Naming, Migration, and Replication for NFSv4

Jiaying Zhang
jiayingz@umich.edu

Peter Honeyman
honey@citi.umich.edu

Center for Information Technology Integration
University of Michigan at Ann Arbor

Abstract

In this paper, we discuss a global name space for NFSv4 and mechanisms for transparent migration and replication. By convention, any file or directory name beginning with /nfs on an NFS client is part of this shared global name space. Our system supports file system migration and replication through DNS resolution, provides directory migration and replication using built-in NFSv4 mechanisms, and supports read/write replication with precise consistency guarantees, small performance penalty, and good scaling. We implement these features with small extensions to the published NFSv4 protocol, and demonstrate a practical way to enhance network transparency and administerability of NFSv4 in wide area networks.

1. Introduction

To meet the distributed filing needs of the world-wide Internet, the NFSv4 protocol [27] is engineered to provide good scaling and consistent sharing. Our goal in this work is to build on that foundation to increase network transparency and to simplify administration.

Network transparency is the collection of the abstract concepts and mechanisms that make a distributed system appear as if it were a single united system. In practice, network transparency is really all about opacity, i.e., a system is made transparent by concealing properties derived from separation. In a distributed file system, network transparency involves three aspects: naming, performance, and failure resiliency.

Naming plays an important role in distributed file systems. A file system is said to provide name transparency if it satisfies the following three requirements. First, any file is accessible from any location. Second, the same name is used at every location. And third, the file location is not reflected in the name.

To achieve name transparency, we develop a naming scheme that allows organizations to export their data autonomously in a single shared global name space. Users on any NFSv4 client, anywhere in the world, can then use an identical rooted path name to access a file or directory. Furthermore, by allowing a NFSv4 file system to be accessed with a logical name, instead of its physical location, we support location independent naming, which facilitates file system migration and replication.

Good performance is always a critical goal. A distributed file system provides performance transparency if it hides access latency for remote resources.

In a distributed file system, a client's access can be interrupted by any number of machine or network failures. Failure transparency requires that the system hides a failure and recovery of resources.

Distributed systems often use replication to improve performance transparency and failure transparency. There are two primary reasons for replicating data. First, replication improves performance by allowing access to distributed data from nearby or lightly loaded servers. Second, replicating data improves availability in the face of failure by allowing users and applications to switch from a failed replication server to a working one.

In distributed file systems, 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 observable. However, doing so may severely degrade performance, especially in large-scale distributed systems.

In this paper, we discuss a read/write (or mutable) replication protocol for NFSv4 that balances the tradeoff among consistency, performance, and availability by offering stringent yet flexible consistency guarantees to applications. The protocol can guarantee either ordered writes or synchronous access, while imposing no performance overhead for normal reads. It can tolerate any number of server crash or link failures, even when these lead to network partitioning. The protocol uses standard POSIX features, which makes it easy and practical to deploy, and opens the door to standardize the extensions to NFSv4 protocol.

Our work is also motivated by the need for easy administration of NFSv4 systems. As storage systems become larger and more complex, storage management plays a prominent and increasing role in system administration. This makes administerability an important goal in file system design. The past twenty years has seen numerous research and development efforts intended to facilitate file system administration. E.g., in AFS, volumes [34] were developed to organize data within a storage complex, which breaks the association between physical storage and the unit of migration and replication. Independent of the physical configuration of the system, volumes provide a degree of transparency in addressing, accessing, and storing files. The volumes abstraction also facilitates data movement and optimization of existing storage.

In NFSv3 and v2, the lack of transparent file relocation support makes data movement among different NFS systems a cumbersome administration task, often disrupting 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 to facilitate administration of large scale distributed file systems.

The remainder of this paper is organized as follows: Section 2 reviews the background and related work. Section 3 describes a naming scheme that supports a global name space and transparent replication and migration. Section 4 presents a mutable replication protocol that guarantees ordered writes and synchronized access. Section 5 evaluates the performance with a prototype implementation, and Section 6 concludes.

2. Background and Related Work

The first popular distributed file system, Sun Microsystem's Network File System (**NFS**) [1, 30] was announced in 1984. NFS was originally developed by Sun for use on its UNIX-based workstations, but NFS designers paid special attentions to portability and heterogeneity. NFS employs a client/server architecture. The NFS protocol defines an RPC interface that allows servers and clients to communicate over the network. The protocol does not specify how servers or clients should implement this interface. As a result, NFS can run easily on a heterogeneous collection of computers.

Version 3 of NFS (NFSv3) [2, 26], the widely adopted current version, was released in 1994. At that time, computers were less powerful than today's and networks were more commonly local area networks rather than wide area networks. This has led to problems in using NFS in today's network environment. So a new version of NFS, NFSv4 [33, 27], is being developed. Like previous versions of NFS, NFSv4 has a straightforward design, simplified error recovery, and independence of transport protocols and operating systems for file access in heterogeneous networks. NFSv4 also introduces some new features intended to improve Internet access and performance, such as the introduction of compound RPC that groups multiple related operations into a single RPC packet, the delegation capabilities to enhance client performance for narrow data sharing applications, the integration of file locking that can support different operating system semantics and error recovery, the mandatory strong security via an extensible authentication architecture built on GSS-API [10], and the facilities to support file system migration and read-only replication. However, the published NFSv4 protocol does not provide mechanisms to support a global name space, transparent file system replication and migration, and mutable replication. These oversights impact the network transparency and administerability that NFSv4 file systems provide.

The absence of a global name space hampers collaborative work and sharing of data because users lack a common frame of reference. No support for transparent file system replication and migration embarrasses data distribution and management. And the limit of read-only replication sacrifices network transparency for write operations. To overcome these deficiencies, we develop a naming and replication scheme that provides the above missing features, as described detailedly in the subsequent sections.

Another widely used distributed file system, the Andrew File System (**AFS**), originated at Carnegie Mellon University in 1983 [31, 24]. The principle goal of AFS is to present a homogeneous, location-transparent name space to personal and time-sharing computers on a campus-wide network, but AFS also pays special attention to scalability, administerability and availability.

AFS clients cache files and directories aggressively on local disk. Servers record the files clients are caching, then execute callback routines to notify clients when cached data has changed. The AFS consistency guarantee is that a client opening a file sees the data stored when the most recent writer closed the file. However, this guarantee is hard to honor in a partitioned network when the callback operation can not be performed. The recent practice of caching partial "chunks" of a file further complicates matters.

To enhance availability and to evenly distribute server load, AFS employs read-only replication on data that is frequently read but rarely modified. Subtrees that contain such data may have read-only replicas at multiple servers, but there is only one read-write replica and all updates are directed to it. Propagation of changes to the read-only replicas is done by an explicit operational procedure.

To enrich file system administration, AFS organizes data into *volumes* [34]. A volume is a collection of files forming a partial subtree of the file system hierarchy. Independent of the physical configuration of the system, volumes provide a degree of transparency in addressing, accessing, and storing files. They also facilitate data movement and optimization of existing storage. However, it is a relatively heavyweight operation to configure a machine as an Andrew server. This is in contrast to NFS, where it is trivial for a machine to export a subset of its local file system.

Coda [32, 17], a cousin of AFS, has been designed for high data availability. It achieves this with two complementary mechanisms: server replication and disconnected operation. When a client opens a file for the first time, it contacts all active 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.

The strategy that allows an object to be be read and modified as long as one of its copies is accessible provides high availability. However, data consistency can not be guaranteed upon partition failures, as data copies might be updated concurrently in two or more partitions. Although the Coda group has investigated automated file and directory conflict detection and resolution mechanisms [19, 18], not all conflicts can be resolved. In some cases, user involvement is needed to get the desired version of data.

Echo [14] and **Harp** [21] are two distributed file systems that use primary copy scheme to support mutable replication with variants of view change protocol for failure recovery. Both systems assume a local area network environment and use a single primary server to perform all updates and reads. Availability is enhanced by electing a new primary server if the original one fails. In some sense, performance can be improved by spreading the load so that different servers act as primaries for different partitions. However, a single primary server remains a potential bottleneck if it contains hot-spot partitions.

Recent years have seen a lot of work in peer-to-peer (P2P) file systems, including **OceanStore** [28], **Ivy** [25], **Pangaea** [29], and **Farsite** [4]. These systems address the design of systems in untrusted, highly dynamic environments. Consequently, reliability and continuous data availability are usually critical goals in these systems; performance or data consistency are often sacrificed. Compared to these systems, our system addresses data access and storage needs of global scientific collaboration, which can employ more reliable hardware but have more stringent requirements on average I/O performance. This leads to different design strategies in our approach.

The **Grid** [11] is an emerging infrastructure that aims to connect globally distributed resources to a shared virtual computing and storage system. Driven by the needs of scientific collaborations, the sharing that the Grid concerns with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative scientific problem-solving patterns.

Various middleware systems have been developed to facilitate data access on the Grid. For example, **Storage Resource Broker (SRB)** [6] utilizes a metadata catalog service to allow location-transparent access for heterogeneous data sets. **NeST** [7], a user-level local storage software, provides best-effort storage space guarantees, mechanisms for resource and data discovery, user authentication, quality of service and multiple transport protocol support, with the goal of bringing appliance technology to the Grid.

A common missing feature among these systems is the lack of semantic supporting for fine-grained data sharing. Furthermore, most of these systems provide extended features by defining their own API. In order to use them, an application has to be re-linked with their libraries. NFSv4 would be a complementary data access scheme to these systems in the sense that it can provide a unified name space, consistency, and file system semantics necessary to support global applications.

3. Naming Scheme

A file system provides access to a file by names. Naming therefore plays an important role in distributed file systems. In wide area networking, several principles guide the design of a naming scheme.

First, a global name space for all files in the system is desirable. By providing everyone a common frame of reference, a global name space encourages collaborative work and sharing of data. Users on any NFS client, anywhere in the world, can then use an identical rooted path name to refer to a file or directory.

Second, location independent naming facilitates transparent migration and replication. A distributed system provides transparent migration and replication by using abstract concepts and mechanisms to hide the fact that a resource is migrated or replicated, and thus present users the appearance of a single united system. With location independent naming, files are not bound by name to individual servers, so they can be transparently migrated or replicated.

Third, name space operations should scale well in the face of wide or massive distribution. A scalable distributed system is one that can easily cope with the addition of users and sites and whose growth involves minimal expense, performance degradation, and administrative complexity. These requirements suggest that a naming scheme should adopt a hierarchical architecture and allow delegation of administration.

Fourth, a naming scheme should be easy to apply in practice. With the purpose to develop a system that can be used in reality, many of our design decisions are motivated by this principle. E.g., we avoid employing a new directory service, but utilize the existing Domain Name Service (DNS) [22], which has been pervasively deployed and used for years, to support a global name space and location independent naming.

In the remaining of this section, we detail the design and implementation of a naming scheme that provides these features. Section 3.1 presents a DNS based global name space and client-side extensions that provide seamless access to the constructed name space. Section 3.2 describes how our design supports transparent file system migration and replication. Following that, Section 3.3 describes the mechanisms to support directory migration and replication.

3.1. DNS Based Global Name Space

To provide worldwide name transparency, a distributed file system should support a global name space so that applications and users can access data from everywhere without any special effort. There are two ways to implement a global name space.

First, a distributed file system can build a global infrastructure on its own to manage naming information. A prominent example that takes this approach is AFS.

In AFS, multiple administration domains are defined as cells, each with its own servers, clients, system administrators, and users. A cell's name space is represented as a mount point entry in the top level /afs directory. Each cell maintains a root volume that contains the mount point information in the top level /afs directory. Thus, AFS allows transparent access to any file in the global name space from any client without requiring a site to relinquish administrative control over its own system. However, with this design, addition or deletion of a mount point in the top-level /afs directory requires each cell to make changes to its root volume, which leads to a potential scalability problem as the number of participating sites increase. Naturally, one solution to this problem is to use a hierarchical naming architecture. However, implementing such a naming infrastructure in a global scale remains a practical problem.

Alternatively, a distributed file system can utilize an existing global name service to avoid extra development and maintenance overhead. As mentioned previously, an important feature of NFS is its simple configuration, i.e., it is trivial to configure a machine as an NFS server, compared with the relatively heavyweight work required in AFS. To provide name transparency in NFS without hurting its simplicity, we use the existing Domain Name System (DNS) to support global naming operations.

DNS is implemented in a hierarchy of name servers, with the hierarchy roughly corresponding to organizational structures. The primary use of DNS is for looking up host addresses and mail servers. However, because DNS is intentionally extensible, researchers are continuously proposing, implementing, and experimenting with new query types and functions.

In DNS, a name server maintains a collection of resource records (RR), each of which maps a name to some resource information. There are different types of resource records. The one we use is service locator resource record (SRV) which is designed to specify the location of the server(s) for a specific protocol and domain [12]. The format of the SRV RR is

`service.proto.name TTL class SRV priority weight port target`,

where `service` is the symbolic name of the desired service; `proto` indicates the transport protocol type, typically either TCP or UDP; `name` is the DNS domain name this RR refers to; `priority` sets the preference for a host specified in the `target` field; `weight` can be used in addition to `priority` for load balancing when multiple servers specified in the `target` field have the same priority; `port` is the server port on the `target` host that provides this service; and `target` specifies the DNS domain name of the host that provides the type of service being requested.

We choose SRV RR to maintain NFS server location information on account of its explicit support for server replication and load balancing. However, SRV RR has a relatively strict format, which imposes two restrictions on NFS server and client implementation, as discussed below.

First, the SRV `target` field can only be used to specify a server's DNS domain name, instead of name and path combinations. For clarification, we specify that a top-level mount entry is always mapped to the root directory exported by that NFS server. In Section 3.3, we explain how the pseudo file system concept introduced in NFSv4 supports this assumption.

Second, in SRV RR, there is no open-text field that we can use to embed file system mount options. An NFS server thus cannot declare its recommended mount options through naming resolution. I.e., with the given global name space, mount options can only be used to specify client preference. The proper options controlling the way a file system is access should be determined through the client's initial communication with the server. Because the published NFSv4 protocol has provided necessary procedures for server and client to dynamically determine proper access options, we consider this requirement easy to satisfy.

On NFS client side, we use an extended Automount/AutoFS utility to provide transparent access to the proposed global name space. 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. Upon receiving a mount request from the kernel, the AMD 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 specified interval, AutoFs instructs the daemon to unmount it.

We extend the Automount daemon program to support DNS query. The global root directory of NFS, `/nfs` by convention, is made an AutoFs mount point with DNS mapping as the associated automount map method. Initially, /nfs is empty. The first time a user or application accesses any NFS file system, the referenced name is forwarded to the modified Automount daemon which queries DNS to map the given name to a file server, and mounts it at the point of reference.

Below we provide a fictional example that shows how naming resolution works. Suppose a user types

```
ls /nfs/umich.edu/lib/file1
```

on a NFS client while the directory `/nfs` contains no entry at that time. The lookup request for entry `umich.edu` is forwarded to the extended Automount daemon which then performs a DNS query with `nfs.umich.edu` as the lookup name and `SRV` as the lookup type. Suppose the DNS server in the University of Michigan contains the following entry

```
nfs  IN SRV 0 0 2049 nfs1.umich.edu
     IN SRV 1 0 2049 nfs2.umich.edu
```

The queried results `nfs1.umich.edu` and `nfs2.umich.edu` indicate the server locations of the NFS file system provided by the University of Michigan. Upon receiving the queried results, the extended Automount daemon selects the server with the lowest priority number, i.e., `nfs1.umich.edu:/` in this example, and mounts it at directory `/nfs/umich.edu`. The subsequent access under `/nfs/umich.edu` directory is then the same as normal NFS access.

Because DNS is hierarchically distributed, addition or deletion a mount entry in the top-level global directory is quite simple. When an NFS file system is created, the administrator updates its local DNS server, adding a mapping from the file system domain name to the corresponding server location. After that, files on that file system is immediately available everywhere.

3.2. File System Replication and Migration

Although the published NFSv4 protocol includes features to support replication and migration, for the lack of dynamic server location resolution, it remains hard to perform these tasks transparently to client access. E.g., after a migration, it remains a question that how long the old server should keep running so that it can re-direct persistently connected clients to the new server. It is also hard to add or delete a replication server for notifying each client every time replication servers change can be a cumbersome administration task with a large number of widely distributed clients. Although the problem can be resolved through the server redirection mechanism suggested in the NFSv4 protocol, a single redirection server is potentially a performance and availability bottleneck.

In contrast, the presented DNS based naming resolution scheme automatically support transparent file system replication and migration. When a file system is replicated to a new server, the administrator updates the DNS entry, adding 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 a special error, NFS4ERR_MOVED for subsequent client requests, as specified by the published NFSv4 protocol. Upon receiving this error, the client queries DNS to get new file system locations and connects to a specified new server. Having redirected all the currently connected clients, the old server only needs to be kept for a short duration, equal to the Time-To-Live (TTL) of the SRV RR that stores corresponding server location information.

3.3. 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 exported directories on an NFS server into a single rooted tree. Portions of the server name space that are not exported are bridged so that an NFSv4 client can browse seamlessly from one export to another. Exported file system structures are controlled by servers, so a server can dictate a common view of the file system that it exports. This feature is essential for support of a single global name space, as mentioned in Section 3.1.

NFSv4 uses `exportfs` utilities on the server side to export a directory. In the Linux kernel, an `export` structure is maintained for each currently accessed directory export. To support directory migration and replication, we extend the `exportfs` interface, allowing the reference string of a replicated directory to be passed into the kernel. After that, the reference string is maintained in the corresponding `export` structure.

A reference string includes information on how to get directory replica locations, such as replica lookup methods and lookup key. When a replicated directory is accessed by a client for the first time, the server resolves the replica locations of that directory with the attached reference string. It then sends this information to the client through the FS_LOCATIONS attribute, as the NFSv4 protocol specifies. Upon receiving the replica locations, the client selects a nearby replication server, mounts it at the place of reference, and continues its access. When a directory is migrated to another server, the old server returns NFS4ERR_MOVED error for subsequent directory requests. Upon receiving the error, the client obtains the new location of the migrated directory by examining the content of the FS_LOCATIONS attribute and connects to the specified server.

In our implementation, the NFS server performs replica location lookup through `rpc-cache`. When an NFS client encounters a directory export with an attached reference string, the server 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. Upon any changes of replica location information, administrator can use `exportfs` facility to flush stale data out of `rpc-cache`.

The current prototype allows four types of reference strings: LDAP, DNS, File and Server Redirect. The format of each type is presented in the appendix. Supporting multiple lookup methods is based on the concern that different NFS systems can have different requirements and preference for replica location management. A system that holds a large number of or frequently changed replicated directories may prefer to using LDAP, with which it is easy for administrator to check and manage replica information through LDAP queries and updates. DNS RR is suitable for managing a moderate number of infrequently changed replicated directories. With a small number of replicated directories, a NFS server can simply store their location information in a file, avoiding extra service management. In case of migration, Server Redirect can be used which requires no extra configuration and guarantees fast response time.

4. Mutable Replication

In this section, we present the design and implementation of an extension to NFSv4 that enables mutable (i.e. read/write) replication with flexible consistency guarantees, small performance penalty, and good scaling properties. The system can guarantee either ordered writes or synchronized access, without adding overhead on normal reads. It can tolerate a large class of server crash or link failures, even when these lead to network partitioning. Our design uses standard POSIX features, which makes it easy to deploy. Below, we first describe a replication scheme that guarantees ordered writes in Section 4.1. Based on that, we present the additional mechanisms to enforce synchronized access in Section 4.2. In the following discussion, we refer the first consistency model as sequential consistency, and the second as synchronized accessed.

4.1. Sequential Consistency

To support mutable replication, we need mechanisms to distribute updates when a replica is modified and to control concurrent accesses when writes occur. To prevent mutable replication from affecting exclusive read or shared read performance, we adopt an extended primary copy scheme with operation forwarding to coordinate concurrent writes. Compared with the traditional primary copy scheme, our design has the following advantages.

First, the overhead to support mutable replication is induced only when writes occur. When there are no writes, the system behaves as a read-only replication system, i.e., a client accesses data from any nearby replication server.

Second, a primary server is selected on the granularity of a single file, which allows fine-grained load balancing.

Third, in our scheme, a primary server is dynamically chosen when a file write opened for write. In most cases (exclusive write cases), a client's write requests are served by a nearby primary server. The solution is well suited for wide-area collaborations in which a replica is often dynamically created and it is hard to decide an optimal primary server for a file in advance.

Fourth, it provides higher availability because a client can choose any working server to read or write a file.

Below we first describe the mechanisms we use to support file updates in Section 4.1.1. Section 4.1.2 then presents the failure recovery mechanism of the protocol in case of server crash and network partitions. For directory updates, a similar approach is used, with several performance improvements presented in Section 4.1.3.

4.1.1. File Updates

When a client opens a file for writing, the chosen server temporarily becomes the primary for that file. All other replication servers are instructed to forward client write requests for that file to the primary server. When the file is closed, the primary server withdraws from its leading role by notifying other replication servers to stop forwarding writes. In the following discussion, we refer to the first procedure as disabling replication, and the latter as re-enabling replication, although by default, read requests received on other replication servers are still processed locally.

While a file is open for writing, the primary server is responsible for distributing updates to other replicas. We consider two strategies for distributing updates. The first is distributing updates when the modified file is closed. The second strategy distributes updated data to other replicas as they arrive.

Although naive, the update-on-close strategy does avoid multiple updates should some or all of the file be written several times. However, if a client writes a massive amount of data to a file and then closes it, the close operation takes a long time to return. Furthermore, we run the risk of losing all client updates if the primary server fails before distributing the new data, which invalidates any assurance of durability to the client for individual write operations.

Distributing updated data to other replication servers every time the primary server receives a write request eliminates the update propagation delay for a close request. It also facilitates recovery from primary server failure: a client receives a positive acknowledgment for every successful write, so if the primary server fails, the client can connect to a new server (using standard NFSv4 client recovery mechanisms) and reissue at most one unacknowledged write request. However, unlike distributing updates at the time the file is closed, this strategy adds to network traffic if a client overwrites file blocks.

We prefer the latter scheme. We hesitate to impose a sweeping change to system call behavior and we are willing to expend some network resources to reduce latency. Yet, by making client-to-server writes synchronous with updates to other replication servers, it appears that we are increasing client write latency, not reducing it. The paradox is resolved by observing that NFSv4 writes are usually through an I/O daemon that delays writes for some seconds [20]. This relaxes the dependency of application performance on primary server latency. The I/O daemon's delayed-write policy also increases the likelihood that the updates will be long-lived [5].

The primary server distributes updates to other replication servers in parallel. Updates must be delivered in order, either by including a serial number with the update or through a reliable transport protocol such as TCP. In addition to the data payload, each update message from the primary server to other replicas also includes metadata related to the update, such as modification time. Each replication server modifies its copy of file metadata appropriately after updating file data. This guarantees that the metadata of the file is consistent among replicas, which as we show in Section 4.1.2, makes it easy to determine the most recent file copies during failure recoveries.

Initially, a replication server is unsure if a received update is valid - e.g., the primary server might send out an update request and then immediately crashed - so it does not apply the update at once. Rather, the request is cached until the next update or a replication re-enabling message is received from the primary.

Two or more servers may try to become the primary for a file at the same time. 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 already disabled more replicas is allowed to continue; the server that has so far disabled fewer replicas hands its collection of disabled replicas to the first server; when a tie happens, the server with bigger IP address is allowed to proceed. If conflicting servers are in different partitions, at most one can collect acknowledgments from a majority of the replication servers. For some kinds of failure, e.g., multiple network partition failures, it is possible that no primary server can be elected. We discuss this case further in the next subsection.

4.1.2. Failure Recovery

Our primary copy scheme guarantees consistent access when all replicas are in working order. However, failure complicates matters. Different kinds of failure may occur, including client failure, replication server crash failure, network partition, and combinations of these cases. Here, we briefly describe the failure detection and recovery mechanisms for each case. A detailed description and proof of correctness is presented elsewhere [35].

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. In the event of 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 any files left open by the failed client on its behalf. If the client was the only writer, then the primary re-enables replication for the file at this time. Unsurprisingly, the file content reflects all writes acknowledged by the primary server prior to the failure.

To support sequential consistency, the system maintains an active group view among replicas and allows updates only in the active group. We require that an active group contain a majority of the replicas to ensure its uniqueness. During file modifications, the primary server removes from its active group view any replicas that fail to acknowledge 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 acknowledgments from a majority of replicas. If the active view shrinks to less than a majority, the primary server "fails" the client request. The primary server sends its active view to other replication servers when it re-enables replication. A server not in the active view may have stale data, so the re-enabled servers must refuse any later replication disabling or update requests that come 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.

If a primary server crashes or is separated in a minority partition, a replication server (in the majority partition) detects this failure when a forwarded request times out. In that case, the replication server starts a failure recovery procedure to become the replacement primary. Basically, the replication server 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, and distributes a new active group view. It then re-enables replication on the active servers.

With these mechanisms, our system can guarantee sequential consistency and continuously serve client requests as long as a majority of replicas are in working order and can communicate. If there are multiple partitions and no partition includes a majority of the replication servers, read requests can continue to be satisfied, but no write requests can be served until the partition heals. We assume this happens rarely.

To tolerate various edge scenarios, e.g., in the case that all servers crash when the primary server is processing a write request, it appears that a replication server should record in its stable storage all information related to an update, such as the current primary server, cached update, serial number and failed replicas. This strategy, however, would reduce write performance. Instead, we rely on the system administrator's involvement in the case that more than majority of replication servers fail. Because our protocol guarantees sequential consistency, the administrator can simply use a synchronization tool (i.e., `rsync`) which compares the data states among replication servers and selects the most recent copy if inconsistent copies are detected. After the synchronization is complete, the administrator can bring all replication servers to normal state. Thus, a replication server needs to record in stable storage only minimal information, i.e., the failed replication servers it currently knows and the primary server it currently admits for a file.

El-Abbadi et al. have studied the failure recovery problem of read-one-write-all replication scheme in distributed database systems. They point out that replicas can not independently or asynchronously update their *can-communicate* views because network connections may be intransitive or replicas may detect connection changes at different time. Taking a further step, they present a series of properties and rules which they show are sufficient conditions for a replication protocol to guarantee one-copy serializability. We have extended this theory into distributed file systems and have proved that our failure recovery protocol guarantees sequential consistency in the face of node crash or network partition. The basic rationale lies in having a single server (primary server or replacement server) decide the view of the majority partition. This view is then distributed to and sustained by all the members contained in it. Thus, a unique and consistent majority view is guaranteed. For more details, readers can refer to our technical report [35].

4.1.3. Directory Updates

Directory modifications include creation, deletion, and modification of entries in a directory. Unlike file writes, a directory modification may involve more than one object. We require replication for all involved objects to be disabled before processing a directory update. These disabling requests are grouped and processed together, so that no deadlock might happen. Furthermore, 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. So instead of redirecting access requests to a replicated directory while an update is in progress, replication servers block access requests to that directory until the primary server re-enables replication. Like directory modifications, attribute updates proceed quickly, so we handle them the same way.

When disabling directory replication, the primary server sends the replication disabling request and the update request together in one compound RPC. A replication server receiving this compound RPC caches the update, begins to block local update, and acknowledges the request. After receiving replies from a majority of replication servers, the primary server acknowledges the client request. Simultaneously, the primary server could send a commit message, notifying other replication servers to apply the update. However, to reduce network traffic, we delay this notification until the primary server re-enables replication, i.e. a replication server applies the cached update when it receives the re-enabling replication request from the primary server.

One issue introduced by this optimization is the possibility for a replication server to receive "invalid" replication disabling requests. Consider a simple example in which a client first creates a file `c` in directory `/a/b/`, then opens it for writing. As described above, with the create request, the connected server sends replication disabling requests for directory `/a/b/` combined with the update request to create entry `c` to other replication servers. It acknowledges the client after receiving replies from a majority of replication servers. As the client might send the write open request for file `c` immediately after receiving this acknowledgment, it is possible that the connected server sends a replication disabling request for file `/a/b/c` before it re-enables replication for directory `/a/b/` on other replication servers. As a result, the replication disabling request for file `c` would be rejected by other replication servers since they have not applied the cached create request. Although the problem can be solved by having the primary server simply keep re-sending the second request, a lot of redundant network traffic would be induced, especially when the system consists of slow (far) replication servers. So instead, we take another approach. In our implementation, before sending a replication disabling request, the primary server first checks whether any parent directory of the to-be-modified entry is being disabled ; if so, it would wait untill that directory was re-enabled.

Consequently, the performance of directory updates is normally decided by the RTT between the primary server and the majority of replication servers; however, when a client issues a burst of directory updates, the performance might be slowed down by a replication server far away as the primary server re-enables the replication for a directory when it receives acknowledgments for the previous replication disabling requests from all other replication servers or upon a timeout if a failure happens. Other solutions exist to solve this kind of problem. For example, the primary server can pre-send a commit request for a directory update. In our future study, we want to compare these different solutions in terms of their performance and induced network traffic with real application operations.

4.2. Access Synchronization

The protocol discussed so far efficiently provides sequential consistency guarantees. However, applications may sometimes require stronger consistency guarantees, i.e., synchronized access. To meet such needs without imposing overhead on data access that requires sequential consistency only, our system provides synchronization guarantee as an option that can be demanded by applications through POSIX synchronization flags in the open system call interface [3].

By the POSIX specification, if an application opens a file with `O_SYNC` flag set, a subsequent write operation is complete only when the written data and all file attributes relative to the operation, e.g., modification time, is written to the permanent storage; if an application opens a file with both `O_SYNC` and `O_RSYNC` flags set, a read operation is complete only when any pending writes affecting the data to be read is successfully transferred to the requesting process.

In a local file system, support for these synchronization requirements usually adopt the same solution, so some operating systems, notably Linux, treat the three synchronization flags equally. However, in a distributed environment, it is beneficial to distinguish these different synchronization requirements, as the cost to support them can be considerably different.

In NFSv4, a client's `WRITE` operation request includes a special flag that declares whether the written data is to be synchronized. A NFSv4 client sets this flag on user's behalf if the `O_SYNC` flag of the file is set or the user issues a `fsync` system call. However, the synchronization requirement specified with `O_RSYNC` flag set is not addressed. We notice that the synchronization guarantees required with `O_SYNC` and `O_RSYNC` flags set correspond to synchronized read requirement. Taking these flags as the hint that the application is demanding synchronized access, we refine our replication protocol as follows.

When the primary server receives a synchronous write request from a client, it must ensure that the replication for the file has been disabled on every other replication server before returning a reply to the client. By default, a replication server forwards write requests only while its replication is disabled. However, if during this period, a client opens the file with both `O_SYNC` and `O_RSYNC` flags set, the replication server forwards the client's read requests to the primary server as well.

In most cases, the update distribution procedure works the same way as that in the sequential consistency model - The primary server acknowledges a client's write request after it gets acknowledgments from a majority of replication servers; if there is a failure detected during update distribution, the primary server can still process client's read and/or write requests as long as it is in the majority partition. However, if the primary server is separated in a minority partition, it is not guaranteed the distributed update reaches the majority partition. If a client opens the file with both `O_SYNC` and `O_RSYNC` flags set, the primary server must refuse its subsequent read requests for the file to guarantee that no stale data is served. In the majority partition, the failure recovery mechanism described in Section 4.1.2 can be used to recover the fresh copy of data. After that, read requests can be served in the majority partition.

With the described mechanism, slight overhead is induced to guarantee synchronized access when applications demand it; longer delay is charged on forwarded operations if concurrent writes occur; If a file is not under modification, any read requests for the file, even those with synchronization requirement, are processed by a nearby server.

It is easy to see that this approach provides synchronization guarantee at the cost of sacrificing system availability for synchronous write operations. I.e., if a failure happens, an application can not synchronously write a file. Several methods can be used to bypass this restriction if it is critical to guarantee system availability for write operations as well as synchronized access. For example, we can use periodic heartbeat messages to detect partition failures, and require a replication server to reject any client requests if it fails to receive replies from a majority of replication servers. Consequently, the system can continue to process synchronous write operations after a heartbeat period, as long as a majority of replication servers are active. However, we believe that the current solution is superior in most scenarios because it adds no overhead or network traffic to normal operations.

There are other possible ways to decide consistency guarantees for a file. For example, we can implement it as an extended attribute associated with the file. The proposed approach is favored for it allows applications to control file sharing behavior more flexibly. Consider the example of an edit-and-run procedure: The program is edited on one client, and then a number of clients are instructed to execute it. Because the execution instruction can be issued immediately after the program editing, the access on the file must be coordinated. In our replication system, correct synchronization behavior can be guaranteed if the editor application (writer) issues a `fsync` system call after completing the editing, and the execution application (reader) opens the file with both `O_SYNC` and `O_RSYNC` flags set. On the other hand, another application, e.g., a snapshot tool, can choose to open the file without setting any synchronization flags as sequential consistency is sufficient to guarantee its correctness. In the extreme case, if a file is always opened with `O_SYNC` and `O_RSYNC` flags set, strict consistency is provided.

However, using open synchronization flags also introduces two issues: First, existing programs might not use these flags to specify synchronization requirements. Thus modifications are required on the program's open calls to ensure synchronized access. We believe such modifications can be performed easily enough that it does not affect the prevalence of the system. Second, the NFSv4 protocol does not provide the mechanism for a client to send open synchronization flags to its server. Our current implementation conveys this information by using the extra bits of the `share_access` flag in `open` operation request, which requires extension to the published NFSv4 protocol.

5. Evaluation

This section presents the evaluation results for the described naming scheme and replication protocol. In all the presented experiments, we use a prototype implemented in Linux 2.6.12 kernel. Servers and clients all run on dual 2.8GHz Intel Pentium4 processors with 1024 KB L2 cache, 1 GB memory, and dual Intel 82547GI Gigabit Ethernet cards onboard. We use TCP as the transport protocol. The number of bytes NFS uses for reading (rsize) and writing files (wsize) is set as 32768 bytes.

5.1. Naming Evaluation

When a client accesses an NFSv4 file system for the first time, the logical name that it uses needs to be mapped to the physical locations. To evaluate this cost, we measured the delay a client experiences when it first accesses an NFSv4 file system in the provided global name space. The evaluation data are collected at the client side with `gettimeofday` utility, which has microsecond resolution in 2.6 Linux. Table 1 presents the evaluation results collected with the NFS client and the connected server within the same LAN. The round trip latency (RTT) between the client and the server is around 200 μsec. As the results show, when server and client are close to each other, the overhead of naming resolution in the top-level directory is small relative to the overall mount time.

Both DNS query latency and mount latency increase as client and server are further apart. DNS performance has been studied in large-scale previously. E.g., in 2000, Jung et al. conducted a detailed analysis of DNS traces collected on the Internet links of the MIT Laboratory for Computer Science and the Korea Advanced Institute of Science and Technology (KAIST) [16]. They observe that more than half of lookups are answered in 0.1 second in all the collected traces. With MIT traces, around 90% of lookups are answered in 1 second. The KAIST trace has more long-latency queries than MIT traces, roughly about 25% of them takes more than 1 second. Most of these lookups correspond to names outside Korea.

In this paper, we do not intend to repeat such detailed DNS evaluations. However, to estimate the relative overhead added by DNS query to NFSv4 over wide area networks, we compare the client-perceived latency for DNS query and NFSv4 mount with hosts taken from Planetlab, an open, globally distributed platform consisting of 160 machines hosted by 65 sites spanning 16 countries [9]. The RTT between the client and the 522 Planetlab hosts used in our experiment ranges from 0.6 millisecond to more than 300 milliseconds. Figure 1 shows the cumulative RTT distribution measured with the client pinging each Planetlab host. Figure 2 presents the cumulative DNS lookup distribution collected by running `dig` on the client. Figure 3 displays the client-perceived mount latency across the range of client-to-server RTT measured in Figure 1 experiment. In the experiment, we use NistNet [8] to simulate the network delay between the server and the client.

As Figure 2 shows, even in a worst case scenario where DNS queries are performed at the first time to a large scale of remote hosts, more than 60% lookups complete in 1 second. Compared with the mount latency depicted in Figure 3, e.g., mount takes around 1.5 seconds corresponding to 0.1 second client-to-server RTT, the overhead of DNS query over WAN is still insignificant. Furthermore, we note that when a client mounts a NFS file system with server host name, it also experiences DNS lookup overhead. Because answers for SRV RR queries normally contain servers' addresses as well, resolving a mount point name in the discussed global name space actually adds little overhead compared to conventional mount.

Phase	Time (ms)
Replica List Query (DNS)	0.973
Mount	6.45
Others (upcall, etc.)	0.619
Total	8.05

Table 1: First-time access latency for a local NFSv4 file system in the top-level directory.

To evaluate the cost of looking up physical locations for a replicated directory, we measured the latency that a server experiences when a client accesses a replicated directory for the first time. Table 2 presents the experimental results for LDAP, DNS, and File query methods. In all experiments, the referenced directory is replicated on two servers.

As the data shows, File query is the fastest with around

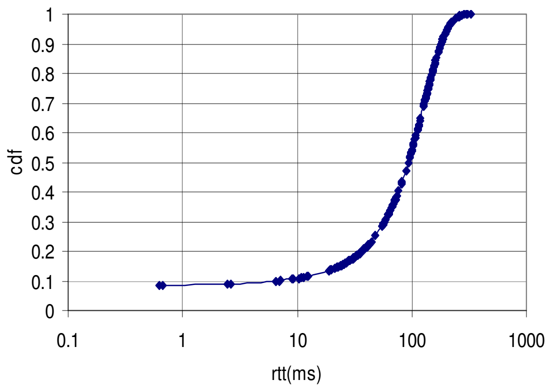

Figure 1: Cumulative distribution of RTT for Planetlab machines.

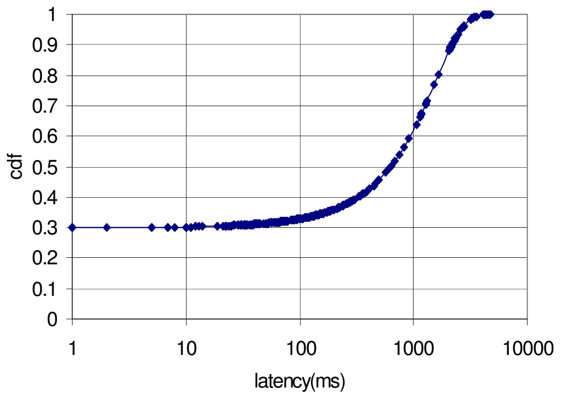

Figure 2: Cumulative distribution of DNS lookup latency for Planetlab machines.

0.1ms query time. The presented result for File query is collected with the file containing no extra entries except for the replica list of the tested directory. The query time increases as the file contains more replica entries. For example, our measurements show that when the file contains 1000 entries, the query time increases to 0.83ms. However, because File method does not provide a central point of information management, we expect it is only used to manage a small number of replicated directories. With the other two, the query time for DNS TXT RR is around 0.8ms, and that for LDAP is about 4.8ms. The performance difference is mostly caused by the different transport protocol used in these two types of queries, i.e., LDAP uses TCP but DNS uses light-weight UDP.

Generally, we consider the measured overhead for all of the supported query methods is acceptable. Furthermore, we note that by using rpc-cache, this query cost is induced only once for a replicated directory that is repeatedly accessed.

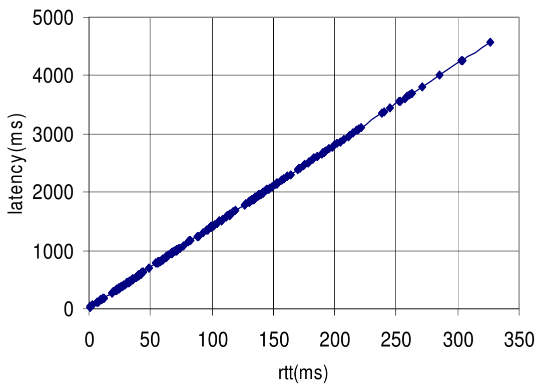

Figure 3: Client-perceived mount latency as client-to-server RTT increases.

Query Method	Time (ms)
LDAP	4.77
DNS TXT RR	0.847
FILE	0.124

Table 2: First-time access latency for a replicated directory with different lookup methods.

5.2. Replication Evaluation

To assess the cost of replication, this subsection presents the experimental results of running a modified Andrew benchmark over rNFS. The Andrew benchmark [15] is a widely used file system benchmark that models a mix of file operations. It 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 our experiments differs from the original Andrew benchmark in two aspects. First, in the last stage, it compiles different source code than that included in Andrew benchmark package. Second, the Andrew benchmark writes a log file in the generated directory; if writes are slow compared to reads, the cost of updating the log file dominates the overall cost of a stage that mostly reads, hindering analysis. Therefore, we use local disk to hold the log file.

Our first experiment looks at replication in a LAN environment, such as a cluster. Figure 4 depicts the performance of the modified Andrew benchmark as the number of replicas increases. The measured RTT between any two machines is around 200 μsec. Figure 4 shows that in a LAN, the penalty for replication is small. Replication induces no performance overhead in Stages (*III*) and (*IV*), as these two phases consist of read operations only. Stage (*V*) is compute-intensive, so the performance difference between single server and replicated servers is negligible. Most of

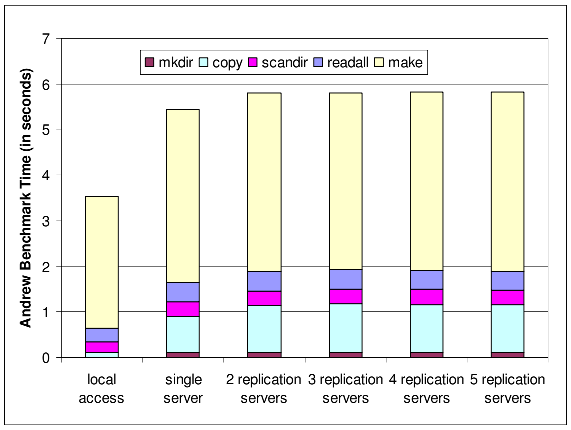

Figure 4: MAB in LAN replication.

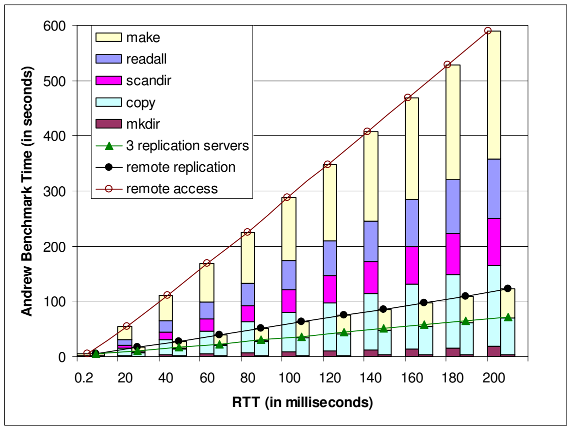

Figure 5: MAB in WAN replication.

the performance penalty for replication comes in Stages (*I*) and (*II*), which consist of file and directory modifications. However, with a fast network, the aggregate penalty is still only a few percent. Furthermore, because a primary server distributes updates to other replication servers in parallel, performance is not adversely affected as the number of replication servers increases.

The next experiment, depicted in Figure 5 compares the cost of replicating to a distant server vs. the cost of accessing a distant server directly. We ran the modified Andrew benchmark with an increasingly distant file server, the upper line in Figure 5, and again with a local replication server and an increasingly distant replication server, the middle of the three lines in Figure 5. The RTT marked on the X-axis shows the round-trip time between the primary server and the remote replication server for the replication experiments, and between the client and the remote file server for the remote access experiments.

In Figure 5, the smallest RTT measured is 200 μsec., the network latency of our test bed LAN. For the other measurements, we use Netem [13], a Linux tool that simulates network delays. Each experiment first warmed the client's cache with two runs.

Figure 5 shows that replication outperforms remote access in all five stages. In Stages (*III*) and (*IV*), the read-only stages, replication is as fast as local access, since no messages need to be sent to the other replication server in these stages. But replication also dominates remote access in the other three stages. To see why, we take a close look at the network traffic in the measured experiments, where we find that with replication, fewer messages are sent to the remote server, accounting for its advantage.

We model this as follows. The running time in each stage can be estimated as

$$T = Tbasic + RTT \times NumRPC \qquad (1)$$

where *Tbasic* denotes the computation time at the client and the request processing time at the connected server. For replication, *RTT* represents the round-trip time between the primary server and the replication server, and *NumRPC* represents the number of RPC messages sent from the primary to the replication server in the corresponding stage. For remote access, *RTT* represents the round-trip time between the client and the remote server, and *NumRPC* is the total number of RPC requests sent from the client to the server.

The *Tbasic* cost is about the same for replication and remote access, so any difference in performance must be accounted for by the second part of the formula. For example, Stage (*I*) creates 20 directories at a cost of 85 RPC requests sent from the client to the connected server (20 `create`, 13 `access`, 17 `getattr` and 35 `lookup`). The reported `access`, `getattr` and `lookup` requests are unavoidable even for a warm cache run because they are requesting information on newly created directories. However, with replication, `access`, `getattr` and `lookup` requests are served locally at the primary server, eliminating their cost altogether at the scale of this experiment. Furthermore, although each create costs two RPC messages, the primary server replies to the client after receiving the response to only the first of these two. Consequently, the number of latency-inducing remote RPC messages in the replication experiment decreases from 85 to 20. Table 3 shows summary RPC counts for the other stages.

One important feature of rNFS is that a primary server can reply to a client request as soon as it gets acknowledgments from half of the other replication servers. Given a set of replication servers, then, the performance of rNFS is dictated by the largest RTT between the primary server and half of the nearest replication servers, which we call the *majority diameter*.

To illustrate how this feature can be used to advantage, we added a third replication server halfway (in terms of RTT) between the other two and re-ran the modified An-

System Model	Mkdir	Copy	Scandir	Readall	Make	Total
Replication	20	228	0	0	71	309
Remote Access	85	735	154	510	589	2073

Table 3: Number of remote RPCs of MAB in Replication and Remote Access.

Replication servers	RTT to the primary server
P (the primary server)	-
R2, R3	20 msec
R4	40 msec
R5, R6, R7	60 msec

Table 4: Servers used in Figure 6 experiments.

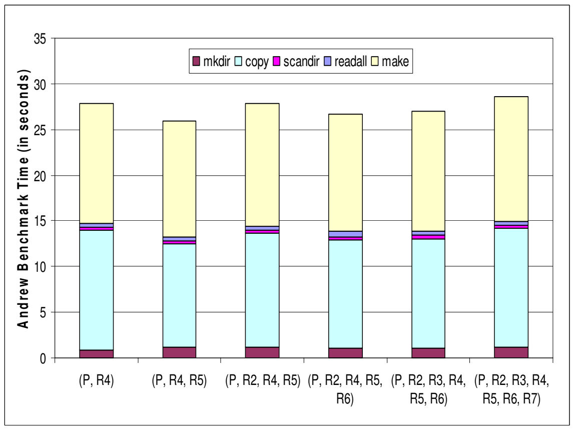

Figure 6: MAB with different replication server sets.

drew benchmark. The result is the lowest of the three lines in Figure 6. Placing the third replication server midway between the local and remote replication servers cuts the majority diameter in half and for the Andrew benchmark, this cuts the overall run time, which is dominated by the cost of remote RPCs, nearly in half.

The results imply if most writes to a replicated file come from one site, the performance overhead for remote replication can be made scant by putting a majority of replication servers near that site. Furthermore, if a site is using local replication, then the penalty for adding a distant replication server, say, for off-site backup, is negligible.

Figure 6 compares the running time of the modified Andrew benchmark with a fixed majority diameter and a varying number of replication servers. The servers used in this experiment are described in Table 4. The experimental results show that with the majority diameter fixed (at 40 msec), increasing the number of replication servers has negligible effect on system performance, which is key to good scaling.

To summarize, the evaluation data presented in this subsection illustrate two main points. First, network RTT is the dominant factor in rNFS WAN performance. By locating a replication server close to the client , rNFS can mask RTT-induced latency. Second, rNFS scales well in this workload. Application performance is unaffected by adding additional replication servers while keeping the majority diameter fixed.

As a gedanken experiment, we might imagine the practical limits to scalability as the number of replication servers grows. A primary server takes on an output bandwidth obligation that multiplies its input bandwidth by the number of replication servers. For the near term, unicast communication and the cost of bandwidth seem to be the first barriers to massive replication.

6. Conclusion

This paper presents the design, implementation, and analysis of a naming and replication scheme 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 mutable replication, we use a variant primary copy scheme with operation forwarding to provide concurrency and consistency during replica updates. The system performs admirably in environments where reading is the dominant operation: replication introduces no additional cost for exclusive or shared reads. The system guarantees either ordered writes or synchronous access, even in partition failures. With these features, we believe that our system presents a promising way for geographically distributed organizations to access and store data over the Internet.

Acknowledgements

This work was partially supported by the NSF Middleware Initiative Grant No. SCI-0438298 and by Network Appliance, Inc.

References

[1] Sun Microsystems. NFS: Network file system protocol specification. RFC 1094, Network Working Group, March 1989.

[2] Sun Microsystems. NFS: Network file system version 3 protocol specification. RFC 1813, June 1993.

[3] *UNIX man pages: open(2)*, second edition, 1997.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of 5th*

Symposium on Operating Systems Design and Implementation, 2002.

[5] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

[6] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proc. of CASCON'98*, 1998.

[7] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Proc. of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[8] M. Carson and D. Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[9] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[10] M. Eisler, A. Chiu, and L. Ling. RPCSEC_GSS protocol specification. RFC 2203, Network Working Group, September 1997.

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

[12] A. Gulbrandsen. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, February 2000.

[13] S. Hemminger. Netem - emulating real networks in the lab. LCA2005, April 2005.

[14] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and consistency trade-offs in the Echo distributed file system. In *Proc. 2nd IEEE Workshop on Workstation Operating Syst*, 1989.

[15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[16] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *IMW '01: Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 153–167, 2001.

[17] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 213–225, 1991.

[18] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the coda file system. In *Proc. of the 2nd international conference on Parallel and distributed information systems*, pages 202–213, 1993.

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

[20] C. Lever. Using the Linux NFS client with Network Appliance Filers. Technical Report Netapp TR-3183, 2003.

[21] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 226–38, 1991.

[22] P. Mockapetris. Domain names - concepts and facilities. STD 13, RFC 1034, November 1987.

[23] P. Mockapetris. Domain names - implementation and specification. RFC 1035, November 1987.

[24] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: a distributed personal computing environment. *Commun. ACM*, 29(3):184–201, 1986.

[25] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[26] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *USENIX Summer*, pages 137–152, 1994.

[27] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proc. of the 2nd International System Administration and Networking Conference (SANE2000)*, page 94, 2000.

[28] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proc. of the Conference on File and Storage Technologies*. USENIX, 2003.

[29] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[31] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: principles and design. In *Proc. of the 10th Symposium on Operating Systems Principles*, pages 35–50, 1985.

[32] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[33] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 protocol. RFC 3530, April 2003.

[34] B. Sidebotham. VOLUMES – the Andrew file system data structuring primitive. In *Proc. of European UNIX Systems User Group Autumn'86*, pages 473–480, 1986.

[35] J. Zhang and P. Honeyman. A replica control protocol for distributed file systems. Technical Report CITI-TR-04-1, Ann Arbor, MI, USA, April 2004.

A. Format of Reference String for Directory Replication Lookup

- **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. We define a new objectclass for NFS replica location information whose format is provided in Appendix B. Each replica location record includes one or more fslocations attribute. An fslocations attribute includes the server name holding that replica, the path where the replicated directory located, and mount options. In our current implementation, the following format is adopted: `server:/path option`.
- **DNS**. The format of a DNS reference string is `dns://lookup-name`. The format of lookup-name follows domain name conventions. As mentioned in Section 3.1, DNS SRV RR does not provide a way to store server name and directory path combinations. So we use another type of resource record, TXT RR [23], to store directory replica location information. TXT RR contains one or more character string fields that are used to hold descriptive text. The semantics of the text depends on the specific application it is used. Our current implementation adopts the following format for directory replica location information:
  `name class TXT NFSv4 server path priority weight`.
  The keyword `NFSv4` indicates that the record stores NFS replica location information. `priority` and `weight` follow the same interpretation as those in SRV RR. E.g., the following TXT RR

  ```
  user IN TXT NFSv4 s1.umich.edu
  /user 0 1
      IN TXT NFSv4 s2.umich.edu
  /pub/u/ 0 1
  ```

  provides two locations for a replicated directory: one is `s1.umich.edu:/user` and the other is `s2.umich.edu:/pub/u`.
- **File**. The format of File reference string is `file://pathname/lookup-key`. The pathname gives the path to the file storing lookup-key to replica location mappings.
- **Server Redirect**. The format of Server Redirect reference string is `server://hostname:/path`, where `hostname:/path` gives the location of the migrated or redirected directory.

B. LDAP Schema for NFS Replica Location Information

```
# Attribute Type Definitions

attributetype ( 1.3.6.1.4.1.250.1.64
  NAME ( 'fslocations' )
  DESC 'NFS version 4 FS_Locations
Information'
  EQUALITY caseIgnoreIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
)

objectclass ( 1.3.6.1.4.1.250.1.65 NAME
'rnfs'
  DESC 'NFS version 4 FS_Locations
Information'
  SUP top AUXILIARY
  MUST ( cn $ fslocations )
  MAY ( description ) )
```