receives an update request under a directory deep-controlled by the latter. Locality of reference suggests that more revoke requests will follow shortly, so the primary server shortens the timer for relinquishing its role for that directory. Note that a replication server *does not* send a revoke request when it receives a directory read request under a deep-controlled directory. This strategy is based on observing that the interval from the time that a client receives a directory update acknowledgment and the time that other replication servers implement the update is small (because the primary server distributes a directory update to other replication servers immediately after replying to the client). This model complies with NFSv4 consistency semantics: in NFSv4, a client caches attributes and directory contents for a specified duration before requesting fresh information from its server.

Third, when a primary server receives a client write request for a file under a deep-controlled directory, it distributes a new shallow control request for that file to other replication servers. The primary server can process the write request immediately without waiting for replies from other replication servers, as it is already the primary server of the file's ancestor. However, with a separate shallow control on the file, subsequent writes on that file *do not* reset the timer of the deep controlled directory. Thus, a burst of file writes has minimal impact on the duration that a primary server holds a deep control. Furthermore, to guarantee close-to-open semantics, a replication server need only to check whether the accessed file is associated with a shallow control before processing a client read request, instead of scanning each directory along the path from the referred file to the mount point.

Fourth, a replication server can further improve its performance by issuing a deep control request for a directory that contains many frequently updated descendants if it observes no concurrent writes. This is easy to implement with the information recorded in the ancestry table: a replication server can issue such a request for directory **D** if it observes that in the ancestry entry of **D**, the counter corresponding to itself is beyond some threshold and the counters of all other replication servers are zero.



Consider three replication servers: S0, S1, and S2. Simultaneously, S0 requests (deep or shallow) control of directory b, S1 requests control of directory c, and S2 requests deep control of directory a. According to the rules in Figure 1, S0 and S1 succeed in their primary server elections, but S2's election fails due to conflicts. S2 then retries by asking for shallow control of a.

Figure 3. Potential conflicts in primary server election caused by deep control.

The introduction of deep control promises substantial performance benefits, but can also adversely affect data availability in the face of failure: if a primary server with deep control on a directory fails, updates in that directory subtree cannot proceed until the failed primary server is recovered. Recapitulating the discussion of false sharing above, this argues in favor of a small value for the timer. In the next section, we show that timeouts as short as one second are long enough to reap the performance benefits of deep control. Combined with our assumption that failure is uncommon, we anticipate that the performance gains of deep control far outweigh the potential cost of servers failing while holding deep control on directories.

4. Evaluation

In this section, we evaluate the performance of hierarchical replication control using the SSH-Build benchmark. The SSH-Build benchmark [11] runs in three phases. The *unpack* phase decompresses a tar archive of SSH v3.2.9.1. This phase is relatively short and is characterized by metadata operations on files of varying sizes. The *configure* phase builds various small programs that check the configuration of the system and automatically generates header files and Makefiles. The *build* phase compiles the source tree and links the generated object files into the executables. The last phase is the most CPU intensive, but it also generates a large number of temporary files and a few executables in the compiling tree.

We conducted the experiments that follow with a prototype implemented in the Linux 2.6.16 kernel. Servers and clients all run on dual 2.8GHz Intel Pentium4 processors with 1 MB L2 cache, 1 GB memory, and onboard Intel 82547GI Gigabit Ethernet card. The NFS configuration parameters for reading (rsize) and writing (wsize) are set to 32 KB, the recommended value for WAN access.

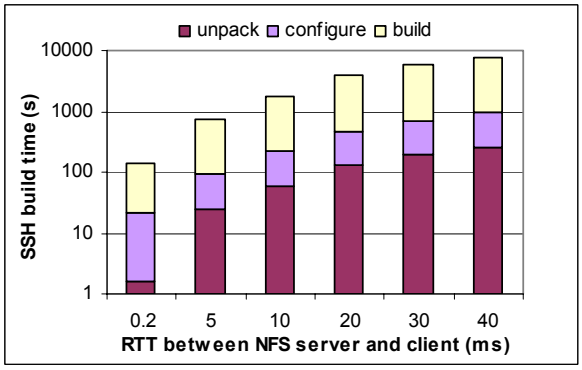


Figure 4. SSH build on a single NFSv4 server.

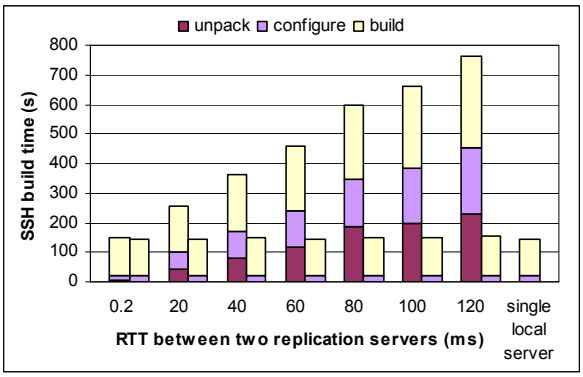


Figure 5. Fine-grained replication control vs. hierarchical replication control. The first column shows the time to build SSH using fine-grained replication control. The second column shows the time when using hierarchical replication control. For runs with hierarchical replication control, the primary server relinquishes deep control if it receives no client updates for one second.

We use Netem [12] to induce network latencies. Our experiments focus on evaluating the performance impact caused by WAN delays. Hence, we do not simulate packet loss or bandwidth limits in our measurements. Although not comprehensive, we expect that our settings resemble a typical Grid environment — high performance clusters connected by long fat pipes. All measurements presented in this paper are mean values from five trials of each experiment with a warm client cache; measured variations in each experiment are negligible.

Before diving into the evaluation of hierarchical replication, we look at performance when accessing a single distant NFSv4 server. Figure 4 shows the measured times (in log-scale) when we run the SSH-Build benchmark with an increasingly distant file server. In the graph, the RTT marked on the *X*-axis shows the round-trip time between the client and the remote server, starting with 200 μsec, the network latency of our test bed LAN. We find that the SSH build that completes in a few minutes on a local NFSv4 server takes hours when the RTT between the server and the client increases to tens of milliseconds. The experiment shows that it is impractical to execute update-intensive applications using a stock remote server. Network RTT is the dominant factor in NFS WAN performance, which suggests the desirability of a replicated file system that provides client access to a nearby server.

Next, we compare the time to build SSH using fine-grained replication control and hierarchical replication control with a local replication server and an increasingly distant replication server. The results, shown (in linear scale) in Figure 5, demonstrate the performance advantage of file system replication. Even with fine-grained replication control, adding a nearby replication server significantly shortens the time to build SSH, as expensive reads from a remote server are now serviced nearby. Moreover, we see dramatic improvement with the introduction of hierarchical replication control: the penalty for replication is now negligible, even when replication servers are distant.

In Section 3, we discussed the use of a timer for each deep-controlled directory to balance performance and concurrency but did not fix the timeout value. To determine a good value for the timer, we measure the time to build SSH for timeout values of 0.1 second, 0.5 second, and 1 second. Figure 6 presents the results. As the data shows, when we set the timeout value to one second, the SSH build with a distant replication server runs almost as fast as one accessing a single local server. Furthermore, almost all of the performance differences among the three timeout values come from the CPU intensive *build* phase. For the *unpack* and *configure* phases, which emit updates more compactly, even a tiny timeout value yields performance very close to that for single local server access. Of course, in practice the "optimal" timeout value depends on the workload characteristics of the running applications. However, the SSH build experiment suggests that a small timer value—a few seconds at most—captures most of the bursty updates.

So far, our experiments focus on evaluation with two replication servers. Generally, our system is designed to be used with a small number of replication servers, say, fewer than ten. Under this assumption, we do not expect performance to suffer when additional replication servers are added because a primary server distributes updates to other replication servers in parallel. To test this conjecture, we measure the time to build SSH as the number of replication servers increases in a LAN and in a simulated WAN. Figure 7 shows that performance is largely unaffected as the number of replication servers increases. Note that distributing client updates consumes progressively more primary server bandwidth as we increase the number of replication servers. As a *gedanken* experiment, we might imagine the practical limits to scalability as the number of replication servers grows. For the near

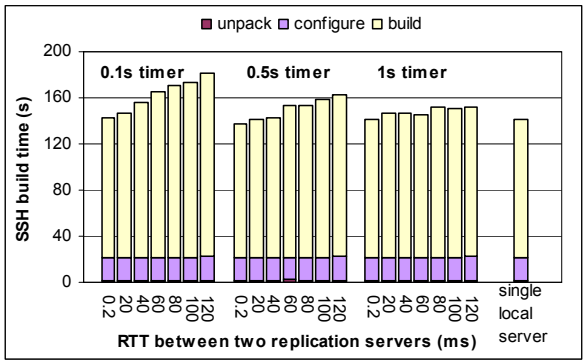


Figure 6. Deep control timeout values. The diagram shows the time to build SSH using hierarchical replication when the timeout for releasing a deep control is set to 0.1, 0.5, and 1 second.

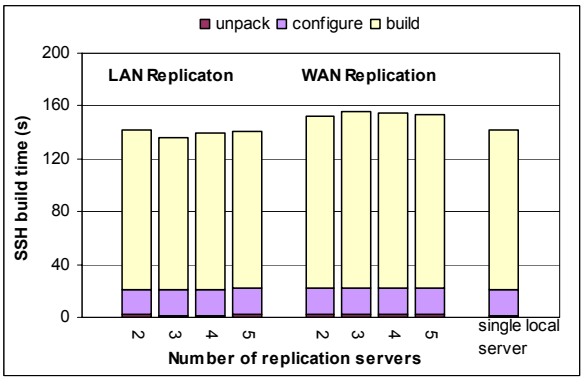


Figure 7. Increasing the number of replication servers. For LAN replication, the RTT between any two machines is around 200 μsec. For WAN replication, the RTT between any two replication servers is set to 120 msec, while the RTT between the client and the connected server is kept as 200 μsec. The primary server relinquishes deep control if it receives no further client updates for one second.

term, then, the cost of bandwidth appears to be a barrier to massive replication with our design.

5. Related Work

Replicated File Systems. Echo [13] and Harp [14] are file systems that use the primary copy scheme to support mutable replication. Both of these systems use a predetermined primary server for a collection of disks, a potential bottleneck if those disks contain hot spots or if the primary server is distant. Our system avoids this problem by allowing any server to be primary for any file, determined dynamically in response to client behavior.

Many replicated file systems trade consistency for availability. Examples include Coda [15], Ficus [16], and Locus [17]. These systems allow continued operations in the presence of failures, at the cost of sacrificing consistency if conflicting updates occur. Typically, automatic tools are provided to reconcile conflicts [18, 19]. However, in some cases, user involvement is needed to get the desired version of data.

Recent years have seen a lot of work in peer-to-peer file systems, including OceanStore [20], Ivy [21], Pangaea [22], and Farsite [23]. These systems address the design of systems in untrusted, highly dynamic environments. Consequently, reliability and continuous data availability are usually critical goals in these systems; performance or data consistency are often secondary considerations. Compared to these systems, our system addresses data replication among file system servers, which are more reliable but have more stringent requirements on average I/O performance.

Hierarchical Replication Control. The use of multiple granularities of control to balance performance and concurrency has been studied in other distributed file systems and database systems. Many modern transactional systems use hierarchical locking [24] to improve concurrency and performance of simultaneous transactions. In distributed file systems, Frangipani [25] uses distributed locking to control concurrent accesses among multiple shared-disk servers. For efficiency, it partitions locks into distinct lock groups and assign them to servers by group, not individually. Lin et al. study the selection of lease granularity when distributed file systems use leases to provide strong cache consistency [26]. To amortize leasing overhead across multiple objects in a volume, they propose volume leases that combine short-term leases on a group of files (volumes) with long-term leases on individual files. Farsite [23] uses content leases to govern which client machines currently have control of a file's content. A content lease may cover a single file or an entire directory of files.

Data Grid. Various middleware systems have been developed to facilitate data access on the Grid. Storage Resource Broker (SRB) [27] provides a metadata catalog service to allow location-transparent access for heterogeneous data sets. NeST [28], a user-level local storage system whose goal is to bring appliance technology to the Grid, provides best-effort storage space guarantees, mechanisms for resource and data discovery, user authentication, quality of service, and multiple transport protocol support. The Chimera system [29] provides a virtual data catalog that can be used by applications to describe a set of programs, and then to track all the data files produced by their execution. The work is motivated by observing that scientific data is often derived from other data by the application of computational procedures, which implies the need for a flexible data sharing and access system.

A commonly omitted feature among these middleware approaches is fine-grained data sharing semantics. Fur-

thermore, most of these systems provide extended features by defining their own API, so an application has to be re-linked with their libraries in order to use them.

6. Conclusion

Conventional wisdom holds that supporting consistent mutable replication in large-scale distributed storage systems is too expensive even to consider. Our study proves otherwise: in fact, it is both feasible and practical. In this paper, we present a replication control protocol that supports mutable replication with strong consistency guarantees. The protocol can be realized with a modest extension isolated to NFSv4 servers. We are working to influence the IETF to adopt such an extension. With the ubiquitous deployment of NFSv4, our work holds great promise for accessing and sharing data in Grid computing, delivering superior performance while rigorously adherence to conventional file system semantics.

References

[1] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," *J Network and Computer Apps.* (2001).

[2] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann (1998).

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The Globus Striped GridFTP Framework and Server," in *Proc. Supercomputing* (Nov. 2005).

[4] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS Version 4 Protocol," in *Proc. 2nd SANE* (2000).

[5] Sun Microsystems, Inc., "NFS Version 4 Protocol," RFC 3010 (2000).

[6] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3 Design and Implementation," in *Proc. USENIX Technical Conf.* (1994).

[7] J. Zhang and P. Honeyman, "NFSv4 Replication for Grid Storage Middleware," in *Proc 4th Intl. Workshop on Middleware for Grid Computing* (2006).

[8] J. Zhang and P. Honeyman, "Naming, Migration, and Replication for NFSv4," in *Proc. 5th SANE* (2006).

[9] L. Lamport, "Lower bounds on Asynchronous Consensus," In Andre Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of Lecture Notes in Computer Science, Springer (2003).

[10] J. Zhang and P. Honeyman, "Hierarchical Replication Control in a Global File System," Tech Report 06-08, Center for Information Technology Integration (2006).

[11] T. Ylonen, "SSH—Secure Login Connection Over the Internet," in *Proc. 6th USENIX Security Symp.* (1996).

[12] S. Hemminger, "Netem—Emulating Real Networks in the lab," linux.conf.au (*LCA)* (2005).

[13] A. Hisgen, A. Birrel, T. Mann, M. Schroeder, and G. Swart, "Granularity and Semantic Level of Replication in the Echo Distributed File System." in *Proc. Workshop on Mgmt. of Replicated Data* (Nov. 1990).

[14] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp File System," in *Proc. 13th ACM SOSP* (Oct. 1991).

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

[16] G.J. Popek, R.G. Guy, T.W. Page, Jr., and J.S. Heidemann, "Replication in Ficus distributed file systems," in *Proc. Workshop on Mgment. of Replicated Data* (1990).

[17] G.J. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A network transparent, high reliability distributed system," in *Proc. 8th SOSP* (1981).

[18] P. Kumar and M. Satyanarayanan, "Log-based directory resolution in the coda file system," in *Proc. 2nd Intl. Conf. on Parallel and Distributed Information Systems* (1993).

[19] P. Kumar and M. Satyanarayanan, "Supporting application-specific resolution in an optimistically replicated file system," in *Proc. Workshop on Workstation Operating System* (1993).

[20] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: the OceanStore Prototype," in *Proc. 2nd USENIX FAST* (2003).

[21] A. Muthitacharoen, R. Morris, T.M. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-peer File System," in *Proc. 5th SOSP* (2002).

[22] Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam, "Taming aggressive replication in the Pangaea wide-area file system," in *Proc. 5th SOSP* (2002).

[23] A. Adya, W.J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, R.P. Wattenhofer, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", in *Proc. 5th SOSP* (2002).

[24] J. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," *IFIP Working Conf. on Modeling in Data Base Management Systems* (1976).

[25] C.A. Thekkath, T. Mann, and E.K. Lee, "Frangipani: A Scalable Distributed File System," in *Proc. 16th SOSP* (1997).

[26] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. "Volume Leases for Consistency in Large-Scale Systems," *IEEE Trans. on Knowledge and Data Engineering* (1999).

[27] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," in *CASCON'98* (1998).

[28] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and M. Livny, "Flexibility, Manageability, and Performance in a Grid Storage Appliance," in *Proc. 11th HPDC* (2002).

[29] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao, "Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation," in *Proc. 14th Scientific and Statistical Database Management Conf.* (2002).