CITI Technical Report 06-07

Hierarchical Replication Control in a Global File System

Jiaying Zhang
jiayingz@eecs.umich.edu

Peter Honeyman
honey@citi.umich.edu

ABSTRACT

To meet the rigorous demands of large-scale data sharing in global collaborations, we present a replication scheme for NFSv4 that supports mutable replication without sacrificing strong consistency guarantees. Experimental evaluation indicates a substantial performance advantage over a single server system. With the introduction of a hierarchical replication control protocol, the overhead of replication is negligible even when applications mostly write and replication servers are widely distributed. Evaluation with the NAS Grid Benchmarks demonstrates that our system provides comparable — and often better — performance than GridFTP, the de facto standard for Grid data sharing.

Sep 06, 2006

Center for Information Technology Integration
University of Michigan
535 W. William St., Suite 3100
Ann Arbor, MI 48103-4978

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

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 significantly 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 [22]. FTP has the advantage of following a strict and simple standard and widespread vendor support. However, FTP has some fundamental inadequacies. Distribution is clumsy and inefficient: 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, this paper introduces 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 [3, 4] — 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 supports the so-called "close-to-open" consistency guarantee, 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 [5], can also serve as an enabling feature for re-using existing software in Grid computing.

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 [6]. To overcome this, we developed a replication protocol for NFSv4 that allows placement of replication servers near the compute nodes [7]. The protocol supports exactly the same semantics that NFSv4 provides 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 upon client updates. With no writers, our system has the performance profile of systems that support read-only replication. But unlike read-only systems, we also support concurrent write access without compromising NFSv4 consistency guarantees. Furthermore, the system can automatically recover from minority server failures, offering higher availability than single server systems.

Although that replication protocol breaks new ground in performance and availability for read-dominant applications, further analysis exposes a considerable performance penalty for large synchronous writes, bursty directory updates, and widely separated replication servers, characteristic data access patterns 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 or 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 scientific 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 the 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 propose 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.

The remainder of the paper is organized as follows. Section 2 reviews 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. 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 prototype implementation and several realistic benchmarks. In Sections 5 and 6, we discuss related work, summarize, and conclude.

2. Fine-grained Replication Control

In this section, we review the design of a mutable replication protocol for NFSv4 that guarantees close-to-open consistency semantics by electing a primary server upon client updates at the granularity of a single file or directory [6]. Section 2.1 introduces the replication protocol, Section 2.2 presents the primary server election algorithm, and Section 2.3 describes the handling of various kinds of failures.

2.1 Replication Control Protocol

Most applications, scientific and otherwise, are dominated by reads, so it is important that a replication control protocol avoids overhead for read requests. We achieve this in our system by using a variant of the well understood and intuitive *primary-copy scheme* to coordinate concurrent writes. Under the conventional primary copy approach, a primary server is statically assigned for each mount point during configuration so all write requests under a single mount point go to the same primary server. On the contrast, in our system, the server to which a client sends the first write request is elected as the primary server for the file or the directory to be modified. With no writers, our system has the natural performance advantages of systems like AFS that support read-only replication: use a nearby server, support transparent client rollover on server failure, etc. However, we also support concurrent write access without weakening NFSv4 consistency guarantees.

The system works as follows. When a client opens a file for writing, it sends the open request to the NFS server that it has selected for the mount point to which the file belongs. An application can open a file in write mode without actually writing any data for a long time, e.g., forever, so the server does nothing special until the client makes its first write request. When the first write request arrives, the server invokes the replication control protocol, a server-to-server protocol extension to the NFSv4 standard.

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, a server must become the primary server for all objects. The common case for this is rename, which needs to make two updates atomically. 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. 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. 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 inherent 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 an 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 [7]. With this requirement, as long as a majority of the replication servers are 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 its last pending request only.

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 are characterized by many synchronous writes or directory updates and replication servers that are far away from each other [7]. 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 primary 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.

2.2 Primary Server Election

Two (or more) servers may contend to become the primary server for the same object (file or directory) concurrently. To guarantee correctness of our replication control protocol, we need to ensure that more than one primary server is never chosen for a given object, even in the face of conflicts and/or failures. This problem is a special case of the extensively studied *consensus problem*.

In the consensus problem, all correct processes must reach an agreement on a single proposed value [13]. Many problems that arise in practice, such as electing a leader or agreeing on the value of a replicated object, are instances of the consensus problem. In our case, if we assign each replication server a unique identifier, the primary server election problem is easily seen to be an instance of the consensus problem: electing a primary server is equivalent to agreeing on a primary server identifier.

Achieving consensus is a challenging problem, especially in an asynchronous distributed system. In such a system, there is no upper bound on the message transmission delays or the time to execute a computing step. A good consensus algorithm needs to maintain *consistency*, i.e., only a single value is chosen, and to guarantee *progress* so that the system is eventually synchronous for a long enough interval [14]. Unfortunately, Fischer et al. showed that the consensus problem cannot be solved in an asynchronous distributed system in the presence of even a single fault [15].

Observing that failures are rare in practice, candidate consensus algorithms have been proposed to separate the consistency requirement from the progress property [16-20]. That is, while consistency must be guaranteed at all times, progress may be hampered during periods of instability, as long as it is eventually guaranteed after the system returns to the normal state. Our system also follows this design principle. Rather than using an existing consensus protocol such as Paxos [16], we develop a primary server election algorithm of our own based on the following considerations.

Most of the proposed consensus algorithms attempt to minimize the amount of time between the proposal of a value and the knowledge of a chosen value by all members. In our system, a replication server initiates the primary server election procedure upon receiving an update request from a client. The server cannot process the client's request until it determines the primary server for the object to be modified. Our aim is therefore to minimize the elapsed time between these two events. Consequently, we use the primary server election algorithm sketched in Figure 1 in our replication control protocol.

It is easy to verify that the algorithm satisfies the consistency requirement: a primary server needs to accumulate the acknowledgments from a majority of the replication servers and a replication server cannot commit to more than one primary server, so only a single primary server is elected for a given object.

```
Upon receiving a client update request, initiate primary
server election if the object's primary server is NULL
set the object's primary server to MyID  // ack self
loop until all active servers ack
    propose <MyID, object> to unacked servers
    wait until all those servers reply or timeout
    if the number of acks received is less than majority then
        identify competitors from the replies
        if any competitor is accepted by a majority of servers, or
        any competitor's identifier is larger than MyID then
            set the object's primary server to NULL
            send abort <MyID, object> to all acked servers
            exit loop
    else mark timed out servers inactive

Upon receiving propose <ServerID, object>
if the object's primary server is NULL then
    set the object's primary server to ServerID
    send ack
else
    send nack <the object's primary server>

Upon receiving abort <ServerID, object>
if the object's primary server equals to ServerID then
    set the object's primary server to NULL
```

Figure 1. Primary Server Election. This pseudocode sketches the election protocol. Section 2.3 discusses failure handling in more detail.

Furthermore, for the common case — no failures and only one server issues the proposal request — primary server election completes with only one message delay between the elected primary server and the farthest replication server. In fact, since the server can process the client's update request as soon as it receives acknowledgments from a majority of the replication servers, the conflict- and failure- free response time is bounded by the largest round-trip time (RTT) separating the primary server and half of the nearest replication servers. We note for emphasis that this improves on many existing consensus algorithms that require two message delays to decide on a chosen value [18].

If multiple servers compete to be the primary server for an object, it is possible that none of them collects acknowledgments from a majority of the replication servers in the first round of the election. Absent failure, the conflict is quickly learned by each competing server from the replies it receives from other replication servers. In this case, the server with the largest identifier is allowed to proceed and its competitors abort their proposals by releasing the servers that have already acknowledged.

In the presented algorithm, the winner of the competition keeps sending proposal requests to replication servers that have not acknowledged its role, subject to timeout. However, the abort request from a yielding competitor may arrive at such a replication server after several rounds of proposal distribution, resulting in redundant network messages. The situation can be improved with a small optimization in the second round of the election: the winning server can append the replies it has collected in previous rounds to its subsequent proposals. With this information, a server that receives a late-round proposal can learn that the server it is currently treating as primary will soon abort the election. Thus, it can briefly delay replying to the new proposal, increasing the chance that the object is released by the old primary server before responding to the late-round proposal. We leave the detailed discussion of failures to the next subsection, but point out that when the system is free of failure, primary server election converges in two message delays even in the face of contention.

2.3 Coping with Failure

The discussion so far focuses on replication control in normal — i.e., failure-free — system states. However, failure introduces complexity. Different forms of failure may occur: client failure, replication server failure, network partition, or any combination of these. In this subsection, we describe the handling of each case. Our failure model is *fail stop* [25], i.e., no Byzantine failures [21].[1]

Following the specification of NFSv4, a file opened for writing is associated with a lease on the primary server, subject to renewal by the client. If the client fails, the server receives no further renewal requests, so the lease expires. Once the primary server decides that the client has failed, it closes any files left open by the failed client on its behalf. If the client was the only writer for a file, the primary server relinquishes its role for the file.

To guarantee consistency upon server failure, our system maintains an *active view* among replication servers [47]. During file or directory modification, a primary server removes from its active view any replication server that fails to respond to its election request or update requests within a specified time bound. The primary server distributes its new view to other replication servers whenever the active view changes. We require an active view to contain a majority of the replication servers. The primary server replies to a client close operation only after a majority of the replication servers have acknowledged the new active view. Each replication server records the active view in stable storage. A server not in the active view may have stale data, so the working servers must deny any requests coming from a server not in the active view. We note that if the server failure is caused by network partition, close-to-open semantics is not guaranteed on the

[1] Security of the protocol follows from the use of secure RPC channels, mandatory in NFSv4, for server-to-server communication

"failed" server(s), i.e., clients may have read stale data without awareness. However, a server excluded from the active view cannot update any working server, which prevents the system from entering an inconsistent state.[2]

If a replication server fails after sending primary server election requests to a minority of replication servers, the failure can be detected by a subsequently elected primary server. As described above, that primary server eliminates the failed server from the active view and distributes the new view to the other replication servers. The servers that have acknowledged the failed server switch to the new primary server after employing the new active view. The consistency of the data is unaffected: the failed server had not received acknowledgements from a majority of the replication servers so it cannot have distributed any updates.

A primary server may fail during file or directory modification. With the relaxed update distribution requirement, the primary server responds to a client update request immediately before distributing updates to the other replication servers. As a result, other active servers cannot recover the most recent copy among themselves. The "principle of least surprise" argues the importance of guaranteed durability of data written by a client and acknowledged by the server, so we make the object being modified inaccessible until the failed primary server recovers or an outside administrator re-configures the system. However, clients can continue to access objects that are outside the control of the failed server, and applications can choose whether to wait for the failed server to recover or to reproduce the computation results.

Since our system does not allow a file or a directory to be modified simultaneously on more than one server even in case of failure, the only valid data copy for a given file or directory is the most recent copy found among the replication servers. This feature simplifies the failure recovery in our system: when an active server detects the return of a failed server, either upon receiving an election or update request from the returning server or under the control of an external administration service, it notifies the returning server to initiate a synchronization procedure. During synchronization, write operations are suspended, and the returning server exchanges the most recent data copies with all active replication servers.[3] After recovery, all the objects that were controlled by the returning server, i.e., those for which it was the primary server at the moment it failed, are released and the server is added to the active view.

Should a majority of the replication servers fail simultaneously, an external administrator must enforce a grace period after the recovering from the failure. To be safe, the administration service should instruct each replication server to execute the synchronization procedure during the grace period.

3. Hierarchical Replication Control

Notwithstanding an efficient consensus protocol, a server can still 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 have developed 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 hold on an object. 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 con-*

[2] Generally, the computation results on a failed server are dubious since they might be generated with stale input data. To be safe, applications should re-compute these results.

[3] This process can be done easily by alternately executing a synchronization program, such as *rsync*, between the returning server and each active replication server, with the option to skip any file or directory whose modification time is newer than the source node.

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 if **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 if **D** is not controlled by a server other than **P**, and no descendant of **D** is controlled by a server other than **P**

Figure 2. Using and granting controls.

*trol 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 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 2. We consider single writer cases to be 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 prevent an object from being 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 scans each directory along the path from the referred object to the mount point. If an ancestor of the object has a primary server other than the one who 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 decedents whose primary servers are not empty. Figure 3 shows entries in the ancestry table and an example that 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.

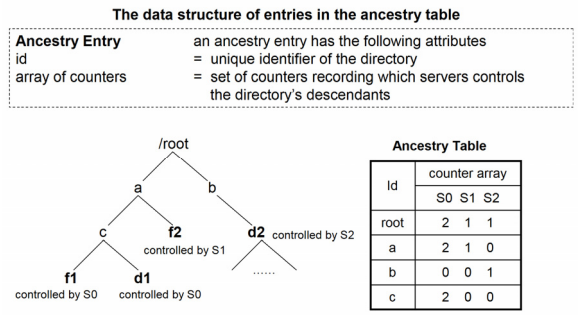
A replication server needs only one lookup in its ancestry table to tell whether a directory subtree holds an object under the control of a different server: It first finds the mapping entry of the directory from its ancestry table, and then examines that entry's counter array. If the counter on any 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 it rejects the deep control request.

3.2 Primary Server Election with Deep Control

With the introduction of deep control, two 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 4 illustrates. To guarantee progress during conflicts, we extend the primary server election algorithm described in Section 2.2 as follows. When a replication server receives a shallow control request for an object **L** from a peer server **P** but cannot grant the control according to the rules listed in Figure 2, it replies to **P** with the identifier of the primary server that currently controls **L**. On the other hand, if a replication server judges that it cannot grant a deep control request, it simply replies 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. Then with shallow controls only, the progress of primary server election follows the discussion in Section 2.2.

3.3 Performance and Concurrency Tradeoff

The introduction of deep control introduces a performance and concurrency trade-off. A primary server



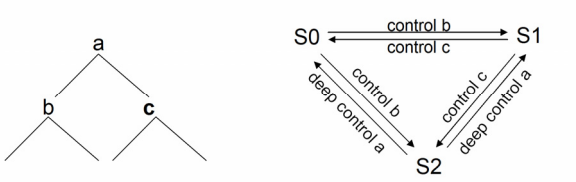
Consider three replication servers: S0, S1, and S2. S0 is currently the primary server of file f1 and directory d1. S1 is currently the primary server of file f2. S2 is currently the primary server of directory d2. The right table shows the content of the ancestry table maintained on each replication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

Figure 3. Structure and maintenance of the ancestry table.

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 postulate 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 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. We 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.



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 listed in Figure 2, 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 4. Potential conflicts in primary server election caused by deep control.

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 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 heuristic 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.

The introduction of deep control provides significant performance benefits, but can 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 infrequent, we anticipate that the performance gains of deep control far outweigh the

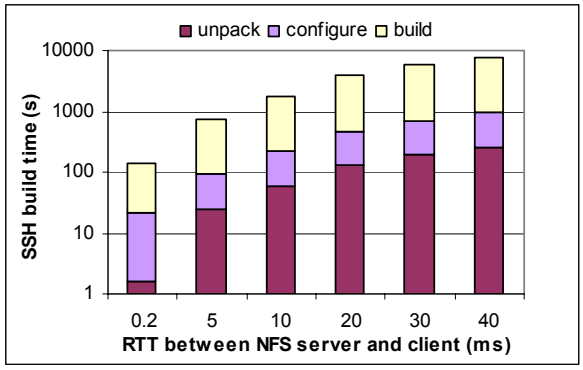


Figure 5. SSH build on a single NFSv4 server.

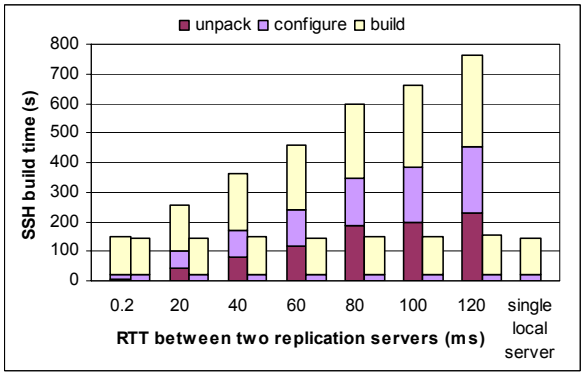


Figure 6. Fine-grained replication control vs. hierarchical replication control. The first column shows the time to build SSH using fine-grained replication control. The second column shows the time when using hierarchical replication control. For runs with hierarchical replication control, the primary server relinquishes deep control if it receives no client updates for one second.

potential cost of servers failing while holding deep control on directories.

4. Evaluation

In this section, we evaluate the performance of hierarchical replication control with a series of experiments over simulated wide area networks. We start with a coarse evaluation in Section 4.1 using the SSH-Build benchmark, and find that hierarchical replication control is very successful in reducing overhead, even when the time that deep control is held is short. In Section 4.2, we explore system performance with the NAS Grid Benchmarks in simulated wide area networks and find that our replicated file system holds a substantial performance advantage over a single server system. At the same time, it provides comparable and often better performance than GridFTP, the conventional approach to moving data sets in the Grid.

We conducted all the experiments presented in this paper 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. We use Netem [23] to simulate 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, and enable the `async` option (asynchronously write data to disk) on the NFS servers. Although not comprehensive, we expect that our settings closely resemble a typical Grid environment — high performance computing clusters connected by long fat pipes.

All measurements presented in this paper are mean values from five trials of each experiment; measured variations in each experiment are negligible. Each experiment is measured with a warm client cache, but the temperature of the client cache has little effect on the presented results.

4.1 Evaluation with SSH-Build Benchmark

The SSH-Build benchmark [24] 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.

Before diving into the evaluation of hierarchical replication, we look at performance when accessing a single distant NFSv4 server. Figure 5 shows the measured times 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. Figure 5 shows (in log-scale) 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 demonstrates that it is impractical to execute update-intensive applications using a stock remote NFS 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.

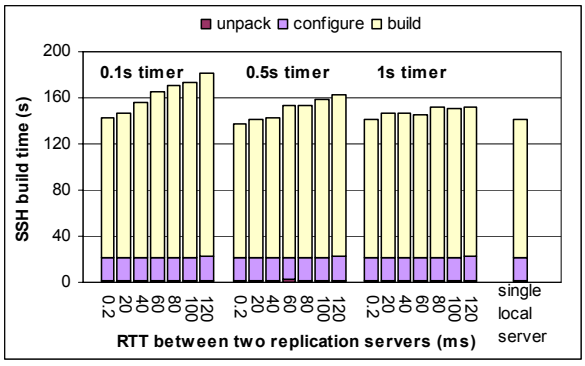


Figure 7. Deep control timeout values. The diagram shows the time to build SSH using hierarchical replication control when the the timeout for releasing a deep control is set to 0.1 second, 0.5 second, and 1 second.

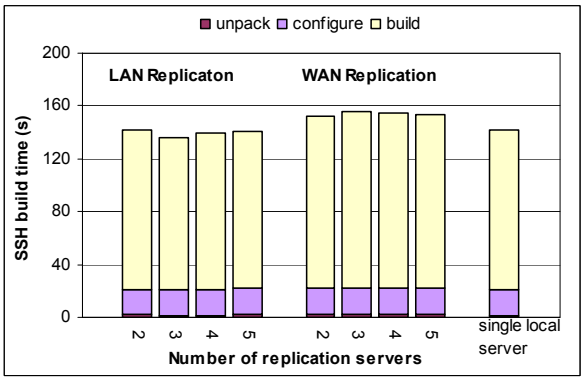


Figure 8. Increasing the number of replication servers. For LAN replication, the RTT between any two machines is around 200 μsec. For WAN replication, the RTT between any two replication servers is set to 120 msec, while the RTT between the client and the connected server is kept as 200 μsec. The primary server relinquishes deep control if it receives no further client updates for one second.

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 6, 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 7 shows the results.

Figure 7 shows that 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 —- can capture 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 local area network and in a simulated wide area network. For local area replication, the measured RTT between any two machines is around 200 μsec. For wide area 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 at 200 μsec.

Figure 8 shows that performance is largely unaffected as the number of replication servers increases. However, 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. A primary server takes on an output bandwidth obligation that multiplies its input bandwidth by the number of replication servers. For the near term, then, the cost of bandwidth appears to be a barrier to massive replication with our design.

4.2 Evaluation with Grid Benchmarks

The NAS Grid Benchmarks (NGB), released by NASA, provide an evaluation tool for Grid computing [26]. The benchmark suite evolves from the NAS Parallel Benchmarks (NPB), a toolkit designed and widely used for benchmarking on high-performance computing [27]. An instance of NGB comprises a collection

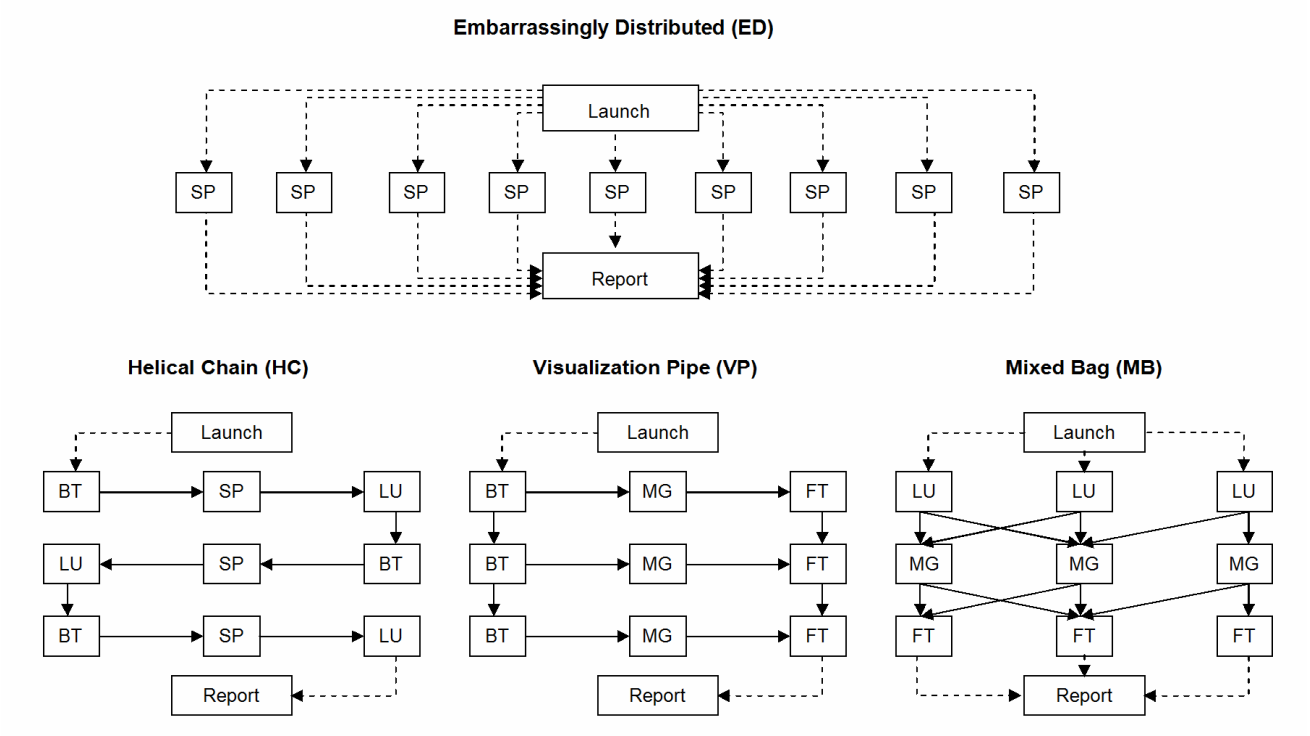


Figure 9. Data flow graphs of the NAS Grid Benchmarks.

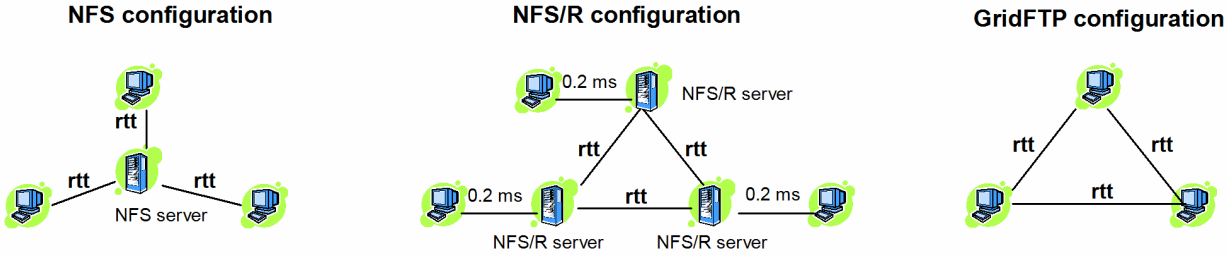


Figure 10. NGB evaluation experiment setup.

Table 1. Amount of data exchanged between NGB tasks

Class	Helical Chain			Visualization Pipe				Mixed Bag	
	BT→SP	SP→LU	LU→BT	BT→MG	MG→FT	BT→BT	FT→FT	BT→MG	MG→FT
S	169K	169K	169K	34K	641k	169K	5.1M	34K	641K
W	1.4M	4.5M	3.5M	271K	41M	1.4M	11M	702K	41M
A	26M	26M	26M	5.1M	321M	26M	161M	5.1M	321M

of slightly modified NPB problems, each of which is specified by class (mesh size, number of iterations), source(s) of input data, and consumer(s) of solution values. The current NGB consists of four problems: Embarrassingly Distributed (ED), Helical Chain (HC), Visualization Pipe (VP), and Mixed Bag (MB).

ED, HC, VP, and MB highlight different aspects of a computational Grid. ED represents the important class of Grid applications called parameter studies, which constitute multiple independent runs of the same program, but with different input parameters. It requires virtually no communication, and all the tasks in it execute independently. HC represents long chains of repeating processes; tasks in HC execute sequentially. VP simulates logically pipelined processes, like those encountered when visualizing flow solutions as the simulation progresses. The three tasks included in VP fulfill the role of flow solver, post processor, and visu-

alization, respectively. MB is similar to VP, but introduces asymmetry. Different amounts of data are transferred between different tasks, and some tasks require more work than others do.

Figure 9 illustrates the Data Flow Graph for each of these benchmarks. The nodes in the graph, indicated by the rectangular boxes, represent computational tasks. Dashed arrows indicate control flow between the tasks. Solid arrows indicate data as well as control flow. Launch and Report do little work; the former initiates execution of tasks while the latter collects and verifies computation results.

The NGB instances are run for different problem sizes (denoted *Classes*). For the evaluation results presented in this paper, we use the three smallest Classes: S, W, and A. Table 1 summarizes the amount of data communicated among tasks for these Classes.

A fundamental goal of Grid computing is to harness globally distributed resources for solving large-scale computation problems. To explore the practicality and benefit of using NFS replication to facilitate Grid computing, we compare the performance of running NGB under three configurations, referred as NFS, NFS/R, and GridFTP.

In the experiments, we use three computing nodes to emulate three computing clusters, with the RTT between each pair increased from 200 μsec to 120 msec. In the NFS configuration, the three computing nodes all connect to a single NFS server. In the NFS/R configuration, we replace the single NFS server with three replicated NFS servers, with each computing node connected to a nearby server. In the GridFTP configuration, we use GridFTP to transfer data among computing nodes. The GridFTP software we use is `globus-url-copy` from Globus-4.0.2 toolkit. In our experiments, we start eight parallel data connections in each GridFTP transfer, which we found provides the best-measured performance for GridFTP. (The NFS/R prototype also supports parallel data connections between replicated NFS servers. But in the experiments presented here, the performance improvement using multiple data connections is small, so we report results measured with a single server-to-server data connection only.) Figure 10 illustrates the experiment setup.

For the GridFTP configuration, we run the NGB tasks using the Korn shell Globus implementation from the NGB3.1 package. In this implementation, a Korn shell script launches the NGB tasks in round robin on the specified computing nodes. Tasks are started through the `globusrun` command with the batch flag set. After a task completes, output data is transferred to the computing node(s), where the tasks require the data as input. A semaphore file is used to signal task completion: computing nodes poll their local file systems for the existence of the semaphore files to monitor the status of the required input files. After all tasks start, the launch script periodically queries their completion using `globus-job-status` command.

For the NFS and NFS/R setups, we extended the original NGB Korn shell scripts. The modified programs use `ssh` to start NGB tasks in round robin on the specified computing nodes. The computing nodes and the launch script poll for the status of the required input data and tasks with semaphore files, as above.

Figure 11 shows the results of executing NGB on NFS, NFS/R, and GridFTP as the RTT among the three computing nodes increases from 200 μsec to 120 msec. The data presented is the "measured turnaround" time, i.e., the time between starting a job and obtaining the result. With GridFTP, turnaround time does not include deployment and cleanup of executables on Grid machines. The time taken in these two stages ranges from 10 seconds to 40 seconds, as the RTT increases from 200 μsec to 120 msec.

Evidently, in Grid computing, deployment and cleanup can sometimes take significant time with large size of executables and input data [28]. Furthermore, in some cases, it is hard for users to determine which files to stage [29]. With NFS and NFS/R, on the other hand, there is no extra deployment and cleanup time, because computing nodes access data directly from file servers. Even so, the times we report do not reflect this inherent advantage.

The histograms in Figure 11 show that performance with a single NFS server suffers dramatically as the RTT between the server and the computing nodes increases. Except for the ED problem — whose tasks run independently — on larger data sets, the experiments take a very long time to execute when the RTT increases to 120 msec. In fact, the times are even longer than the times measured when running the problems on a single computing node without parallel computing. (Table 2 shows NGB execution times on a single computing node with a local ext3 file system.) Clearly, in most cases it is impractical to run applications on widely distributed clients connected to a single NFS server, even for CPU intensive applications.

On the other hand, with NFS/R and GridFTP on large class sizes, run times are not adversely affected by increasing RTT. When the class size is small (e.g., the results of Class S), NFS/R outperforms GridFTP, because the latter requires extra time to deploy dynamically created scripts and has extra Globus-layer overhead. The NGB experiments demonstrate that well-engineered replication control provides superior file system semantics and easy programmability to WAN-based Grid applications without sacrificing performance.

Table 2. Times of executing NGB on a single computing node with a local ext3 file system

Class	S				W				A			
Benchmark	ED	HC	VP	MB	ED	HC	VP	MB	ED	HC	VP	MB
Time (s)	2	1	9	6	217	31	83	101	1380	223	930	870

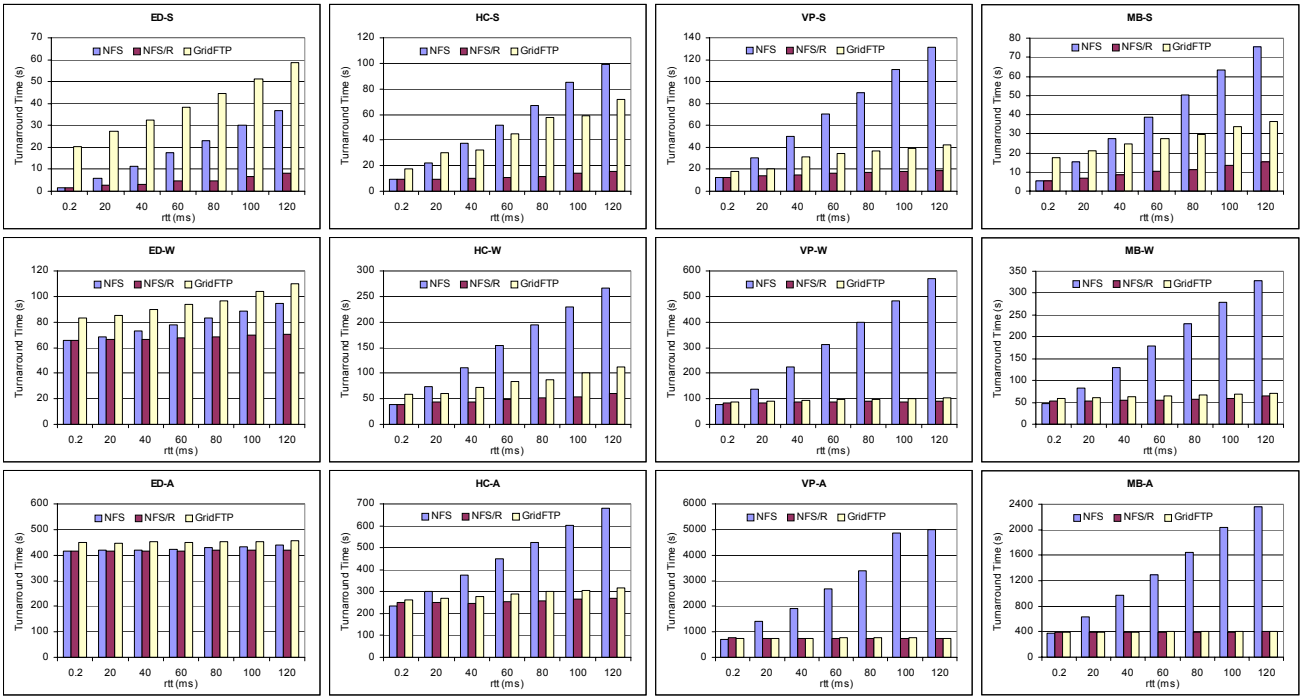


Figure 11. Turnaround times (seconds) of NGB on NFS, NFS/R, and GridFTP.

5. Related Work

Replicated File Systems. Echo [34] and Harp [35] are file systems that use the primary copy scheme to support mutable replication. Both of these systems use a pre-determined primary server for a collection of disks, a potential bottleneck if those disks contain hot spots or if the primary server is distant. In contrast, 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 [12], Ficus [30], and Locus [31]. 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 [32, 33]. 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 [36], Ivy [37], Pangaea [38], and Farsite [39]. 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.

The importance of maintaining strong consistency with mutable replication is underscored by recent work on storage systems, called Chain replication [46]. The system intends to support high throughput and availability without sacrificing strong consistency guarantees. It does this by disseminating updates to a chain of replication servers serially, which provides high throughput but results in slow response time for each update request.

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 [40] to improve concurrency and performance of simultaneous transactions. In distributed file systems, Frangipani [41] 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 [42]. 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 [39] 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) [43] provides a metadata catalog service to allow location-transparent access for heterogeneous data sets. NeST [44], 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 [45] 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. Furthermore, 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, practical, and can be realized today. This replicated file system presented in this paper supports mutable replication with strong consistency guarantees. Experimental evaluation shows that the system 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 Applications* (2001).

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

[3] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS Version 4 Protocol," *2nd Intl. Conf. on System Administration and Network Engineering*, Maastricht (2000).

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

[5] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. "NFS Version 3 Design and Implementation," In *Proceedings of the USENIX Summer 1994 Technical Conference* (1994).

[6] J. Zhang and P. Honeyman, "Naming, Migration, and Replication for NFSv4," *5th Intl. Conf on System Administration and Network Engineering*, Delft (2006).

[7] J. Zhang and P. Honeyman, "Reliable Replication at Low Cost," Technical Report 06-01, Center for Information Technology Integration (2006).

[8] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," In *Proceedings of the 18th IEEE Mass Storage Conference* (2001).

[9] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, "The ITC distributed file system: principles and design," *SIGOPS Oper. Syst. Rev.*, 19:5 (1985).

[10] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw, "Legionfs: a secure and scalable file system supporting crossdomain high-performance applications.," In *Proc. of the 2001 ACM/IEEE conference on Supercomputing* (2001).

[11] P. Kumar and M. Satyanarayanan, Supporting application specific resolution in an optimistically replicated file system. In *Workshop on Workstation Operating Systems* (1993).

[12] 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* 39:4 (1990).

[13] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," Technical report, Department of Computer Science, Yale University (1983).

[14] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, 35(2):288--323 (1988).

[15] M. J. Fischer, N. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, 32(2):374 (1985).

[16] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, 16(2):133–169 (1998).

[17] L. Lamport, "Fast Paxos," Technical Report MSR-TR-2005-112, Microsoft Research (2005).

[18] L. Lamport, "Lower bounds on asynchronous Consensus," In Andre Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Di-*

rections in Distributed Computing, volume 2584 of Lecture Notes in Computer Science, pages 22--23. Springer (2003).

[19] D. Malkhi, F. Oprea, and L. Zhou, "Ω meets Paxos: Leader election and stability without eventual timely links," In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, pages 199--213 (2005).

[20] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, 43(2):225-267 (1996).

[21] F. Cristian, H. Aghali, R. Strong and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," in *Proc. 15th FTCS* (June 1985).

[22] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The Globus Striped GridFTP Framework and Server," In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (Nov. 2005).

[23] S. Hemminger, "Netem – Emulating Real Networks in the lab," *LCA2005* (Apr. 2005).

[24] T. Ylonen, "SSH - Secure Login Connection Over the Internet," *6th USENIX Security Symp.* (1996).

[25] Schneider, F. B., "Byzantine generals in action: Implementing fail-stop processors," *ACM Transactions on Computer Systems* 2(2), 145-154 (1984).

[26] M. Frumkin and R. F. V. der Wijngaart, "NAS Grid Benchmarks: A tool for grid space exploration," *Cluster Computing*, 5(3):247--255 (2002).

[27] D.H. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," NAS Technical Report RNR-9 1-002, NASA Ames Research Center, Moffett Field, CA (1991).

[28] H. Holtman, "CMS data grid system overview and requirements," The Compact Muon Solenoid (CMS) Experiment Note 2001/037, CERN, Switzerland (2001).

[29] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Pipeline and batch sharing in grid workloads," In *Proceedings of the 12th IEEE Symposium on High Performance Distributed Computing* (2003).

[30] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann, "Replication in Ficus distributed file systems," In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 24–29 (1990).

[31] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A network transparent, high reliability distributed system," in *Proceedings of the Eigth Symposium on Operating Systems Principles*, pp. 169--177, (December 1981).

[32] P. Kumar and M. Satyanarayanan, "Log-based directory resolution in the coda file system," In *Proceedings of the second international conference on Parallel and distributed information systems*, pages 202–213 (1993).

[33] P. Kumar and M. Satyanarayanan, "Supporting application-specific resolution in an optimistically replicated file system," In *Workshop on Workstation Operating Systems*, pages 66–70 (1993).

[34] 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).

[35] 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).

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

[37] A. Muthitacharoen, R. Morris, T.M. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-peer File System," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[38] Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam, "Taming aggressive replication in the Pangaea wide-area file system," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[39] 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 *Proceedings of 5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[40] 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).

[41] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System," In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (1997).

[42] 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).

[43] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," In *Proceedings of CASCON'98 Conference* (Nov. 1998).

[44] 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 *Proceedings of the 11th IEEE international Symposium on High Performance Distributed Computing* (July 2002).

[45] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao, "Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation," In *Proceedings of the Scientific and Statistical Database Management Conference* (July 2002).

[46] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," *Symposium on Operating Systems Design and Implementation*, pages 91--104 (2004).

[47] A. E1 Abbadi, D. Skeen, and F. Cristian, "An Efficient Fault-tolerant Protocol for Replicated Data Management," in *Proc. 5th ACM SIGMOD*, (1985).