CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

A Replicated File System for Grid Computing

Jiaying Zhang[†] and Peter Honeyman[‡]

Center of Information Technology Integration, University of Michigan, Ann Arbor, MI, U.S.A

SUMMARY

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.

KEY WORDS: replication; distributed file system; Grid

1. INTRODUCTION

Driven by the needs of scientific collaborations, the emerging Grid infrastructure [17, 11] aims to connect globally distributed resources to form a shared virtual computing and storage system, offering a model for solving large-scale computation problems. Sharing in Grid computing is not merely file exchange but also entails direct access to computers, software, data, and other resources, as is required by a range of collaborative scientific problem-solving patterns. To make such sharing simple and effective demands data access schemes that are scalable, reliable, and efficient.

The primary data access method used today in the Grid infrastructure is GridFTP [2]. Engineered with Grid applications in mind, GridFTP has many advantages: automatic negotiation of TCP options to fill the pipe, parallel data transfer, integrated Grid security,

[*]Correspondence to: Peter Honeyman, 535 W. William St., Suite 3100, Ann Arbor, MI, 48105, U.S.A.
[†]E-mail: jiayingz@umich.edu
[‡]E-mail: honey@citi.umich.edu
Contract/grant sponsor: NSF Middleware Initiative Grant; contract/grant number: SCI-0438298
Contract/grant sponsor: Network Appliance, Inc.; contract/grant number:

Received 2007
Revised 2007

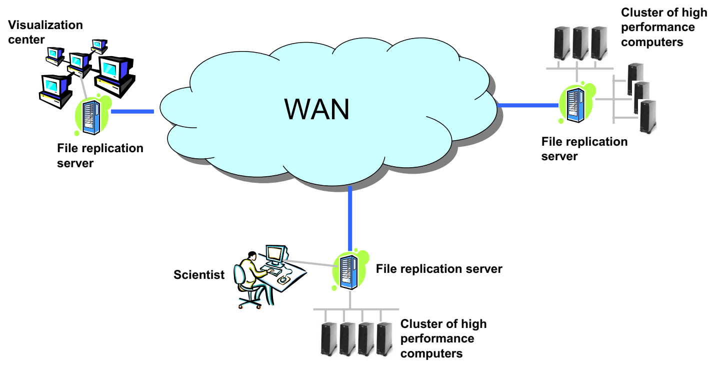

Figure 1. A Grid use case example.

and resumption of partial transfers. In addition, because it runs as an application, GridFTP is easy to install and support across a broad range of platforms.

On the other hand, GridFTP does not offer sophisticated distributed data sharing, which impedes the convenient use of globally distributed resources for scientific studies. For example, in a common Grid use-case, a scientist wants to run a simulation on high performance computing systems and analyze results on a visualization system. With the Grid technologies available today, the scientist submits the job to a Grid scheduler, such as Condor-G [19]. The Grid scheduler determines where to run the job, pre-stages the input data to the running machines, monitors the progress of the running job, and when the job is complete, transfers the output data to the visualization system through GridFTP. The output data is reconstructed in the visualization site, and then the results are returned to the scientist.

The scenario has the advantage of enabling the scientist access to more computing resources and speeding up his simulation. However, the whole process is performed in a batch mode. The scientist cannot view intermediate results before the entire scheduled job is complete. Scientific simulations are often iterative and interactive processes, so the need to wait for hours or days to examine experimental results is very inconvenient. To overcome this disadvantage, the Grid infrastructure requires more flexible data distribution and sharing in its middleware.

To facilitate Grid computing over wide area networks, we developed a replicated file system that provides users high performance data access with conventional file system semantics. The system supports a global name space and location independent naming, so applications on any client can access a file with a common name and without needing to know where the data physically resides. The system supports mutable (i.e., read/write) replication with explicit consistency guarantees, which lets users make data modification with ease, safety, and transparency. The system provides semantics compatible with POSIX, allowing easy deployment of unmodified scientific applications. We implemented our design by extending NFSv4, the emerging standard protocol for distributed filing [32]. In latter discussions, we refer to the replication system we implemented as **NFSv4.r**.

Using NFSv4.r, the scientist in the example described above can now monitor and control the progress of the simulation in real time. As illustrated in Figure 1, with the support of a global name space, the scientist can run programs on remote machines with the same pathname and without any reconfiguration. By using a replicated file system, the intermediate output of simulation is automatically distributed to the visualization center and the scientist's computer. The scientist can view intermediate results and immediately determine if parameters or algorithms need to be adjusted. If so, he can update them from his local computer and restart the simulation on the remote site, as simply as if he were running the experiment locally. Meanwhile, remote computation nodes can still access data from a nearby server.

In the rest of this paper, we detail the design, implementation, and evaluation of the replicated file system we developed. The naming scheme for supporting a global name space and location independent naming can be referred in a previous paper [43]. The remainder of the paper proceeds as follows. Section 2 presents a mutable replicated file system that coordinates concurrent writes by dynamically electing a primary server at various granularities. Section 3 examines the system performance over wide area networks. Following that, we review related work in Section 4 and conclude in Section 5.

2. CONSISTENT MUTABLE REPLICATION

To meet availability, performance, and scalability requirements, distributed services naturally turn to replication, and file service is no exception. While the concept of file system replication is not new, existing solutions either forsake read/write replication totally [6, 41, 36] or weaken consistency guarantees [37, 24, 33]. These compromises fail to satisfy the requirements for global scientific collaborations.

Returning to the example described in Section 1, experiment analysis is often an iterative, collaborative process. The stepwise refinement of analysis algorithms employs multiple clusters to reduce development time. Although the workload during this process is often dominated by read, it also demands the underlying system to support write operations. Furthermore, strong consistency guarantees are often assumed. For example, an executable binary may incorporate user code that is finished only seconds before the submission of the command that requires using the code. To guarantee correctness, the underlying system needs to ensure that the modified data is successfully transferred to the remote machine where the code is running.

The conventional NFS consistency model, the so-called "close-to-open semantics" [38], guarantees that an application opening a file sees the data written by the last application that writes and closes the file. This strategy has proved to provide sufficient consistency for most applications and users [31].

We also consider close-to-open semantics to be important in Grid data access, as the above example illustrates. To provide such a guarantee, 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 writes without compromising NFSv4 consistency guarantees.

The rest of this section presents the design of the mutable replication control mechanisms in detail. Section 2.1 describes a *fine-grained replication control protocol* that coordinates concurrent writes by electing a primary server at the granularity of a single file or directory. The protocol offers good performance for read-dominant applications, but introduces considerable overhead for bursty metadata updates. To reduce the performance overhead of replication control, we propose a *hierarchical replication control protocol* that extends the fine-grained replication control protocol by allowing a primary server to assert control at various granularities. Section 2.2 details the design of this extended protocol.

2.1. Fine-grained Replication Control

In this subsection, we describe 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. Section 2.1.1 introduces the replication protocol, Section 2.1.2 presents the primary server election algorithm, and Section 2.1.3 discusses the handling of various kinds of failures.

2.1.1. 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 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 writes 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 acknowledgments 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 acknowledgments from a majority of the replication servers [43]. 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 [43]. 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.

2.1.2. 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 protocol, we need to ensure that more than one primary server is never chosen for a given object, even in the face

```
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, or
            set the object's primary server to NULL
            send abort <MyID, object> to all acked servers
            wait until the object's primary not NULL or timeout
            exit loop
    else  // have collected acks from majority
        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 2. Primary server election pseudocode.

of conflicts and/or failures. The 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 [15]. 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 [16].

Observing that failures are rare in practice, candidate consensus algorithms have been proposed to separate the consistency requirement from the progress property [25, 27, 26, 29, 10]. 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. Following this principle, we implement a primary server election algorithm that achieves the lower time bound of Fast Consensus. The algorithm assumes that all messages are delivered in order, which can be achieved by including a serial number in each message or through a reliable transport protocol. Figure 2 presents the pseudo code of the algorithm.

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. 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.

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. A server that aborts its election then waits for the late-round proposal request from the identified winner server. The waiting is associated with a timer to guarantee progress in case that the winner server fails. Upon receiving the late-round proposal from the winner, the server acknowledges the request and forwards the client update to the newly elected primary server.

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.

2.1.3. 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*, i.e., no Byzantine failures [12]. Security of the protocol follows from the use of secure RPC channels, mandatory in NFSv4, for server-to-server communication.

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 [4]. 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. We require an active view to contain a majority of the replication servers. A replication server rejects any further client update request should its active view shrinks less than majority. The primary server can relinquish its role only after distributing the new active view to all active servers. 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 "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. [†]

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 (this is possible since a majority of acknowledges can still be gathered from the left active servers). 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 acknowledgments from a majority of the replication servers so it cannot process any client 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 re-produce the computation results.

In the special case that a primary server fails after distributing a new active view to some but not all replication servers, the active views maintained by the replication servers become inconsistent. A request from a server that is eliminated from the new active view can be accepted by the servers that still hold the old active view. However, since the primary server has not relinquished its role for the updated file or directory, no other replication servers can modify that same object. Thus, the consistency of data is unaffected. The copies of active view converge until a new active view is generated and distributed during the modification of another file or directory.

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. After recovery, all the

[†]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.

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.

2.2. 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 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 2.2.1 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 2.2.2 discusses revisions to the primary server election. Section 2.2.3 then investigates mechanisms to balance performance and concurrency related to the two control types.

2.2.1. 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** *controls* 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.

```
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 3. Using and granting deep and shallow controls.

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 3. 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 descendants whose primary servers are not empty. 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.

2.2.2. 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. To guarantee progress during conflicts, we extend the consensus algorithm for primary server election 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 3, 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 is guaranteed with the original consensus algorithm.

2.2.3. 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 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 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.

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. Our study shows that timeouts as short as one second are long enough to reap the performance benefits of deep control [44]. Combined with our assumption that failure is infrequent, we anticipate that the performance gains of deep control far outweigh the potential cost of server's failing while holding deep control on directories.

3. EVALUATION

In this section, we explore the performance of NFSv4.r with the NAS Grid Benchmarks over simulated wide-area networks. We measured all the experiments presented in this paper with a prototype implemented in Linux 2.6.16 kernel. Servers and clients all run on dual 2.8GHz Intel Pentium4 processors with 1024 KB L2 cache, 1 GB memory, and Intel 82547GI Gigabit Ethernet card onboard. The number of bytes NFS uses for reading (rsize) and writing files (wsize) is set to 32768 bytes. We use Netem [22] to simulate the network latencies. To focus on evaluating the performance impact caused by WAN delays, we do not simulate packet loss in our measurements, and enable the `async` option (asynchronously write data to disk) on the NFS servers. All numbers presented are mean values from five trials of each experiment; standard deviations (not shown) are within five percent of the mean values.

The NAS Grid Benchmarks (NGB), released by NASA, provide an evaluation tool for Grid computing [20]. The benchmark suite evolves from the NAS Parallel Benchmarks (NPB), a toolkit designed and widely used for benchmarking on high-performance computing [13]. An instance of NGB comprises a collection 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 in VP fulfill the role of flow solver, post processor, and visualization, 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 4 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.

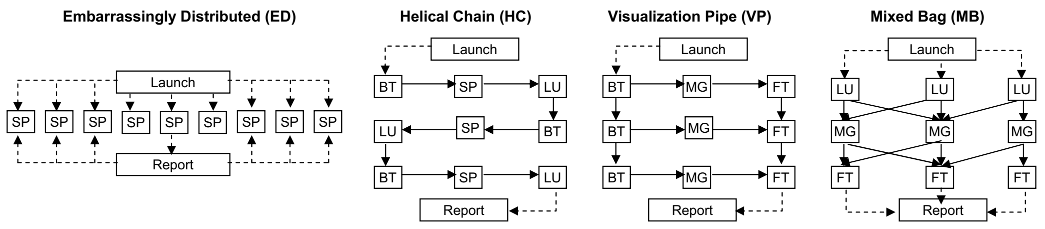

Figure 4. Data flow graphs of the NAS Grid Benchmarks.

Table I. Amount of data exchanged between NGB tasks.

	HC			VP				MB		
Class	BT→SP	SP→LU	LU→BT	BT→MG	MG→FT	BT→BT	FT→FT	BT→LU	LU→MG	MG→FT
S	169K	169K	169K	34K	641k	169K	5.1M	N/A	34K	641K
W	1.4M	4.5M	3.5M	271K	41M	1.4M	11M	N/A	702K	41M
A	26M	26M	26M	5.1M	321M	26M	161M	N/A	5.1M	321M
B	102M	102M	102M	21M	321M	102M	641M	102M	21M	321M

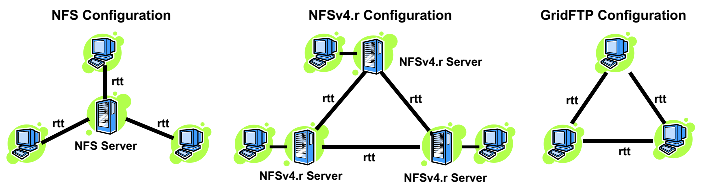

Figure 5. NGB evaluation experiment setup.

Launch and Report do little work; the former initiates execution of tasks while the latter collects and verifies computation results. [‡] The NGB instances include different problem sizes (denoted *Classes*). For the experiments presented in this paper, we use four Classes: S, W, A, and B. Table I 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, NFSv4.r, and GridFTP. Figure 5 illustrates the experiment setup.

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 NFSv4.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 software we use is `globus-url-copy` from Globus-4.0.2

[‡]Figure 4 illustrates the Data Flow Graphs for Class S, W, and A. For Class B, ED includes 18 parallel SP tasks; HC includes 6 layers of BT, SP, and LU; VP includes 6 layers of BT, MG, and LU; MB includes 4 layers: BT, LU, MG, and FT, and each layer includes 4 tasks [20].

toolkit. In our experiments, we start eight parallel data connections in each GridFTP transfer, which we found provides the best-measured performance for GridFTP. NFSv4.r also supports parallel data connections between replicated NFS servers, but this optimization provides small performance benefit in the experiments presented here. Thus, we report results measured with a single server-to-server data connection only. Focusing on performance comparison, we disable encryption of GridFTP, NFS, and NFSv4.r. The authentication options are enabled in all of the systems, but their performance impacts are negligible according to our analysis.

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: tasks 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 NFSv4.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 tasks and the launch script poll for the status of the required input data and tasks with semaphore files, as above.

Figure 6 shows the results of executing NGB on NFS, NFSv4.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 [23]. Furthermore, in some cases, it is hard for users to determine which files to stage [39]. With NFS and NFSv4.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 6 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 (W and A), the experiments take a very long time to execute when the RTT increases to 120 msec. 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 NFSv4.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), NFSv4.r outperforms GridFTP, because the latter requires extra time to deploy dynamically created scripts and has extra Globus-layer overhead. For larger class sizes, the performances of NFSv4.r and GridFTP are generally comparable. The only exception is the experiment with Class B of VP. There, the running time on GridFTP is about 50% longer than the time measured when running the benchmark on NFSv4.r. A closer analysis shows that the observed performance difference is caused by the memory contention that Globus introduces.

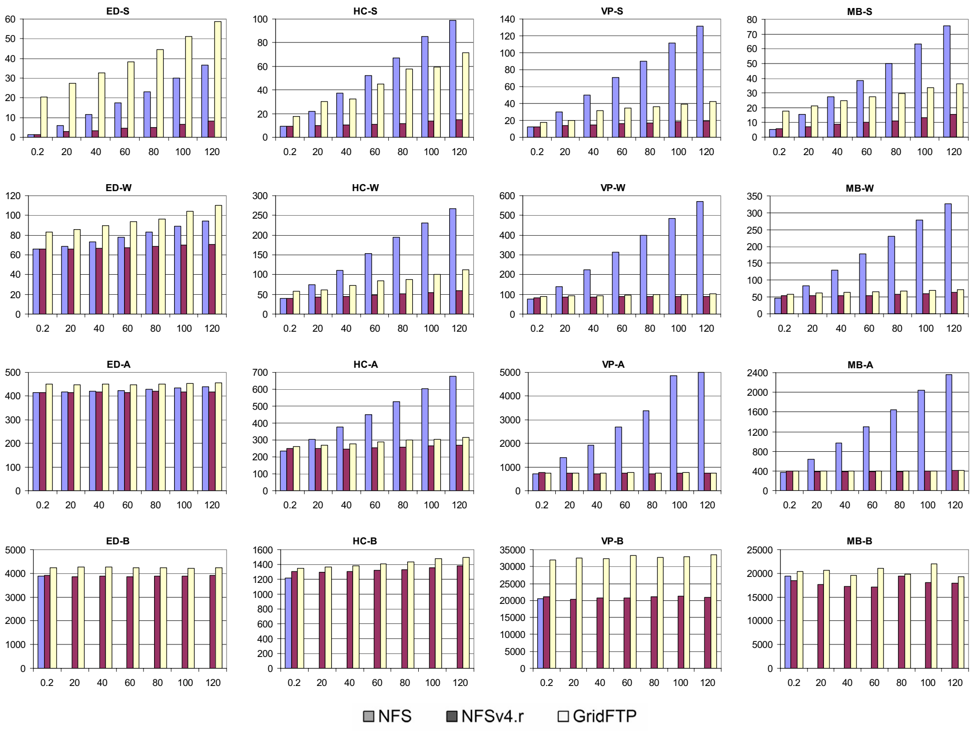

Figure 6. Turnaround times (seconds) of NGB on NFS, NFSv4.r, and GridFTP.

In each figure, X-axis denotes the RTT (milliseconds) shown in Figure 5, and Y-axis denotes the measured turnaround times. For Class B, we omit the experiments of accessing a single remote NFS server, which take extremely long times; the results of using a single local NFS server are presented on the left side of each figure for comparison.

Globus starts a job manager for each scheduled task on a computing node. The performance overhead added by Globus job managers is usually small. However, a FT task of Class B demands a large amount of memory. As a result, the Globus job managers cause the memory contention on the computing node where FT tasks execute and lead to a noticeable slowdown. The experiment with Class B of MB also experiences the memory contention problem. The Class B of MB consists of 4×4 tasks (see footnote 2). As we assign tasks in round robin, two FT tasks are scheduled to run on the same computing node. Due to memory contention, the execution times for these two tasks are significantly longer than executing them individually and we observe the relatively high variances in the measured results of this experiment. However, all the experiments with Class B of MB experience this slowdown, so the performance difference is small. We anticipate that the memory contention problem described here can be avoided with a well-designed job scheduler. Since the focus of our study is on data access in Grid computing, more detailed discussions are beyond the scope of this paper.

In summary, 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.

4. RELATED WORK

Replicated File Systems. Echo [9] and Harp [28] are file systems that use the primary copy scheme to support replication. Both 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.

Recent years have seen a lot of work in peer-to-peer (P2P) file systems, including OceanStore [34], Ivy [30], and Pangaea [35]. These systems address 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 [21] to improve concurrency and performance of simultaneous transactions. In distributed file systems, Frangipani [40] 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 [5] 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, such as Storage Resource Broker (SRB) [7], NeST [8], and Chimera [18]. Most of these systems provide extended features by defining their own API. In order to use them, an application has to be re-linked with their libraries. Consequently, scientific researchers are generally hesitant to install and use these Grid softwares.

In GPFS-WAN [1], each computing site provides its own cluster file system that is exported to the other sites as part of a common global file system. The system, in operation on the TeraGrid [3], demonstrates the promise of using standard Unix I/O operations for Grid data access. The existing prototype focuses on delivering high throughput for massive file transfers, rather than reducing the cost of individual I/O and metadata operations.

5. 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. The 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.

ACKNOWLEDGEMENTS

We gratefully acknowledge the support of NSF Middleware Initiative Grant No. SCI-0438298 and Network Appliance, Inc.

REFERENCES

1. GPFS. http://www.teragrid.org/userinfo/data/gpfswan.php.
2. GridFTP: Universal data transfer for the Grid. Globus Project, white paper, 2000.
3. TeraGrid. http://www.teragrid.org/.
4. A. E. Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of database systems*, pages 215–229, 1985.
5. A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Operating System Review*, 36(SI):1–14, 2002.
6. B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Proceedings of the 8th IEEE Symposium on Mass Storage Systems and Technologies*, page 13, 2001.
7. C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5, 1998.
8. 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 Symposium on High Performance Distributed Computing*, pages 3–12, 2002.
9. A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Palo Alto, CA, USA, 10 1993.
10. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.
11. 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. *Journal of Network and Computer Applications*, 23:187–200, 1999.
12. F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, pages 158–179, 1985.
13. T. L. D.H. Bailey, J. Barton and H. S. (Eds.). The NAS parallel benchmarks. Technical Report RNR-9 1-002, NASA Ames Research Center, Moffett Field, CA, 1991.
14. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, 1988.
15. M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 127–140, 1983.
16. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, 1985.
17. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1998.
18. 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 14th Conference on Scientific and Statistical Database Management*, pages 37–46, 2002.
19. J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

20. M. Frumkin and R. F. V. der Wijngaart. NAS Grid benchmarks: A tool for grid space exploration. *Cluster Computing*, 5(3):247–255, 2002.
21. J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modeling in Data Base Management Systems*, pages 181–208, 1976.
22. S. Hemminger. Netem - emulating real networks in the lab. In *LCA2005*, April 2005.
23. K. Holtman. CMS Data Grid system overview and requirements. The Compact Muon Solenoid (CMS) Experiment Note 2001/037, CERN, Switzerland, 2001.
24. J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, 1991.
25. L. Lamport. The parttime parliament. *ACM Transactions on Computing System*, 16(2):133–169, 1998.
26. L. Lamport. Lower bounds on asynchronous consensus. In H. W. André Schiper, Alex A. Shvartsman and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.
27. L. Lamport. Fast paxos. Technical Report MSR-TR-2005-112, Microsoft Research, Mountain View, CA, USA, 2005.
28. B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 226–38, 1991.
29. D. Malkhi, F. Oprea, and L. Zhou. Omega meets paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 199–213, 2005.
30. A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2002.
31. B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *USENIX Summer*, pages 137–152, 1994.
32. B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd international system administration and networking conference*, page 94, 2000.
33. 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.
34. S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The Oceanstore prototype. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 1–14, 2003.
35. Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. *SIGOPS Opererating System Review*, 36(SI):15–30, 2002.
36. 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 Operating System Review*, 19(5):35–50, 1985.
37. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.
38. S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol, RFC 3530, 2003.
39. 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 International Symposium on High Performance Distributed Computing*, page 152, 2003.
40. C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.
41. B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 59–59, 2001.
42. J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):563–576, 1999.
43. J. Zhang and P. Honeyman. Naming, migration and replication in NFSv4. In *Proceedings of the 5th International Conference on System Administration and Network Engineering*, 2006.
44. J. Zhang and P. Honeyman. Hierarchical replication control in a global file system. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 155–162, 2007.