Naming, Migration, and Replication in NFSv4

Jiaying Zhang and Peter Honeyman

Center for Information Technology Integration
University of Michigan
Ann Arbor

Abstract

The NFSv4 protocol includes features intended to enhance its use in global, wide-area networks. In this paper we describe the design and implementation of a prototype for global name space, transparent migration and fine-grained replication that uses and extends these features. In our system, directory migration and replication use the FS_LOCATIONS attribute to redirect I/O requests. For read/write replication, server redirection provides concurrency and consistency during replica updates. Strict consistency is guaranteed when the system is free of failures. In case of network partition or server failure, we offer the option of increasing availability while relaxing consistency by supporting ordered writes instead of strict consistency. The result is a system built on the Internet standard for distributed filing that features good performance and scalability, high availability, and fine-grained consistency.

1. Introduction

The Network File System [1, 2] is a popular distributed file system developed by Sun Microsystems in the early 1980s. The primary goal of NFS was to give users at workstations transparent access to files across a local area network (LAN). Over its long history, that original goal has been extended.

Version 3 of NFS, the popular and widely adopted current version, was designed for an environment quite different from today's:

- Computers were much less powerful.
- Operating systems were much less reliable.
- Access to remote computers was largely limited to LANs.
- The wide area network known as the Internet was still in its infancy.
- Popular security mechanisms were easily breached.

Today's highly connected world, populated by computers that would have been considered massively powerful when NFSv3 was designed, presents problems for NFS over the Internet, where security, performance, and interoperability are critical. For example, the security vulnerability reported by Sommerfeld in 1987 [29], which allows an impostor to gain unauthorized access to an NFS file system by spoofing file handles, remains a potent threat to present day NFSv3 deployments.

Version 4 of NFS [3] addresses these problems. NFSv4 retains essential characteristics of previous versions of NFS, yet also features improved access, performance, and security over WANs. To adhere to these goals, this paper focuses on mechanisms that facilitate wide area interoperation and access: a global name space and support for transparent migration and replication.

Naming plays an important role in distributed file systems. In wide area networking, several principles guide the design of a distributed file system name space. First, a global name space that provides a common frame of reference is desirable. Second, name space operations should scale well in the face of wide or massive distribution. Third, transparent migration and replication require location independent naming.[1]

We pay particular attention to data replication. There are two primary reasons for replicating data. First, replicating data improves availability in the face of failure by allowing users and applications to switch from a failed data server to a working one. Second, replication improves performance by allowing access to distributed data from nearby or lightly loaded servers. This is especially significant in allowing a distributed system to scale in size and span.

The fundamental challenge to data replication is keeping replicas synchronized when updates occur. Although strong consistency guarantees are ideal, performance and availability requirements often force system designers to compromise. We also take these considerations into account. Our server redirection mechanism provides strict consistency without affecting normal read performance when the system is

[1] By t*ransparent migration and replication* we mean using abstract concepts and mechanisms to hide the fact that a resource is migrated or replicated, thus presenting users the appearance of a single unified system [6].

failure-free. In the case of network partition failures, we offer a choice of consistency models: strict consistency or ordered writes, allowing different availabilities. File system workload reports and the evaluation data collected with our prototype implementation suggest that our system offers most applications acceptable performance and consistency, even distributed applications that span the Internet.

Our work is also motivated by the need for easy administration of the file system. In an environment in which storage systems are becoming larger and more complex, storage management plays a prominent and increasing role and expense in system administration. This makes ease of administration an important goal in file system designs. The past twenty years has seen numerous research and development efforts intended to facilitate file system administration. For example, in AFS, *volumes* were developed to organize data within a storage complex by breaking the association between physical storage and the unit of migration and replication [28]. Because they are independent of the physical configuration of the system, *volumes* provide a degree of transparency in addressing, accessing, and storing files. This also facilitates data movement and optimization of existing storage.

The lack of transparent file relocation support in NFSv3 makes data movement among file servers a cumbersome administration task, and can disrupt the ongoing work of users and applications while data is distributed to new locations. Migration and replication address this problem: our design allows data to be created, copied, removed, and relocated easily within NFSv4 without disrupting service to clients. We also provide a framework for automatic failover and load balancing, which are highly desirable in the wide area environment.

The remainder of this paper is organized as follows. Section 2 introduces the system consistency and failure model. Section 3 presents the design of the system. Section 4 describes a prototype implementation. Section 5 evaluates the performance of the prototype. Section 6 discusses related work. Section 7 summarizes and concludes.

2. System Model

2.1 Consistency Model

The main problem introduced by replication is maintaining consistent copies: any time a replica is updated, that copy becomes different from the others. To synchronize replicas, we need to propagate updates in a way that hides temporary inconsistencies, but doing so may degrade performance, especially in large-scale distributed systems. The performance problem is easier to solve if consistency guarantees can be relaxed.

Most distributed file systems provide one of four consistency models:

- **Strict consistency** in file systems gives "one copy" semantics, i.e., a guarantee that all I/O operations yield identical results at all nodes at all times [2]. Strict consistency guarantees that any read to a shared data item X returns the value stored by the most recent write operation on X.
- An essential correctness criterion for replicated databases, known as *one-copy serializability,* requires that the concurrent execution of transactions on replicated data be equivalent to a serial execution on non-replicated data [16]. A quiescent, failure-free system eventually achieves agreement at all replicas. In file systems, this is equivalent to enforcing **ordered writes**. Applications do not necessarily see updates simultaneously, but they are guaranteed to see them in the same order.
- The popularity and success of NFSv3 prove that many applications can briefly tolerate some degree of inconsistency. In a **time bound inconsistency** model, a client may access an old version copy of data if the update to the object was made in the bound time. After the bound time, clients are guaranteed to see the updated data. This consistency model is often implemented in systems using periodic messages to check data consistency or to detect network partitions.
- Some distributed file systems adopt **optimistic replication** to trade consistency for availability. In these systems, any copy can be read or updated at anytime. This is important for applications that require continuous access to data. However, overlapping optimistic reads and writes can introduce inconsistencies during partition, forcing optimistic systems to provide conflict detection schemes and resolution heuristics.

Strict consistency, although ideal, comes at a great cost, so it is not unusual for a distributed file system to relax consistency for performance or availability. Furthermore, not all applications need strict guarantees. Based on these considerations, we offer the option of relaxing our strict consistency guarantee in a failure-free system to guaranteeing ordered writes in the case of node or network failure. We stop short of supporting time bound inconsistency because of the cost in network overhead [23] and demonstrated inconsistent access in day-to-day operation [25]. Although optimistic replication addresses the former problem, it exacerbates the latter; we rule out optimistic replication so that we can offer conflict-free guarantees.

2.2 Failure Model

A system that does not adequately provide the services for which it was designed *fails*. There are several types of failure in distributed systems, *omission failure*, *performance failure* and *Byzantine failure* [9].

An *omission failure* occurs when a component fails to respond to a service request. Typical omission failures include replication server or communication link crashes. A *performance failure* occurs when a system component fails to respond to a service request within the time limit specified for the delivery of that service. Occasional message delays caused by overloaded replicas or network congestion are examples of performance faults. An important subclass of omission and performance failures is *partition failure*. *Partition failure* occurs when a network failure partitions replicas into two or more communicating groups. *Byzantine failure* is also called *arbitrary* or *malicious* failure [17, 18]. In Byzantine failure, components may act in arbitrary, even malicious ways. Compromised security can lead to Byzantine failure.

Although security breach is increasingly common in the Internet, Byzantine failure is beyond the scope of the work described here, which narrows our focus to two kinds of failure: crashed nodes and partitioned networks. Furthermore, we assume these failures are rare, although this does not affect the security or correctness of our protocol. Finally, rapid and dramatic changes in server state or network conditions are not the focus of this paper.

3. Design

To cope with the requirements of a wide area environment, we pursue the following goals in our design.

A single global name space. A global name space for all files in the system encourages collaborative work and dissemination of information by providing everyone a common frame of reference. Users on any NFS client, anywhere in the world, can then use an identical rooted path name to refer to a file or directory.

Performance. Good system performance is always a critical goal. Our design aims to make common accesses fast. Insights from workload analysis of real file systems [15, 24–26] guide our design. We consider the following cases, ordered by expected frequency.

- **Exclusive read:** most common. Support for replication should add negligible overhead to the cost of unshared reads.
- **Shared read:** common. Blaze [15, 24] observes that files that are used by multiple workstations make up a significant proportion of read traffic. For example, in testing, files shared by more than one user make up more than 60% of read traffic, and files shared by more than ten users make up more than 30% of read traffic. This motivates us to impose minimal additional cost for shared reads.
- **Exclusive write:** less common. Workload studies show that writes are less common than reads in file systems. When we consider access patterns for data that need to be replicated in wide area network, this difference should become even larger. This allows us to design a replication file system within which data updates are more expensive than in one-server-copy cases, and still get good average-case performance.
- **Write with concurrent access:** infrequent. A longer delay can be tolerated when a user tries to access an object being updated by another client.
- **Server failure and network partition:** rare. Special failure recovery procedures can be used when a server crashes or a network partitions. During the time of failure, write accesses might even be blocked if strong consistency must be guaranteed without doing much damage to overall throughput averages.

Scalability. Successful distributed systems tend to grow in size. To provide scalability, our design follows two principles. First, centralized services are avoided. Second, the performance of the most common file accesses are affected little when the system grows in size and span.

Availability. In distributed systems, replication is often used to increase data availability. To design a system that operates in partition failures, the competing goals of availability, consistency, and performance must be balanced. One of our objectives is a replication scheme that tolerates a large class of failures, guarantees ordered writes or strict consistency, and provides superior read performance.

In the rest part of this section, we present the system design details targeting these goals.

3.1 Global Name Space and File System Migration & Replication

In this project, the NFSv4 protocol is extended to provide a single shared global name space. By convention, a special directory /nfs is the global root of all NFSv4 file systems. To an NFSv4 client, /nfs resembles a directory that holds recently accessed file system mount points.

Entries under /nfs are mounted on demand. Initially, /nfs is empty. The first time a user or application accesses any NFSv4 file system, the referenced name is forwarded to a daemon that queries DNS to map the given name to one or more file server locations, selects a file server, and mounts it at the point of reference.

The format of reference names under /nfs directory follows Domain Name System [4] conventions. We use a TXT Resource Record [5] for server location information.[2] A RR in DNS can map a reference name to multiple file servers, in this case replicas holding the same data. This provides for transparency in our file system migration and replication implementation. When a file system is replicated to a new server, the administrator updates the DNS server to add a mapping from the file system reference name to the new NFSv4 server location. Similarly, when a file system is migrated to another NFSv4 server, the old mapping is updated to point to the new server. Once the migration is completed, the old server returns NFS4ERR_MOVED for subsequent client requests. Upon receiving the NFS4ERR_MOVED error, the client queries DNS to get the new file system locations and connects to the specified new server. In addition to file system locations, other information such as mount options can also be carried in DNS RRs.

Here is an example of a pathname in the global name space: /nfs/umich.edu/lib/file1, where umich.edu is the reference name of the NFSv4 file system provided by the University of Michigan and /lib/file1 is the path under that file system.

3.2 File System Name Space and Directory Migration & Replication

The file system name space provided by an NFSv4 server is called a *pseudo file system*. A pseudo file system glues all the rooted hierarchies exported by an NFS server into a single tree rooted at /. Portions of the server name space that are not exported are bridged into one exported file system so that an NFSv4 client can browse seamlessly from one export to another. Exported file system structures are controlled by servers, thus a server dictates a common view of the file system that it exports. This feature is essential for support of a global name space, and reflects the intention of the NFSv4 protocol designers to provide that support.

We implement directory migration and replication by exporting a directory with an attached reference string that includes information on how to get directory replica locations, such as replica lookup methods and lookup key. Four types of reference string are implemented in our prototype: *LDAP, DNS, FILE* and *SERVER REDIRECT.* The format of each type is described in Section 4.2.

When a client first accesses a replicated directory, it is sent the reference string for that directory. The client uses the reference string to query the replica locations of that directory. After selecting a nearby replication server, the client continues its access. When a directory is migrated to another server, the original server returns NFS4ERR_MOVED error for subsequent directory requests. Receiving this error, the client obtains the reference string of the migrated directory by examining the FS_LOCATIONS attribute and using its contents to connect to the specified server.

Support for multiple lookup methods allows an organization to maintain replica location information as it desires. No centralized name service is required.

3.3 FS_LOCATIONS Extensions

FS_LOCATIONS, a recommended attribute in the NFSv4 protocol, intended to support file system migration and read-only replication. An attempted client access to a migrated file system yields an NFS4ERR_MOVED error; retrieving the FS_LOCATIONS attribute gives new locations for the file system. For replication, a client's first access to a file system might yield the FS_LOCATIONS attribute, which provides alternative locations for the file system.

The FS_LOCATIONS attribute allows clients to find migrated/replicated data locations dynamically at the time of reference. We also use the FS_LOCATIONS attribute to communicate migration and replication location information between servers and clients, in a way that varies from the published protocol. First, in our design, the FS_LOCATIONS attribute is used to provide reference strings of replica locations, instead of real locations. It is the client's responsibility to translate reference strings into replica locations. Second, we use the FS_LOCATIONS attribute to provide directory migration and replication, instead of the coarser-grained file system migration and replication. Finally, we use the FS_LOCATIONS attribute to

[2] We intend to replace this with the emerging SRV standard for locating servers [32]. There is already an NFS type defined for SRV resource records.

support concurrent accesses in mutable replications, which we describe in detail in the next section.

3.4 Mutable Replication

The scheme described in the last two subsections can efficiently support read-only replication. However, to support read/write replication, we need proper mechanisms to distribute updates when a replica is modified and to control concurrent accesses when writes occur. We considered several mechanisms for our design.

- **Reader checks.** One strategy is for a reader to check a number of replicas on every read request to be sure that it sees the most up-to-date data. This solution can guarantee strong consistency, but adds substantial overhead to normal read requests.
- **Periodic query:** By having replicas send periodic query messages, the system allows a narrow window of inconsistency but can guarantee data consistency after a defined time bound. With this method, the system can automatically discover and recover from failure. However, periodic query messages add considerable overhead and network traffic to the system, which is especially wasteful when no updates are undertaken and all servers are working properly, the most common case by far.
- **Server redirection.** The strategy we adopt provides strong consistency at little cost to exclusive or shared readers. When a client opens a file for writing or modifies a directory, the selected server temporarily becomes primary for that file or directory by contacting all other replication servers and instructing them to redirect all accesses to it. The strategy differs from the usual primary copy scheme in that it allows late and dynamic binding of the primary server. We support two consistency models that differ only in case of failure. We guarantee strict consistency by blocking all updates while a failed server is under repair. Alternatively, by allowing updates only in a partition that includes a majority of the replication servers, we can guarantee ordered writes. We present details in the following subsections.

3.4.1 File Updates

We enforce consistent access by redirecting all clients to a primary server when a file is opened for writing. When a client opens a file for writing, the relevant server temporarily becomes the primary server for that file. All other replication servers are instructed to redirect subsequent accesses to the primary server. When the file is closed, the primary

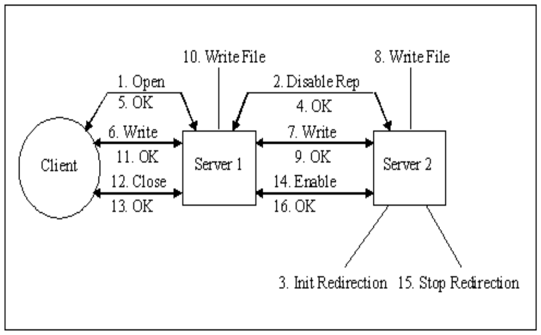

Figure 1: File modification. 1. A client issues an open request to a server. 2. The server instructs other replication servers to redirect requests, making it the primary server for the file. 3. Replication servers comply. 4. Replication servers acknowledge the request. 5. The primary server acknowledges the open request. 6. The client sends writes to the primary server. 7. The primary server distributes the writes to other replicas. 8. Other servers update the written file. 9. Other servers acknowledge the update. 10. The primary server updates the written file. 11. The primary server acknowledges the client's writing request. (Steps 6 through 11 may be repeated several times.) 12. The client issues a close request. 13. The close request is acknowledged. 14. The primary server instructs the redirected servers to re-enable replication. 15. The redirected servers disable redirection. 16. The (formerly) redirected servers acknowledge the request to re-enable replication.

server withdraws from its leading role by re-enabling replication on the other replication servers.

While the file is open for writing, the primary server is the only server for the file. Client access requests sent to other servers are redirected to the primary server.

While replication is disabled, the primary server is responsible for distributing updates to other replicas. We considered two strategies for update distribution during the project design. The first is the simple and obvious strategy of distributing updates when the modified file is closed. The second strategy distributes updated data to other replicas as it arrives.

Although naïve, the update-on-close strategy does avoid multiple updates, should the file be written several times. This strategy is sound if a file is overwritten several times while it is open. However, if a client writes a massive amount of data to a file and then closes it, the close operation takes a very long time to return.

On the other hand, distributing updated data to other replicas every time the primary server receives a write request eliminates the response time penalty for a client's close request.[3] It also facilitates recovery

[3] It also removes the possibility that the close might fail for extraordinary reasons, negating successful writes. E.g., AFS users learned the hard way about applications that

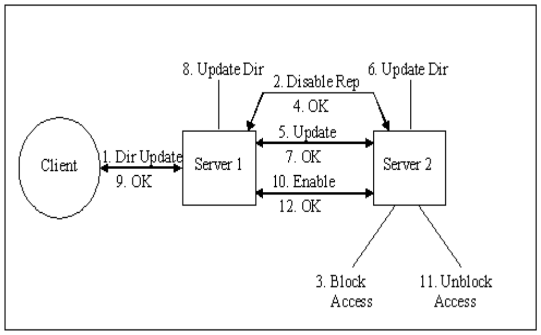

Figure 2: Directory modification. 1. A client issues a directory update request to a server. 2. The server instructs other replication servers to block any access to this directory. 3. Replication servers comply. 4. Replication servers acknowledge the request. 5. The primary server distributes the update request. 6. Other servers update the directory. 7. Other servers acknowledge the update. 8. The primary server processes the directory update request. 9. The directory update request is acknowledged. 10. The primary server instructs the other servers to re-enable access. 11. The redirected servers restore access to the directory. 12. Other servers acknowledge the request to re-enable replication.

from primary server failures: the client can simply reissue the failed request after connecting to a new server. However, this strategy may produce unnecessary network traffic caused by overlapped writes. Moreover, a client that writes too quickly can exhaust its local buffers before they drain and eventually block.

We prefer the latter scheme: we are willing to squander network to improve latency and we want to provide a simple failure recovery procedure for clients. Yet, by making client writes to the primary server synchronous with updates to the disabled replication servers, it appears that our strategy adds considerable latency instead of improving it. The paradox is resolved by observing that NFSv4 clients themselves delay their writes, detaching application performance from the high-latency updates to the primary server. Furthermore, the client's delayed-write policy makes it likely that the updates will be long-lived.

In addition to written data, each update message from the primary server to other replicas also includes the metadata related to the update, such as modification time and file size. Each replication server modifies the metadata accordingly after updating the file. This guarantees that the metadata of the file is consistent among replicas, which as we show in Section 3.4.3, facilitates recovery from failures. Figure 1 illustrates the file modification procedure with two replication servers.

3.4.2 Directory and Attribute Updates

Directory modifications include the creation, deletion, and modification of entries in a directory. Unlike file writes, little time elapses between the start and finish of a directory update, which reduces the likelihood of concurrent access to a directory while it is being updated. Therefore, instead of redirecting access requests to a replicated directory while an update is in progress, replicas block access requests to a replicated directory until the primary server distributes the update and re-enables replication. Directory modification is illustrated in Figure 2.

Like directory modifications, attribute updates are processed quickly, so we handle them in the same way.

3.4.3 Special Cases

Conflict. Two or more servers may try to disable replication of a file or directory at the same time, with the result that some replication servers are disabled by one server, some by another server. When these servers are in the same partition, contention is always apparent to the conflicting servers. We resolve the conflict by having conflicting servers cooperate: the server that has disabled more replicas is allowed to continue; the server that has disabled fewer replicas hands its collection of disabled replicas to the first server. Servers that are not in the same partition cannot conflict because we do not allow redirection in a minority partition.

Failure. Our system guarantees strict consistency when all replicas are in working order. Failure complicates matters. Different kinds of failure may occur, including client failure, replication server crash failure and network partitions. Here we briefly describe the failure detection and recovery mechanisms for each case. A detailed description and proof of correctness forms a companion paper [21].[4]

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. After a client failure, the server receives no further renewal requests, so the lease expires. Once the primary decides that the client has failed, it closes each file opened on behalf of the failed client. If the client was the only

(unwisely) rely on the success of their writes and neglect to check the status of close, but lose data if the file server runs out of space.

[4] Note to reviewers: the companion paper has been submitted for publication elsewhere. If you wish to see it, and if it is not yet available as a CITI Tech. Report, please ask the Program Chair for a copy and we will provide it.

writer, then the primary re-enables replication for the file at this time. The file contents reflect all writes acknowledged by the primary server prior to the failure.

A server might be unable to ensure data consistency at all other replicas when there is a crashed server or network partition. Although they are hard to distinguish from afar, there is an essential difference: a crashed server no longer responds to any clients, but a partitioned server can still serve client requests that originate in its partition. Consequently, replication servers in a minority partition may unwittingly serve stale data.

To address this problem, we support two options that offer different consistency guarantees and availabilities in case of failure: ordered writes and strict consistency. In the first option, write operations are guaranteed to be serialized in the system. However, in a minority partition, a read request may be served with stale data. In the second option, the system guarantees that the data read by a client is fresh at the cost of blocking write operations in the system.

To support ordered writes, the system maintains an *active group* view among replicas and allows updates only in the active group. We require an active group to contain a majority of the replicas. During file or directory modifications, the primary server removes from its active group view any replicas that fail to acknowledge the replication disabling requests or update requests. The primary server updates its local copy and acknowledges a client write request only after it has received update acknowledgements from a majority of active replicas. If the active view shrinks to less than a majority, the primary server fails the client's request. The primary server sends its active view to the replication servers when it re-enables replication. A server not in the active view may have stale data, so the servers that re-enabled replication refuse any later request that comes from a server not in the active group. A failed replication server can rejoin the active group only after it synchronizes with the up-to-date copy.

We emphasize that the granularity of a view is a single file: different files, even ones in the same directory, can have different views. We assume failure is rare, so we expect most views to contain the full set of replication servers, which suggests an economical representation for a full view.

If a primary server crashes or is separated in a minority partition, a client (in the majority partition) detects this failure when an access request times out. The client selects a working replication server and informs it of the failure of the primary server. After verifying that replication is disabled and that the primary server has failed, the replacement works with the other replication servers to recover from the primary server failure. Briefly, the replacement asks other active replicas for permission to become the new primary server. If this succeeds, the replacement synchronizes all active replicas with the most up-to-date copy found in the majority partition. It then distributes a new active group view and re-enables replication on the active servers.

With the mechanisms described above, our system guarantees ordered writes and continuously serves clients' requests as long as a majority of replicas are in working order. If there are multiple partitions and no partition includes a majority of the replication servers, no write requests can be served until the partition heals. We assume this happens rarely.

Using the view change properties and rules proposed by El-Abbadi et al. [11], we are able to prove that our protocol guarantees ordered writes in the face of node crash or network partition [21]. Security of the protocol follows from the use of secure RPC channels, mandatory in NFSv4, for server-to-server communication.

4. Implementation

In this section, we report on a prototype implementation of the design described in Section 3. First, we describe a modified Automount daemon, used to automatically mount and unmount NFSv4 servers. Then we discuss our implementation of replication support for NFSv4. All development was done in Linux.

4.1 Automount

Automount and AutoFs are tools that allow users of one machine to mount a remote file system automatically at the instant that it is needed. Automount, often referred as AMD, is a daemon that installs AutoFs mount points and associates an automount map with each AutoFs mount point. AutoFs is a file system implemented in the kernel that monitors attempts to access a subdirectory within a designated directory and requests AMD to perform mounts or unmounts there. On receiving a mount request from the kernel, the daemon uses the automount map to locate a file system, which it then mounts at the point of reference within the AutoFs file system. If the mounted file system is not accessed for a while, AutoFs instructs the daemon to unmount it.

Although Automount supports numerous mapping methods, support for DNS mapping is not provided in the current implementation, so we extended the

Automount daemon to support a DNS mapping method. The global root directory of NFSv4, `/nfs` by our convention, is made an AutoFs mount point with DNS mapping as the associated automount map method. We also made changes to provide communication between the NFSv4 client and the Automount daemon. When an NFSv4 client receives a reference string from a connected server, it passes the reference string to the modified Automount daemon. After receiving the request, the daemon uses the mapping method indicated in the reference string to locate one or more replicas. It then selects and mounts a replication server.

4.2 Replica List Maintenance and Lookup

NFSv4 uses `exportfs` utilities on the server side to export a directory. In the kernel, an export structure is maintained for each current accessed export. We extended the `exportfs` interface so that the reference string of a replicated directory can be passed into the kernel. The reference string is maintained in the corresponding export structure. When an NFS client encounters an export with an attached reference string, the server notifies the client and sends the reference string via the FS_LOCATIONS attribute.

We support four types of reference strings: *LDAP, DNS, FILE* and *SERVER REDIRECT*. The format of each type is listed below.

- **LDAP.** The format of an LDAP reference string is `ldap://ldapserver/lookup-key [-b searchbase] [-p ldapport]`. The LDAP server stores replica location records that can be queried with the lookup-key. A replica location record includes the server name holding that replica, the directory path where the replica located, and the server mount options. The lookup-key needs only to be unique in the LDAP server, which can be guaranteed when the mapping entry is created.
- **DNS:** the format of a DNS reference string is `dns://lookup-name`. The lookup-name format follows domain name conventions. The DNS carries replica location information as described in Section 3.1.
- **FILE:** the format of a FILE reference string is `file://pathname/lookup-key`. The pathname gives the path to the file storing a lookup-key to replica location mappings. The file should be stored in a place accessible to NFS clients, e.g., the parent directory of the replicated directory.
- **SERVER REDIRECT:** the SERVER REDIRECT lookup method is used to support directory migration and relocationas well as concurrent write access to replicas. The format of a SERVER REDIRECT reference string is `server://hostname:/path [mount-options]`, where hostname:/path gives the location of the replicated directory.

4.3 Mutable Replication Implementation

In mutable replication, a server needs to know the replica list before it issues a replication-disabling directive. Our implementation maintains this information dynamically, using the RPC cache. When a server wants to send replication disabling messages, it calls cache lookup with the reference string as the lookup key. If there is a cache hit, the cached value is returned. If a cache miss occurs, an upcall is made to a user-level handler, which performs the lookup and adds the queried data to the cache.

RPC calls are used to propagate updates and to transmit replication disabling and enabling messages among replicas. These messages are sent in parallel by the primary server, i.e., the primary server sends an RPC request to each of the disabled replication servers, and then waits for the reply from each replication server. In this way, the disabled replication servers can process updates concurrently; otherwise, serialized updates would clog the primary server. This mechanism is similar to MultiRPC [27].

5. Evaluation

Having describing the system architecture and implementation, we now present performance data collected with the prototype implementation.

To measure the cost of the global name space support and replication in our system, we ran several microbenchmarks that measure the overhead on individual system calls. The collected results are presented in Section 5.1 and 5.2. The system performance results for common read requests are not presented, as they are the same as the one-server-copy case. In Section 5.3, we present performance data that uses a modified Andrew benchmark to model a mix of file operations.

We implemented the NFSv4 client prototype on a 400MHz AMD-K6 with 128MB of memory. In our experiments with multiple replicated servers, the primary server is on the same LAN as the client. The client and the primary server are connected with switched 100 Mbps Ethernet. The round-trip latency (RTT) between the client and the primary server is around 400 μs. The client runs Linux 2.5.68 and the primary server runs Linux 2.5.70. The evaluation

Phase	Time (ms)
Upcall	1.28
Replica List Query (DNS)	1.49
Mount	37.5
Total	40.2

Table 1: First access. This table shows the delay when a client first accesses a file system within the provided global name space. The client is a 400MHz AMD-K6 with 128MB memory. The server is and Intel 1.8GHz P4 with 128MB memory. The client and the server are on a switched100 Mps Ethernet.

Query Method	Time (ms)
DNS TXT RR	1.49
LDAP	12.3
FILE	13.1
SERVER REDIRECT	0.007

Table 2: Lookup time for different querying methods. This table shows the lookup time of different querying methods, measured at the client. The DNS server and LDAP server used for querying are within the same local network as the client. The size of the file storing replica location information is 248 bytes.

data were measured at the client side with `get-timeofday`, which has microsecond resolution in Linux 2.5.68. All numbers presented in this section are mean values from three trials of each experiment. In all experiments, the standard deviations are within five percent of the mean values, so we omit reporting them.

5.1 Cost to Support Global Name Space

We measured the delay experienced by a client when first accessing an NFSv4 file system in the provided global name space. Table 1 presents the measured time in each phase during the first access. As expected, the mount phase takes the most time, as client and server need to mutually authenticate during mount. The total delay seen by the client is approximately 40 ms. This can be used to estimate the response time when the client first accesses a migrated or replicated directory, which will vary slightly when using different query methods. The client and the server used in this experiment are in the same local network; the response time increases in a WAN, but a client experiences this delay only on its first reference.

Table 2 presents the different query times used to lookup replica locations at the client. The DNS server and LDAP server used in the experiment are on the same LAN as the client. SERVER REDIRECT requires the least time, as the client does not need to process any query in this case. Among the other three methods, DNS TXT RR is the fastest with around 1.5 ms query time. LDAP and FILE – a little slower at 12 ms and 13ms query time, respectively – are still acceptable, as a client needs to query replica locations only on its first reference.

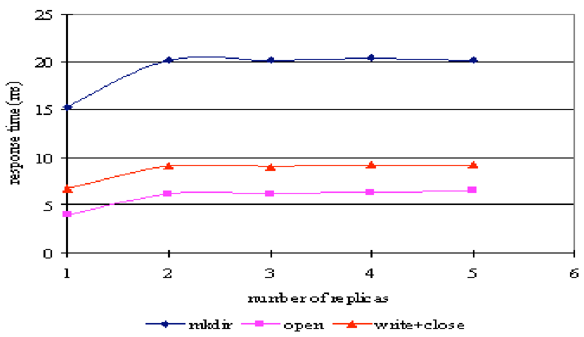

Figure 4: Three micro-benchmarks with varying number of replicas. This diagram shows the running time for open, write one byte + close, and mkdir system calls as the number of replicas increases in a LAN.

5.2 Replication Performance

To assess the cost of replication, we measured the client-perceived run times for `open`, `write` and `mkdir` system calls. Replication does not add any overhead for read-only opens in our system, so we always open for writing. Because the NFS client does not immediately send data written to the NFS server during write system calls, we measured the combined time of writing one byte and close.[5] In all experiments presented in this section, we use TCP as the transport protocol. The *rsize* and *wsize* are set as 4096 bytes.

We first measured the elapsed time for three micro-benchmarks respectively as the number of replicas increases in a local area network. The collected data are displayed in Figure 4. In this experiment, the RTT between the primary server and each replication server is around 260 µs. Except for the primary server, all servers used in this experiment have dual Intel 1.7 GHz CPUs and 2 GB memory. As shown in the table, replication induces a small overhead in the three system calls.

To test system performance in a wide area environment, we used the PlanetLab [30] infrastructure to simulate other NFS servers holding the replicated data. PlanetLab is an open, globally distributed platform consisting of 160 machines hosted by 65 sites spanning 16 countries. All of the machines are connected to the Internet, which creates a unique envi-

[5] The NFSv4 client does not send close request immediately after receiving a close system call, so the measured response time for write+close operation does not include the processing time for close RPC requests, but the cost of close is not affected in our design.

Location	Server	RTT (ms)
CITI (primary), Ann Arbor	1.8 GHz 128 MB	
Washington U, St. Louis	1.8 GHz 1 GB	21.8
Duke, Durham	128 MHz 1 GB	47.6
UCSD, San Diego	128 MHz 1 GB	67.4
HP Labs, Bristol	2.4 GHz 1 GB	119
U of Tokyo, Japan	140 MHz 512MB	169
CUHK, Hong Kong	1 GHz 512 MB	230

Table 3: Servers used in Figure 4 experiments. This table describes the PlanetLab Pentium servers used for the experiments in Figure 4. The RTT refers to the round-trip time between the replica and the primary server.

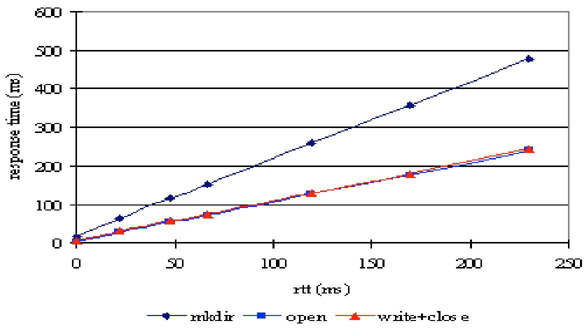

Figure 5: Three micro-benchmarks with varying RTT. This diagram shows the running time for open, write one byte + close, and mkdir system calls as the round-trip time between the primary server and the replica increases. In each measurement, the data are replicated on two servers: the primary server and a replica.

ronment for conducting Internet-scale experiments. Compared with network simulation tools, the most obvious advantage of PlanetLab is that network services deployed on PlanetLab platform experience all of the behaviors of the real Internet. A second advantage is that PlanetLab provides a diverse perspective on the Internet in terms of connection properties, network presence, and geographical location.

In our implementation, each NFSv4 server runs a kernel level daemon to receive RPC requests from other replicas. These RPC requests include replication disabling/enabling messages and update distribution messages, etc. However, it is not currently possible to test modified kernels on PlanetLab platforms. To overcome this, we conducted the experiments by running a user level RPC server to simulate the kernel daemon. The RPC request processing time on each replication server is simulated by delaying the reply by a specified amount time. We measured the processing time for each type of RPC message in a local machine with the real kernel daemon over many iterations and used the average as the specified

Location	Server	RTT (ms)
P (primary)	1.8 GHz, 128 MB	
Umich	2 servers, 128 MHz, 1 GB	1.18
Wustl	1.8 GHz 1 GB	21.8
HP	3 servers, 2.4 GHz, 1 GB	119

Table 4: Servers used in Figure 6 experiments. This table describes the PlanetLab Pentium servers used for the experiments in Figure 5. The RTT refers to the round-trip time between the replica and the primary server.

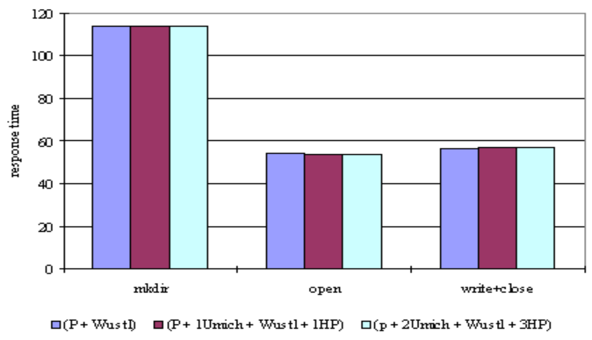

Figure 6: Three micro-benchmarks with varying replica sets. This diagram shows the running time for open, write one byte + close and mkdir system calls, with the fixed largest RTT between the primary server and the majority of other replicas, but different replica sets.

delay time in the simulation.[6] Although the request processing times vary on different machines, this difference has negligible effect on our evaluation, as the total response time for a RPC request over the Internet is dominated by the round-trip latency (RTT) between the RPC server and client.

Figure 5 displays the elapsed time overhead for three micro-benchmarks across a range of RTTs. The zero RTT case represents NFSv4 performance without replication. The related information about the servers used in this experiment is listed in Table 3. The chosen replication servers range from a host in St. Louis to one in Hong Kong. Except for the primary server, all other servers used in the experiment are on PlanetLab. In each measurement, the data are replicated on only two servers: the primary server and another replication server. The RTT listed in Table 3 refers to the round-trip time between the replication server and the primary server. This time varies as network conditions change. The time presented in Table 3 is the stable time measured empirically by

[6] We found almost no performance difference between the kernel implementation and user level simulator in our experiments. For this reason, and to simplify system configuration, the performance data for LAN replication are also collected this way.

Open Type	Time (ms)
Normal Open	5.41
Redirected Open	62.7

Table 5: Redirected open time. This table shows the response time of normal open and redirected open seen by the client. The servers and clients are on the same LAN.

sending ping messages from the primary server to the corresponding replication server.

As shown in Figure 5, all three micro-benchmarks demonstrate linear increases as the RTT between the primary server and the replication server grows. The growth rate for *open* and *write* operations are around 1. The response time for *mkdir* operation increases at twice the rate: before acknowledging a directory update request from the client, two RPCs (replication disabling and update) are sent from the primary server to the other replication server. The results are consistent with our expectation that over the Internet, the response time for a file or directory update request is dominated by the RTT between the primary server and other replication servers.

Because we use parallel RPC messages to distribute replication disabling/enabling messages and updates, we expect the total response time for a client's update request to be dominated by the largest RTT between the primary server and a majority of nearby replicas.[7] To test this conjecture, we measured the performance of the three micro-benchmarks with the fixed largest RTT between the primary server and the majority of other replicas, but different replica sets. The evaluation data are presented in Figure 6. The related information about the servers used in this experiment is listed in Table 4. Figure 6 shows that increasing the number of replicas has little effect on overall performance if the largest RTT among the primary server and the majority of replicas is kept fixed.

So far, we have not considered concurrent writes. Table 5 compares the response time for normal open and for redirected open. The redirected open refers to a client trying to access a file already open for writing at another server (primary server). In this case, the client is notified to connect to the primary server for file accessing. The two servers and two clients used in this experiment are all on the same local network. Table 5 shows that redirected open is more than ten times slower than normal open. Measurements and simulations [15, 24, 25] suggest that files are rarely write-shared in real workloads, so that redirected open will not occur often.

5.3 Andrew Benchmark

Tables 6-8 show the execution times for a modified Andrew benchmark run on a replicated NFSv4 directory in local and wide area networks. The Andrew file system benchmark [23] measures five stages in the generation of a software tree. Stage *(I)* creates the directory tree, *(II)* copies source code into the tree, *(III)* scans all the files in the tree, *(IV)* reads all of the files, and finally *(V)* compiles the source code into a number of libraries. The modified Andrew benchmark used in this section differs from the original Andrew benchmark in two aspects. First, in the last stage, it compiles automount source code instead of the code included in Andrew benchmark package. Second, instead of logging output into the replicated directory as specified by original Andrew Benchmark, we put the output into a local log file.

Table 6 presents the modified Andrew Benchmark performance in a local area network as the number of replicas increases. The replicas used in this experiment are the same as those in Figure 4. As shown in the table, the penalty for replication in a LAN environment is small.

The servers used in Table 7 experiment are described in Table 3. Not surprisingly, in a WAN, the relative overhead introduced by replication is greatest in stages *(I)* and *(II)*. When the RTT between the primary server and the replication server is 230 ms, the running times in these two stages increase by a factor of 24. Stage *(V)* is also affected by replication support since during the compilation a number of library and executable files are created. Stages *(III)* and *(IV)* measure file and directory read performance. As shown in the table, replication adds no overhead in these two stages.[8]

Table 8 presents the running time for the modified Andrew benchmark with the fixed largest RTT between the primary server and the majority of other replicas, but different replica sets. The servers used in this experiment are described in Table 4. Like Figure 6, this experiment demonstrates that with the fixed RTT between the primary server and the majority replicas, increasing the number of replicas has negligible effect on system performance. For example, when the number of replicas grows from 2 to 7, the total running time for the modified Andrew bench-

[7] This is for ordered writes. For strict consistency, the total response time for a client's update request is dominated by the largest RTT between the primary server and *all* the other replication servers.

[8] The experiment that is missing here is running the modified Andrew benchmark across a WAN without replication. It Will Be Done.

Replicas	I: Mkdir (s)	II: Copy (s)	III: Scandir	IV: Readall (s)	V: Make (s)	Total (s)
P	0.406	3.74	2.50	2.42	45.8	54.9
P + L	0.487	4.62	2.55	2.44	49.4	59.5
P + 2L	0.483	4.44	2.49	2.44	49.1	58.9
P + 3L	0.518	4.55	2.47	2.46	49.1	59.1
P + 4L	0.495	4.53	2.46	2.48	49.7	59.6

Table 6: Modified Andrew Benchmark in LAN replication. This table shows the run times for the modified Andrew Benchmark for LAN replication. All times are in seconds. xL represents the number of local replicas used is x. In this experiment, the RTT between the primary server and each replica is around 260 μs. Except for the primary server, all servers used in this experiment have dual Intel 1.7 GHz CPUs and 2 GB memory.

RTT (ms)	Mkdir (s)	Copy (s)	Scandir (s)	Readall (s)	Make (s)	Total (s)
0	0.406	3.74	2.50	2.42	45.8	54.9
21.8	1.33	12.6	2.51	2.44	83.8	103
47.6	2.39	20.8	2.50	2.46	117	145
66.8	3.14	26.9	2.50	2.48	144	180
119	5.34	48.3	2.49	2.40	231	290
169	7.25	65.6	2.61	2.41	305	383
230	9.69	86.4	2.50	2.43	405	506

Table 7: Modified Andrew Benchmark in WAN replication. This table shows the run times for the modified Andrew Benchmark for WAN replication. In each measurement, the data are replicated on two servers: the primary server and a replication server. The servers used in this experiment are described in Table 3.

Replicas	Mkdir (s)	Copy (s)	Scandir (s)	Readall (s)	Make (s)	Total (s)
P + Wustl	1.33	12.6	2.51	2.44	83.8	103
P + Umich + Wustl + HP	1.41	12.6	2.56	2.49	86.0	105
P + 2Umich + Wustl + 3HP	1.38	13.3	2.56	2.46	86.5	106

Table 8: Modified Andrew Benchmark with varying replica sets. This table shows the run times for the modified Andrew Benchmark with the fixed largest RTT between the primary server and the majority of other replicas, but different replica sets. The servers used in this experiment are described in Table 4.

mark grows by only 3%. The result suggests that if most writes of a replicated file set come from one site, the performance overhead for remote replication can be masked by putting a majority of the replication servers near that area.

6. Related Work

In this section, we survey the research literature related to our work.

NFSv3 [1], the current NFS version, available in all modern operating systems, is the leading choice for distributed file access. In NFSv3, a client expands its name space by mounting remote file systems. A single name space is not supported. NFSv3 does not provide one-copy semantics, but guarantees only time bound data inconsistency. One distinguishing feature of NFSv3 is its stateless servers, which simplifies server implementation and failure recovery but induces a lot of extra communication between server and client. In NFSv3, availability is not a design goal. NFSv3 can support read-only replication.

NFSv4 [3], now under development retains the essential characteristics of the previous versions: design for easy recovery; agnosticism of transport protocol, operating system, and local file systems; simplicity; good performance. In addition, NFSv4 is designed to improve access and performance on the Internet. Quite a few features are introduced in NFSv4: mandatory strong security, compound procedures, the ability for a server to delegate certain responsibilities to a client, as well as the mechanisms that are discussed in this paper: support for a global name space, replication, and migration.

The Andrew File System (AFS) [7, 8, 23] originated at Carnegie Mellon University. In AFS, autonomous administration domains are called *cells*, each with its own servers, clients, system administrators, and users. AFS supports consistent file naming on a global scale through a convention followed by cooperating AFS administrators: each cell's root entry is represented as a mount point in the top level AFS root directory called `/afs`. The location mapping information for these mount points is kept in a file on the local disk of every AFS client in the world. This leads to some scaling problems. As the number of entries in `/afs` increase, search time through the directory also increases. Furthermore, as cells are created and as their servers move, discovery of new AFS cells and their integration into each existing cell's name space becomes cumbersome.

In AFS3, clients cache entire files and directories to improve performance.[9] Servers keep track of which files are in which caches and invoke callback routines to notify clients when cached data has changed. The AFS consistency model guarantees that a client opening a file sees the data stored by the most recent close. This guarantee is hard to honor in a partitioned network. AFS also supports read-only replication.

Coda [19, 20], a cousin of AFS, achieves its primary design goal of constant data availability through server replication and disconnected operation. When a client opens a file for the first time, it contacts all replicas to make sure it will access the latest copy and that all replicas are synchronized. Upon close, updates are propagated to all available replicas. In the presence of failure, Coda sacrifices consistency for availability. When a Coda client is not connected to any servers, users can still operate on files in their cache. The modified files are automatically transferred to a preferred server upon reconnection. This strategy can lead to conflicting updates. In some cases, user involvement is needed to get the desired version of data.

Echo [14] and Harp [10] are file systems that use a primary copy scheme to support mutable replication. In these systems, there is only one primary server for a collection of disks, a potential bottleneck if those disks contain many hot spots. We avoid this problem by allowing dynamic binding of the primary server, which is chosen at the granularity of a single file or directory. Furthermore, by taking advantage of the features provided by the primary-copy method, failure detection and recovery in our system are totally driven by client accesses. This eliminates the need for periodic heartbeat messages or special group communication services.

A lot of work in peer-to-peer (P2P) file systems has been undertaken in recent years, including OceanStore [12], Ivy [13], Pangaea [31] and Farsite [22]. 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; whereas performance or data consistency are often traded off. Compared to these systems, our system addresses data replication among file system servers, which are more reliable but have higher requirements on average I/O performance. This leads to different design strategies in our approach.

[9] In reality, AFS3 caches chunks of files, not whole files, which moots any consideration of consistent shared writes. Like most AFS aficionados, we are in denial about this.

7. Conclusion

This paper presents the design, implementation, and analysis of support for migration, consistent read/write replication, and a global name space for NFSv4. By convention, any file or directory name beginning with `/nfs` is part of a global shared name space. File system migration and replication are supported through DNS resolution. Directory migration and replication use the FS_LOCATIONS attribute to redirect I/O requests. For read/write replication, a novel primary-copy method with server redirection provides concurrency and consistency during replica updates.

This work makes the following contributions. First, the system provides strong consistency guarantees – strict consistency or ordered writes – without penalizing performance for common read operations. Because reads outnumber writes in most applications, good I/O performance is achieved on average. Second, in our system, failure detection and recovery are driven by client accesses, so no heartbeat messages or special group communication services are needed. Third, we developed the system on NFSv4, the Internet standard for distributed filing poised for worldwide deployment.

8. References

1. Sun Microsystems, Inc., "NFS: Network File System Protocol Specification," RFC 1094 (March 1989).
2. Sun Microsystems, Inc., "Design and Implementation of the Sun Network File System," in *Proc USENIX Summer Conf.* (June 1985).
3. Sun Microsystems, Inc., "NFS Version 4 Protocol," RFC 3010 (Dec. 2000).
4. P. Mockapetris, "Domain Names – Concepts and Facilities," RFC 1034 (Nov. 1987).
5. P. Mockapetris, "Domain Names – Implementation and Specification," RFC 1035 (Nov. 1987).
6. ISO, "Open Distributed Processing Reference Model," International Standard ISO/IEC IS 10746 (1995).

7. J. Howard, “An Overview of the Andrew File System,” in *Proc. USENIX Winter Tech. Conf.*, Dallas (Feb. 1988).
8. G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems*, ISBN 0201-61918-0 (2001).
9. F. Cristian, H. Aghali, R. Strong and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", in *Proc. 15th FTCS* (June 1985).
10. B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, “Replication in the Harp File System”, in *Proc. 13th. SOSP*, Pacific Grove (Oct. 1991).
11. A. E1-Abbadi, D. Skeen, and F. Cristian, “An Efficient Fault-tolerant Protocol for Replicated Data Management,” in *Proc. 5th ACM SIGACTSIGMOD Symposium on the Principles of Database Systems*, (1985).
12. S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, “Pond: the OceanStore Prototype,” in *Proc. 2nd USENIX Conf. on File and Storage Tech.* (Mar. 2003).
13. A. Muthitacharoen, R. Morris, T.M. Gil, and B. Chen, “Ivy: A Read/Write Peer-to-peer File System”, in *Proc. 5th OSDI,* Boston (Dec. 2002).
14. 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*, Houston (Nov. 1990).
15. M. Blaze, *Caching in Large-Scale Distributed File Systems*, PhD thesis, Princeton University (Jan. 1993).
16. S.B. Davidson and H. Garcia-Molina, “Consistency in Partitioned Networks,” *ACM Computing Surveys* **17**(31) (1985).
17. M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” *J. ACM* **27** (April 1980).
18. L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” *ACM Trans. on Prog. Lang. and Systems* **4**(3) (July 1982).
19. J. Kistler and M. Satyanarayanan, “Disconnected Operation in the Coda File System,” *ACM Trans. on Comp. Sys.* **10**(1) (February 1992).
20. M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, “Coda: A Highly Available File System for a Distributed Workstation Environment,” in *Proc. IEEE Trans on Comp.* **39**(4) (April 1990).
21. J. Zhang and P. Honeyman, “A Replica Control Protocol for Distributed File Systems”, CITI Tech. Report 04-1 (April 2004).
22. A. Adya, W.J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, R.P. Wattenhofer, “FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment”, in *Proc. 5th OSDI,* Boston (Dec. 2002).
23. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and Performance in a Distributed File System,” *ACM Trans. Comp. Sys.* **6**(1) (Feb. 1988).
24. M. Blaze, “NFS Tracing by Passive Network Monitoring,” in *Proc. Winter USENIX Conf.* (Jan. 1992).
25. M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout, “Measurements of a Distributed File System,” in *Proc. 13th ACM SOSP*, Pacific Grove (Oct. 1991).
26. D. Roselli, J.R. Lorch, and T.E. Anderson, “A Comparison of File System Workloads,” in *Proc. USENIX Ann. Tech. Conf.* (June 2000).
27. M. Satyanarayanan and E. Siegel, “Parallel Communication in a Large Distributed Environment,” *IEEE Trans. on Comp.* **39**(3) (Mar. 1990).
28. R. Sidebotham, “Volumes — the Andrew File System Data Structuring Primitive,” in *Proc. European UNIX System User Group Conf.*, Manchester (September 1986).
29. W.E. Sommerfeld, ‘‘Re: Ethernet Bridge (really: NFS ‘security’),’’Message 1761@bloom-beacon.MIT.EDU, TCP-IP mailing list (Nov., 1987).
30. B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak and M. Bowman, “PlanetLab: An Overlay Testbed for Broad-Coverage Services,” PlanetLab Design Note PDN-03-009, in *Proc. ACM SIGCOMM Comp. Comm. Rev.* **33**(3) (July 2003).
31. Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam, “Taming aggressive replication in the Pangaea wide-area file system,” in Proc. 5th OSDI, Boston (Dec. 2002).
32. A. Gulbrandsen and P. Vixie, “A DNS RR for specifying the location of services (DNS SRV)”, RFC 2052 (Oct. 1996).