This is a draft version of a forthcoming CITI Technical Report
Please do not circulate or cite this paper
Version 10/14/03 3:33:22 PM

Naming, Migration, and Replication in NFSv4

Jiaying Zhang
jiayingz@eecs.umich.edu

Peter Honeyman
honey@citi.umich.edu

Center for Information Technology Integration
University of Michigan
Ann Arbor

Abstract

In this paper, we propose the design of a global name space for NFSv4 and mechanisms for transparent migration and replication. By convention, any file or directory name beginning with /nfs at a NFS client is part of this shared global name space. File system migration and replication are supported through DNS resolution. Directory migration and replication are provided by making use of FS_LOCATIONS attribute. For mutable replication, server redirection is utilized to provide concurency and consistency during replica updates. Strong consistency is guaranteed when the system is free of failures. In case of network partition failures, two kinds of consistency can be provided: one-copy serializability and write serializability, which allow different availabilities.

1. Introduction

The Network File System (NFS) [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, the original goal has been extended.

The current version of NFS, Version 3 (NFSv3), is widely adopted, yet it has some troublesome drawbacks. The NFSv3 protocol was designed for an environment quite different than today's:

- Computers were much less powerful.
- Operating systems were much less reliable.
- Access to remote computers was mostly limited to LANs.
- The wide area network (WAN) known as the Internet was still in its infancy.
- Popular security mechanisms were relatively easy to exploit. For example, the security hole reported by William E. Sommerfeld in 1987 [REF] allowed an impostor to gain unauthorized access to an NFS file system by spoofing file handles.

Today's highly-connected world has led to problems in using NFS over the Internet, where security, performance, and interoperability are critical. To address these difficulties, Version 4 of NFS (NFSv4) [3] was designed. NFSv4 retains the essential characteristics of previous versions of NFS, but is designed to improve access and performance over WANs. To help realize 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.[†]

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 administerability an important goal in file system designs. The past

[†] We discuss transparency in Section 2.1, but for now we use the term loosely.

twenty years has seen numerous research and development efforts intended to facilitate file system administration. For example, in AFS, *volumes* [28] were developed to organize data within a storage complex by breaking the association between physical storage and the unit of migration and replication. Because they are independent of the physical configuration of the system, they provide a degree of transparency in addressing, accessing, and storing files. This also facilitates data movement and optimization of existing storage. One of the outcomes of the volumes abstraction is support for self-healing in storage systems, an important goal in recovery-oriented or autonomic computing [ROC ref, autonomic ref].

The lack of transparent file relocation support makes data movement among different NFS file systems a cumbersome admistration 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.

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.

We also pay special attention to replicating data. There are two primary reasons for replicating data. First, replicated data increase the reliability of a system by allowing users and applications to continue working after one replica crashes by simply switching to another. Second, replication improves performance by allowing access to distributed data from local servers. This is especially significant in allowing a distributed system to scale in size and geographical area.

One of the primary challenges to data replication is consistency across replicas. Although strong consistency is ideal, the tradeoff between performance, availability, and consistency must be taken into consideration during the design of replication systems.

In this paper, we propose a global name space for NFSv4 file system that supports transparent migration and replication. In particular, we adhere to the above principles in our project design. The remainder of this paper is organized as follows, Section 2 introduces background material. Section 3 describes related work. Section 4 presents the design of the system proposed in this paper. Section 5 discusses a prototype implementation. Section 6 evaluates the performance of the prototype implementation. Section 7 discusses the outcome of the project and outlines some issues to explore in the future. Section 8 summarizes and concludes.

2. Background

This section provides a background of Distributed File System design principles, useful for understanding the material presented in later sections.

2.1 Distribution Transparency

Distribution transparency is defined as the abstract concepts and mechanisms that make a distributed system appear as if it were a single united system. Ironically, transparency is really all about opacity, i.e., a system is made transparent by concealing properties derived from separation. There are several forms of distribution transparency [6].

- **Access Transparency:** Hide differences in data representation and how a resource is accessed.
- **Location Transparency:** Hide the location of a resource.
- **Migration Transparency:** Hide that a resource may move to another location.
- **Relocation Transparency:** Hide that a resource may be moved to another location while in use.
- **Replication Transparency:** Hide that a resource is replicated.
- **Concurrency Transparency:** Hide that a resource may be shared by several competitive users.
- **Failure Transparency:** Hide the failure and recovery of a resource.
- **Persistence Transparency:** Hide whether a resource is in memory or on disk.

Although distribution transparency is generally preferable for any distributed system, there is a trade-off between a high degree of transparency and system performance.

2.2 Consistency Models

The main problem introduced by replication is maintaining consistency: whenever a replica is updated, that replica becomes different from the others. To keep replicas consistent, we need to propagate updates in such a way that temporary

inconsistencies are not noticed. However, doing so may severely degrade performance, especially in large-scale distributed systems. The problem is simplified if consistency can be somewhat relaxed.

Most distributed file systems provide one of four consistency models, listed below.

- **One-copy serializability.** Ideally, a file system supporting transactional semantics would implement one-copy serializability, which requires that the execution of operations by distinct clients be equivalent to a joint serial execution of those operations on non-replicated data items shared by the two processes [16]. One-copy serializability guarantees that a client always reads the newest copy in the system.
- **Write serializability.** In many situations, it is important only to guarantee that all write operations are serialized. That is, all writes are executed in the same order everywhere. This guarantees the system will eventually enter a consistent state. In this model, a client is not guaranteed to read the most recent copy.
- **Time bounded inconsistency.** Many applications can tolerate some degree of inconsistency. In a time bounded inconsistency model, a client may access an old version 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 heartbeat messages to check data consistency or to detect network partitions.
- **Optimistic consistency.** Some distributed file systems adopt optimistic schemes to trade consistency for availability. In these systems, any copy can be read or updated at anytime. This is important for applications that wish to have continuous, guaranteed access to data. However, these read-any, write-any schemes can introduce inconsistencies during network partition, so optimistic systems need to provide conflict detection and resolution schemes.

2.3 Failure Models

A system that fails is not adequately providing the services for which it was designed. Several types of failure can occur in distributed file systems.

- **Crash failure.** Crash failure, or fail-stop, occurs when a server prematurely halts but was working correctly until it stopped. An important property of crash failure is that once the server has halted, nothing more is heard from the server.
- **Network partition.** Network partition occurs when there is a network failure that partitions replicas into two or more communicating groups. Communications may still occur within each group, but no communication can be made between groups. Recognizing that partition has occurred can be costly. In addition, it is difficult to tell the difference between a crashed server and a partitioned network.
- **Byzantine failure.** Byzantine failure, also referred to as arbitrary failure or malicious failure, was analyzed in depth by Pease , Shostak, and Lamport [17, 18]. When a server experiences Byzantine failure, it can exhibit any faulty behavior, include arbitrary changes of state. Moreover, a faulty server may even be working maliciously in concert with other servers to produce wrong or misleading answers. This situation illustrates why security is considered an important requirement in distributed systems.

In distributed systems, redundancy is the key technique for masking failure. As we stated in Section 1, replication is often used to enhance reliability in distributed file systems.

3. Related Work

In this section, we describe the research literature related to our work. First, three popular distributed file systems are introduced: NFSv3, AFS and Coda. Following that, other related work is summarized.

3.1 NFSv3

NFSv3 [1], the current NFS version, has been adopted by many users for distributed file access. In NFSv3, each client expands its name space by mounting remote file systems. Access transparency and location transparency are then provided. However, a single name space is not supported. NFSv3 does not provide one-copy semantics, but guarantees only time bounded data consistency. One distinguishing feature of NFSv3 is that servers are stateless. The main advantage is simplicity in server implementation and failure recovery. The disadvantage is that more communication messages are needed between server and client. In NFSv3, availability is not a design goal. NFSv3 supports read-only replication.

3.2 AFS

The Andrew File System (AFS) [7, 8, 23] originated at Carnegie Mellon University. In AFS, multiple administration domains are defined as *cells*, each

with its own servers, clients, system administrators, and users. AFS supports consistent file naming on a global scale. Each cell's file space entry is represented as a mount point object in the top level AFS root directory.

AFS clients cache entire files and directories for better performance. Servers record the files clients are caching, then execute callback routines to notify clients when cached data has changed. This strategy eliminates superfluous network messages, but complicates server recovery in the case of failure. Like UNIX, the AFS consistency model provides that the "last writer wins" and guarantees that a client opening a file sees the data stored when the most recent writer closed the file. This guarantee is hard to honor in a partitioned network. AFS also supports read-only replication.

3.3 Coda

Coda [19, 20], a cousin of AFS, achieves its primary design goal of constant data availability through server replication and disconnected operation. A volume is replicated at a set of servers. 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, upgrades are propagated to all available replicas.

In the absence of failures, the consistency model of Coda is identical to that of AFS. In the presence of failures, 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.

The Coda group has investigated automated file and directory conflict detection and resolution mechanisms [21, 22]. However, not all conflicts can be resolved; in some cases, user involvement is needed to get the desired version of data. The requirement for clients to check each replica at the time of cache misses introduces latencies. The requirement to hoard files on the local machine to make them available while off-line makes Coda impractical to use on diskless machines.

3.4 Other Distributed File Systems

WebFS. WebFS [9] is a kernel-resident file system that provides access to the global HTTP name space. Upon mounting WebFS, the root directory is defined to contain all HTTP/WebFS sites that the file system is initially aware of. Thus, the root directory is initially empty. Directory entries are created on reference. The first time an application attempts to access an HTTP or WebFS site, the system checks for the presence of (first) a WebFS server, or (second) an HTTP server. If either server exists, a directory entry is created in the root WebFS directory. The name of the directory is set to the hostname of the remote server. Currently, no hierarchy is provided in WebFS; all HTTP/WebFS sites are under the flat name space of the root directory. For general file sharing, WebFS provides "last writer wins" consistency, similar to AFS.

UFO. UFO [10] employs the UNIX tracing facility to intercept system calls and transfers whole files from FTP and HTTP servers. A global file system is implemented to allow applications transparent access to files on remote machines. Names of remote files can be specified through a URL; through a regular file name implicitly containing the remote host, user name, and access mode; or through mount points specified explicitly in a `.uforc` file. UFO provides read and write access to FTP servers and read-only access to HTTP servers. UFO stores modified files on the server when they are closed.

Ficus. Ficus [11] is a peer-to-peer file replication system. It uses a single-copy availability update policy with a reconciliation algorithm to detect concurrent updates and restore the consistency of replicas. In Ficus, replica location information is stored at each replica. A bit mask is used to indicate whether a replica holds the replica location information. Updates to replica location information are allowed as long as any such replica is available. A special algorithm called laissez faire is used to propagate updates. Periodic reconciliation is used to maintain consistency.

Global Computing. Projects such as Legion [12] and Globus [13] aim to provide an infrastructure for global computation, i.e. a world-wide supercomputer. NFSv4 would be a candidate data access mechanism to such projects in the sense that it can provide the global name space, security, consistency, and file system semantics necessary to support such global applications.

4. Design

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

- **A single global name space:** One of the requirements in a wide area environment is a single name space for all files in the system. A global name space encourages collaborative work and dissemination of information, as

everyone has a common frame of reference. Users on any NFS client, anywhere in the world, should be able to use an identical file name to refer to a given object.

- **Transparency:** Transparency is an important goal of distributed systems. As stated in Section 2.1, there are different forms of transparency. In this paper, all forms of transparency except access transparency and persistence transparency are discussed. An implicit requirement of location transparency is location independent naming. Resources are accessed through logical names, instead of physical locations. Location independent naming can also provide transparent migration and replication. To support relocation transparency, a distributed system should be able to redirect connected clients to the new resources transparently. Mutable replication and transparent concurrency requires a distributed file system to provide consistency among replicas. A distributed system should implement its consistency model in such a way that also balances performance and availability. Finally, to provide failure transparency, automatic failure detection and recovery mechanisms are needed.
- **Performance.** Good system performance is a critical goal in this project. In particular, we aim to make common accesses fast. Previous research [15, 24–26] has investigated work load patterns in real file systems that give us insight in our design. Ordered by expected frequency, the following cases are considered.

  Exclusive read: most often. Support for replication should not add 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. Consequently, a distributed system should provide performance for shared read accesses similar to that of the single reader case.

  Exclusive write: less common. Previous work in file system tracing shows that writes are less common than reads in file system workloads. 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 that 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 can even be blocked to provide strong consistency without doing much damage to overall throughput averages.

- **Scalability:** Successful distributed systems tend to grow in size. To provide scalability, two principles are followed in our design. First, centralized services are avoided. Second, less functionality is put on server side.

This project does not consider Byzantine failures: a server is assumed to be good as long as it can authenticate itself. Furthermore, servers are assumed to be in good state most of the time. Dramatically changed server states and network conditions are not the focus of our design.

4.1 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 defined as the global root of all NFSv4 file systems. To an NFSv4 client, `/nfs` appears as a special directory, which 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 NFS 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 [REF] conventions. We use a TXT Resource Record (RR) [4, 5] for server location information. 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 needs only to update its DNS server to add a mapping from the file system reference name to the new NFS server location. Similarly, when a file system is migrated to

another NFS server, the old mapping is updated to point to the new server. In addition to file system locations, other information such as mount options can also be carried in DNS RRs.

4.2 File System Name Space and Directory Migration & Replication

The file system name space provided by an NFS 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 `/`. In this way, 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. This feature is an essential element for support of a global name space, and reflects the intention of the NFSv4 protocol designers to provide support for it.

Directory migration and replication among different NFSv4 file systems is implemented by exporting a directory with an attached reference string on servers serving that directory. A reference string includes information on how to get directory replica locations. Four types of reference string are implemented in our prototype: *LDAP, DNS, FILE* and *SERVER REDIRECT.*

- **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 way DNS carries replica location information is the same as described in Section 4.1.
- **FILE:** the format of FILE reference string is `file://pathname/lookup-key`. The pathname gives the path to a file that contains the stores a lookup-key for replica location mappings. The file should be stored in a place accessible to NFS clients, e.g., the parent of the replicated directory.
- **SERVER REDIRECT:** SERVER REDIRECT lookup method is used to support directory migration and relocation. As we will see in Section 4.4, it is also used to support concurrent accesses in mutable replications. The format of a SERVER REDIRECT reference string is `server://hostname:/path [mount-options]`, where hostname:/path gives the location of the replicated directory.

When a client first accesses a replicated directory, the reference string of the directory is sent to the client. With the reference string, the client can query the replica locations of that directory, select a nearby replica and continue its access.

Support for multiple lookup methods allows organizations to maintain replica location information as they desire. No centralized name service is required. Replica location querying and selection are left to clients. These choices promote scalability.

4.3 FS_LOCATIONS Attribute

FS_LOCATIONS is a recommended attribute in the NFSv4 protocol. According to the protocol interpretation, the FS_LOCATIONS attribute is intended to support file system migration and read-only replication. In the case of migration, an attempted client access might yield an FS_LOCATIONS attribute that gives a new location. For replication, a client's first access of 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. In this project, we also use FS_LOCATIONS attribute to communicate migration and replication location information between servers and clients. However, the way we use the attribute is different from that stated in the 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 lookup replica locations with reference strings. Second, we use the FS_LOCATIONS attribute to provide directory migration and replication, instead of the coarse-grained file system migration and replication. Finally, we use the FS_LOCATIONS attribute to support concurrent accesses in mutable replications, which we describe in detail in the next section.

4.4 Mutable Replication

The scheme described in the last two subsections can efficiently support read-only replication. However, to support mutable replication, we also need a mechanism to distribute updates whenever a replica is modified. As analyzed in Section 2.2, a distributed

file system also needs to guarantee some kind of consistency during updates. We considered several mechanisms during our design.

- **Reader checks.** One strategy is for a reader to check all replicas on every read request to be sure it sees the most up-to-date data. This solution can guarantee one-copy serializability, but adds substantial overhead to normal read requests.
- **Replica heartbeat:** By having replicas send periodic heartbeat messages, the system allows a narrow window of inconsistency but can guarantee data consistency after the defined time bound. With this method, the system can automatically discover and recover from failure. However, heartbeat messages add undesirable overhead and network traffic to the system even when all replica servers are working properly.
- **Server redirection.** The strategy that we have adopted provides one-copy serializability at little cost to exclusive or shared readers. When a client opens a file for writing, the selected server disables replication by contacting all other servers replicating the file and instructing them to redirect all accesses to itself. We support two consistency models that differ only in the event of the failure of a replication server. We offer strict, one-copy serializability at the cost of blocking all access while the failed server is under repair. By relaxing the consistency model to write serializabilty, we can allow continued operation in the majority partition. Details are presented in the following subsections.

4.4.1 File Updates

We accomplish consistent replication by redirecting all accesses to a single server when a file is opened for write. When a client opens a file for writing, the relevant server instructs all other replication servers for that file to redirect subsequent accesses to itself. When the file is closed, the updated file is propagated to the redirected servers and file replication is re-enabled.

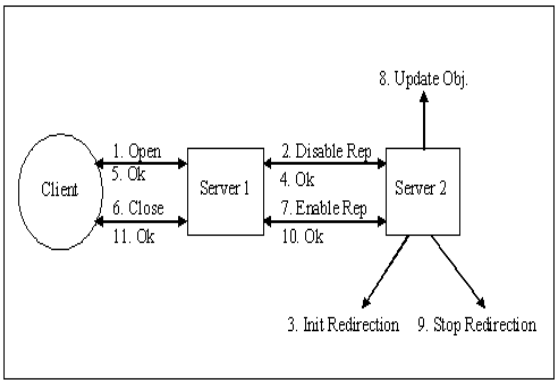

Figure 1: File modification. 1. A client issues an open request to a replication server. 2. This server instructs other replication servers to redirect requests to itself, making it the primary server for the file. 3. Replication servers comply. 4. Replication servers acknowledge the request. 5. The primary server acknowledges the open request. 6. The client issues a close request. 7. The primary server instructs the redirected servers to re-enable replication. 8. The redirected servers obtain an update from the primary server. 9. The redirected servers disable redirection. 10. The (formerly) redirected servers acknowledge the request to re-enable replication. 11. The close request is acknowledged.

While the file is open for writing, the server issuing replication disabling messages behaves as a primary server. Client access requests sent to servers are redirected to the primary server. As mentioned in Section 4.2, a special type of reference string, Server Redirect, is utilized for server redirection. The procedure is illustrated in Figure 1. Here we assume files are opened in asynchronous mode. We do not support files opened for synchronous access; we discuss this case in Section 8.

4.4.2 Directory Updates

Directory modifications include the creation and removal of entries in a directory, entry renaming, etc. Unlike file writes, directory updates do not involve much elapsed time between start and finish, and clients tend to expect transactional behavior. Therefore, we decide not to use server redirection to support concurrent directory access while an update is in progress. Instead, when a server receives a directory update request from a client, the server disables replication of the directory at other servers, serves the request, and then re-enables replication. If another replica receives an access request for the directory being updated, that request is blocked until the directory replication is re-enabled. The directory modification procedure is shown in Figure 2.

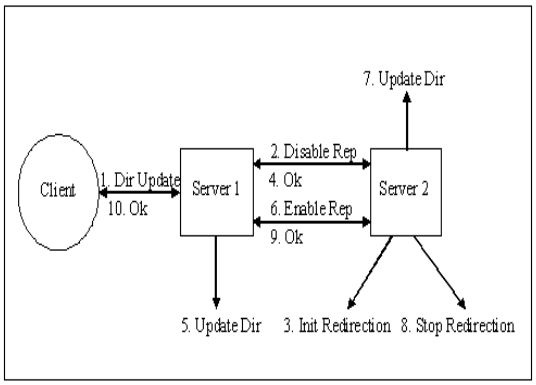

Figure 2: Directory modification. 1. A client issues a directory update request to a replication server. 2. This server instructs other replication servers to block any access to this directory. 3. Replication servers comply. 4.. Replication servers acknowledge the request. 5. The primary server processes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

4.4.3 Special cases

Conflict. Two or more servers may try to disable replication of a file or directory at the same time. The result is that some replicas are disabled by one server, while some are disabled by other servers. In the absence of failure or partition, conflict 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. It is easy to see that this approach converges. However, this simple strategy may have drawbacks when used in a WAN, so we plan to explore other leader election schemes in future work.

Failure. We can guarantee one-copy serializability when all replicas are in working order. However, failure complicates matters. A server might be unable to ensure redirection 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 can no longer response to any clients, but a partitioned server can still serve client requests that originate in the 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.

The first option provides write serializability. When failure occurs, an automatic failure detection and resolution program (FDR) is engaged. If the primary server crashes while serving a writing client, the client notice the server failure and reports it to FDR. FDR determines the state of all replication servers. If replication is disabled everywhere, FDR re-enables replication in available replicas and informs the client of the request failure. In this case, the client must reconnect to another replica and reissue the failed operation. If replication is enabled on some servers, FDR will select an enabled replica as the primary server, redirect the client to the new server and inform other replicas to synchronize with the selected server. If some other server fails, the primary server will detect the failure and report it to FDR. That replica will be removed from the replication set and the system can proceed.

For one-copy serializability, we use a different strategy. If the primary server crashes while serving a writing client, the procedure is as described above: verify that replication is disabled everywhere, find a new primary, and inform the client. similar to the first option. However, if any other replication server fails, all replication servers enter a read-only state. The system returns to normal state after the failed server is repaired or with administrator involvement.

5. Implementation

In this section, we report on a prototype implementation of the design described in Section 4. First we a describe a modified Automount daemon, used to automatically mount and unmount NFSv4 servers. Then we discuss our implementation of replication support for NFSv4. We used Linux 2.5.68 throughout.

5.1 Automount

Automount and AutoFs are tools that allow users of one machine to mount different file systems automatically at the moment they are needed. Automount, often referred as AMD, is a user level 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. AutoFs monitors attempts to access a subdirectory within a designated directory and notifies the AMD daemon to perform mounts or unmounts there. Upon receiving an mount request from the kernel, the daemon uses the 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 specified amount of time, AutoFs instructs the daemon to unmount it.

Although Automount supports numerous mapping methods, no support for DNS mapping, required for

this project, is 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 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 the NFSv4 client receives a reference string from the 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 replica.

5.2 Exportfs

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 notifes the client and sends the reference string in FS_LOCATIONS attribute. A new option is also provided in the exportfs interface, allowing users to set required consistency for exported data.

5.3 Mutable Replication Implementation

In mutable replication, a server needs to know the replica list before it issues a replication disabling procedure. Our implementation maintains this information dynamically, using 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 up-call is made to a user-level handler, which performs the lookup and adds the queried data to the cache.

We use RSYNC to synchronize replicas with the primary server during update propagation. RSYNC is an open source utility that provides fast incremental file transfers. It uses the "rsync algorithm", which allows RSYNC to transfer just the differences between two sets of files across the network. Tridgell [14] gives details about the update algorithm.

In our system, when a replica receives a replication enabling message from the primary server, an up-call request is send to a user-level daemon, which executes RSYNC to update the object. The advantage of using RSYNC is the simplicity of implementation. The disadvantage is the performance cost introduced. This problem is analyzed in detail later.

RPC procedure calls are used to transfer replication disabling and enabling messages among replicas. In our prototype implementation, these messages are sent serially by the primary server.

6. Evaluation

Having describing the system architecture and implementation, we now present performance data collected with a prototype implementation. Our testing environment is extremely simple, and perhaps a bit naive. All measurements were made in a local network. Replication performance is tested with three replicas. The performance investigation in wide area environment and with more replicas is left for future work.

In this paper, all data were obtained by using an AMD-K6 400MHz with 128MB of memory as client, and an IBM NetVista with one Intel 1.8GHz Pentium4 processor and 128MB memory as the primary server. One replica used in the testing, referred to as Big Replica below, is a Dell PowerEdge2650, with two Intel 1.8GHz Xeon processors and 1G of memory. Another replica is a PowerEdge2650 with one Intel 1.8GHz Xeon and 512MB memory, referred to as Small Replica. The client and the primary server are connected through a Gigabit router and a 100Mbps switch. The three servers are connected by a 100Mbps switch. All machines run Linux 2.5.68. The data were measured by running gettimeofday, which has microsecond resolution in Linux 2.5.68.

6.2 Results

Table 1 lists the NULL Remote Procedure Call (NULL RPC) response times related to the evaluations presented in this part. These are useful in helping to assess subsequent performance measurements.

NULL RPC	Time (ms)
Client → Primary Server	482 (0.008)
Primary Server → Big Replica	264 (0.009)
Primary Server → Small Replica	261 (0.01)

Table 1: NULL RPC response time. The client has a 400MHz AMD-K6 processor, with 128MB memory. The Primary Server is IBM NetVista with one Intel 1.8G P4 processor and 128MB memory. The Big Replica is a Dell PowerEdge2650 with two Intel 1.8G XEON processors and 1.0G memory. The Small Replica is a Dell PowerEdge2650 with one Intel 1.8GHz XEON and 512MB memory. All machines are within a local network. The client and the primary server are connected through a Gigabit router and a

100Mbps switch. The three servers are connected by a 100Mbps switch. The numbers presented here are mean values from five trials of each experiment. Figures in parentheses are standard deviations.

Table 2 presents the different query times used to lookup replica locations from clients. As shown in the table, 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 only around 1.5 ms query time. LDAP and FILE are a little slower with 12 ms and 13ms query time respectively, which are still acceptable since a client needs to query replica locations only on its first reference.

Query Method	Time (ms)
DNS TXT RR	1.49 (0.006)
LDAP	12.3 (0.8)
FILE	13.1 (0.7)
SERVER REDIRECT	0.007 (0.001)

Table 2: Lookup time for different querying methods. This table shows the lookup time of different querying methods at the client side. The numbers presented here are mean values from five trials of each experiment. Figures in parentheses are standard deviations. 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.

Table 3 demonstrates the time spent in each phase when a client first accesses an NFSv4 file system in the provided global name space. As shown in the table, the mount phase takes the most time. This is not surprising as client and server need to mutually authenticate during mount. The total response time seen by the client is approximately 0.04 second. 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. Although the response time will increase in a WAN, we judge this overhead to be acceptable as a client experiences such delay only at the first time of reference.

Phase	Time (ms)
Upcall	1.28 (0. 06)
Replica List Query (DNS)	1.49 (0.006)
Mount	37.5 (0.4)
Total	40.2 (0.4)

Table 3: First access. This table shows the time spend at the client side when it first accesses a file system within the provided global name space. The numbers presented here are mean values from five trials of each experiment. Figures in parentheses are standard deviations. These measurements can also 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

Table 4 compares the response time for normal open and that for redirected open. As shown, redirected open is more than ten times slower than normal open. However, measurements and simulations in [15] [24] and [25] show that files are rarely write-shared in real workload, so that redirected open will not occur often.

Replica list query	Time (ms)
Normal Open	5.41 (0.06)
Redirected Open	62.7 (0.7)

Table 4: Normal open and redirected open time. This table shows the response time of normal open and redirected open seen by the client. The redirected open refers to the situation when the client wants to access a file being written at another server (primary server). In this case, the client is notified to connect to the primary server for file accessing. The numbers presented here are mean values from five trials of each experiment. Figures in parentheses are standard deviations.

The replica list query time spent at server side is listed in Table 5. This proves that caching can appreciably reduce query time

Replica list query	Time (ms)
Cache miss	14.1 (1)
Cache hit	0.004 (0.001)

Table 5: Replica list query time at server side. This table shows the different replica list query time in cache miss and cache hit at the server side. It is only used by mutable replicas. The querying method used in the testing is DNS TXT RR. The numbers presented here are mean values from five trials of each experiment. Figures in parentheses are standard deviations. The query method used in this testing is DNS RR.

Tables 6 and 7 present the response time of file open for writing and file close after writing with different replica sets. When opening a file, disabling replication introduces a small overhead. Much more delay is added in file close after writing. Table 7 shows that the update distribution phase takes the most time in close procedure.

Replicas	Time for each phase (seconds)	
	Disable Replication	Client Response Time
(P)	0	0.00349 (0.04)
(P, B)	0.000421 (0.000006)	0.00392 (0.008)
(P, S)	0.000413 (0.000005)	0.00387 (0.005)

(P, B, S)	0.000503 (0.00005)	0.00398 (0.05)

Table 6: File open for writing. **This table shows the times spend in each phase during file open for writing operations with different replica sets. The numbers presented here are mean values from five trials of each experiment. Figures in parentheses are standard deviations.**

Replicas	Time for each phase (seconds)		
	Update Distribution	Enable Replication	Client Response Time
(P)	0	0	0.00413 (0.00001)
(P, F)	0.463 (0.009)	0.000372 (0.000002)	0.467 (0.01)
(P, S)	0.644 (0.02)	0.000386 (0.000009)	0.649 (0.03)
(P, F, S)	0.687 (0.04)	0.000461 (0.00004)	0.692 (0.04)

Table 7: File close after writing. This table shows the response times of file close after writing operations with different replica sets. The data were collected by appending 10 bytes to a 1M size file. Update distribution phase refers to the stage during which each replica pulls update from the primary server. This is implemented in the prototype by running rsync with ssh as the remote shell program.

Table 8 shows the time spent in each phase during three common directory updates. Similar to file close after writing, most of the overhead is added by update distribution. We expect to reduce this overhead by investigating more efficient update distribution methods. We discuss this further in the next section.

7. Discussion

Synchronous writes. Earlier, we assumed clients write files asynchronously. This assumption is violated if an application opens a file in synchronous mode. In this case, if we use the same strategy as described in Section 4.2, and if the primary server fails before it distributes updates to other replicas, it is difficult for the client to redo write operations it already made at the failed server, because in synchronous mode the client is assured that the modified data has been stored in the system reliably after a write returns. But we may use another strategy, in which the primary server distributes updates each time it receives a commit request from the synchronous client. This will increase the response time for write operations, which is the cost paid for synchronous writing.

Update distribution. As shown in Tables 7 and 8, update distribution takes the most time during file close after writing and directory updates. Although it is not surprising that update distribution introduces additional delay to the total response time, we should expect a smaller overhead in this phase. We speculated the high overhead seen in update distribution is due to the use of RSYNC. To gauge this, we measured the time spent by RSYNC when synchronizing identical directories and files. In this case, no updates need to be made at the replicas. The collected data are shown in Table 9.

Synchronization Unit	Replicas	Time (seconds)
Directory	F → P	0.270 (0.001)
	S → P	0.444 (0.003)
File	F → P	0.245 (0.02)
	S → P	0.422 (0.02)

Table 9: Time used by rsync to synchronize the same two objects. This table shows the times spend by rsync when synchronizing the same two directories and/or files. In these cases, no updates are made at each replica. The directory used in this testing is the same as that used in Table 8, which includes four files and three directories.

Operations	Replicas	Time for each phase (seconds)			
		Disable Replication	Update Distribution	Enable Replication	Client Response Time
File Rename	(P)	0	0	0	0.00813 (0.0002)
	(P, F)	0.000413 (0.00002)	0.321 (0.02)	0.000366 (0.00002)	0.330 (0.02)
	(P, S)	0.000417 (0.00001)	0.500 (0.02)	0.000372 (0.00001)	0.510 (0.02)
	(P, F, S)	0.000494 (0.00003)	0.556 (0.02)	0.000579 (0.00002)	0.566 (0.02)
MakeDir	(P)	0	0	0	0.00852 (0.0001)
	(P,F)	0.000413 (0.00001)	0.288 (0.02)	0.000376 (0.00001)	0.298 (0.02)
	(P, S)	0.000402 (0.00003)	0.459 (0.02)	0.000374 (0.00001)	0.474 (0.03)
	(P, F, S)	0.000474 (0.00004)	0.492 (0.04)	0.000442 (0.00001)	0.505 (0.04)
RemoveDir	(P)	0	0	0	0.00750 (0.0003)
	(P,F)	0.000384 (0.00002)	0.280 (0.02)	0.000397 (0.00003)	0.296 (0.02)
	(P, S)	0.000393 (0.00002)	0.451 (0.02)	0.000374 (0.00001)	0.462 (0.02)
	(P, F, S)	0.000477 (0.00004)	0.478 (0.02)	0.000456 (0.00003)	0.492 (0.02)

Table 8: Three common directory updates. This table shows three common directory update operations: File Renaming, MakeDir and RemoveDir, with different replica sets. Data shown in File Renaming operation were collected by renaming a 1K size file within a directory. Update distribution phase refers to the stage during which each normal replica pulls update from the primary server. This is implemented in the prototype by running RSYNC with SSH as the remote shell program.

As shown in Table 9, even when no updates are made at each replica, a long delay is still seen. This includes the time spent for SSH authentication and the cost to run the RSYNC algorithm. We also note that the figures shown in Table 9 for directory synchronization are very close to the update distribution times shown in Table 8. This suggests the real time needed by update distribution may be considerably less. A bigger difference is seen for file synchronization. We speculate that this is caused by the file update strategy used in RSYNC. That is, when a file needs to be updated, RSYNC reconstructs the file instead of making modifications directly to the original file. Thus when small modifications are made to a large file, as the case shown in our experiments, extra delay is introduced.

The above analysis indicates the need for a better update distribution mechanism in the system implementation. RSYNC was not designed for operations as used in our prototype; we employed it as the update distribution mechanism in the prototype as an expedient, simplifying our implementation (and saving the time and effort of implementing our own update mechanism). In the future, we plan to explore other mechanisms to distribute updates. For example, having the primary server propagate updates with a parallel RPC [27] is an attractive option.

Tables 7 and 8 also show that update speed is limited by the slowest replica in the system. Here, the slowest replica refers to the replica that takes the longest time to get updates from the primary server. This results from our design decision that a primary server has to wait until it gets replies from all available replicas in the system. An alternative strategy that can help reduce total response time is to further relax the consistency model and allow a primary server to reply to its client after a defined number of replicas have finished the update. We plan to explore this idea further in our future work.

Performance testing. System performance in a wide area environment and with more replicas will be measured in the future. We are investigating using network emulation tools, such as NistNet, to simulate wide area network conditions.

Client caching. Clients can cache reference strings to replica location mappings to reduce replica location query time. We left this implementation for future work.

Replica selection. Replicas can be selected based on several criteria. For example, servers on the same subnet can be given the strongest preference, with servers on the local net given the second strongest preference. Among servers equally far away, response times can be used to determine a preference. Replica selection can be also policy based. Additionally, the LDAP server holding a replica list can monitor requests to that replica list and give a recommendation list for load balance purpose.

Automatic failure detection and resolution. As described in Section 4.3.3, an automatic failure detection and resolution program is needed when failure occurs. We left its detailed design and implementation to future work.

8. Conclusion

This paper presents the design and implementation of support for migration, mutable consistent replication, and a global name space for NFSv4. By convention, any file or directory name beginning with `/nfs` is part of this name space. File system migration and replication are supported through DNS resolution. Directory migration and replication are provided by making use of FS_LOCATIONS attribute. For mutable replication, server redirection is used to provide concurency and consistency during replica updates. Strong consistency is guaranteed when the system is free of failures. In the case of network partition, two kinds of consistency can be provided: one-copy serializability and write serializability, which allow different availabilities.

9. Reference

[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 *USENIX Summer Conference Proceedings*, 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 *Proceedings of the USENIX Winter Technical Conference*, Dallas, TX, February 1988.

[8] Coulouris, G., Dollimore, J., Kindberg, T, Distributed Systems, 2001, ISBN 0201-61918-0

[9] A. M. Vahdat, P. C. Eastham, and T. E. Anderson. Webfs: A global cache coherent file system. Technical report, University of California, Berkeley, 1996.

[10] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: the ufo global file system. In *1997 Annual Technical Conference on Unix and Advanced Computing Systems (USENIX '97)*, January 1997.

[11] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63-71, Anaheim, CA, June 1990.

[12] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf. Campus-Wide Computing: Results Using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, March 1995.

[13] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *Proc. Workshop on Environments and Tools*, 1996.

[14] A. Tridgell. *Efficient algorithms for sorting and synchronization*. PhD thesis, The Australian National University, 1999.

[15] M. Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[16] Susan B. Davidson, Hector Garcia-Molina: "*Consistency in Partitioned Networks*", Computing Surveys 17(3), pp. 341–370, 1985.

[17] M. Pease, R. Shostak, and L. Lamport, "*Reaching agreement in the presence of faults*," Journal of the ACM, vol. 27, pp. 228–234, Apr. 1980.

[18] L. Lamport, R. Shostak and M. Pease. The Byzantine Generals Problem. ACM Trans. on Prog. Lang. and Systems 4, 3 (July 1982), 382-401.

[19] J.J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[20] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers 39*, 4 (April 1990).

[21] Kumar, P., and Satyanarayanan, M. Supporting Application-Specific Resolution in an Optimistically Replicated File System. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems* (Napa, CA, October 1993).

[22] Kumar, P., and Satyanarayanan, M. Log-based Directory Resolution in the Coda File System. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems* (San Diego, CA, January 1993).

[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 Transactions on Computer Systems, 6(1):51--81,* February 1988.

[24] Matt Blaze. NFS Tracing by Passive Network Monitoring. In the Proceedings of the Winter USENIX Conference, pages 333--343. USENIX Association, Jan 1992.

[25] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, John K. Ousterhout, Measurements of a distributed file system, Proceedings of the thirteenth ACM symposium on Operating systems principles, p.198-212, October 13-16, 1991, Pacific Grove, California, United States.

[26] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. USENIX Annual Technical Conference (San Diego, CA, 18--23 June 2000.

[27] Satyanarayanan, M., Siegel, E.H. Parallel communication in a large distributed environment. *IEEE Transactions on Compute.* Mar. 1990, Vol. 39, No. 3.

[28] R. Sidebotham. *Volumes –- the Andrew file system data structuring primitive.* In European UNIX System User Group Autumn '86 Conference Proceedings, pages 473–80, Manchester, UK, September 1986