The NFS Version 4 Protocol

Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler,
David Noveck, David Robinson, Robert Thurlow

Abstract

The Network File System (NFS) Version 4 is a new distributed file system similar to previous versions of NFS in its straightforward design, simplified error recovery, and independence of transport protocols and operating systems for file access in a heterogeneous network. Unlike earlier versions of NFS, the new protocol integrates file locking, strong security, operation coalescing, and delegation capabilities to enhance client performance for narrow data sharing applications on high-bandwidth networks. Locking and delegation make NFS stateful, but simplicity of design is retained through well-defined recovery semantics in the face of client and server failures and network partitions.

This paper describes the new features of the protocol, focusing on the security enhancements, integrated locking support, changes to fully support Windows file sharing semantics, support for high performance data sharing, and the design points that enhance performance on the Internet. We describe applications of NFS Version 4. Finally, we describe areas for future work.

1. Background

The Network File System, or NFS, was developed by Sun Microsystems to provide distributed transparent file access in a heterogeneous network. In the summer of 1998, Sun Microsystems ceded change control of NFS to the Internet Engineering Task Force [RFC2339]. IETF assumed the responsibility to create a new version of NFS for use on the Internet.

Prior to the formation of the IETF NFS Version 4 working group, Sun Microsystems deployed portions of the technology leading up to NFS Version 4, notably WebNFS [RFC2054, RFC2055] and strong authentication with Kerberos [MIT] within a GSS-API framework [RFC2203]. In August 1998 Sun submitted a strawman NFS Version 4 protocol specification to the newly formed working group. Following discussions in the working group, and contributions by many members, prototype implementations of the protocol began to prove out the concepts. Initial implementation testing of prototypes (including a Java prototype) based on the working drafts occurred in October 1999 to verify the design. The specification was submitted to the Internet Engineering Steering Group for consideration as a Proposed Standard in February 2000. Further implementation work and interoperability testing occurred early March 2000.

1.1. Requirements

As part of the IETF process, Sun Microsystems submitted an initial draft of a requirements document for NFS Version 4 to the newly formed working group. After wide review and some minor revisions [RFC2624], the requirements for NFS Version 4 were specified to be:

- Improved access and good performance on the Internet
- Strong security, with security negotiation built into the protocol
- Enhanced cross-platform interoperability
- Extensibility of the protocol

Additionally, we sought improvements in locking and performance for narrow data sharing applications.

2. The NFS Version 4 protocol

> Old Marley was as dead as a door-nail.
> Dickens, *A Christmas Carol*

The NFS Version 4 protocol is stateful.

NFS is a distributed file system designed to be operating system independent. It achieves this by being relatively simple in design and not relying too heavily on any particular file system model. NFS is built on top of the ONC Remote Procedure Protocol [RFC1831]. A *Remote Procedure Call* (*RPC* or *procedure*) defines a procedural model for distributed applications, and is the underlying architecture of all NFS implementations. The *External Data Representation* (*XDR*) [RFC1832] enables heterogeneous operation by defining a canonical data encoding over the wire. A *server* in the RPC architecture provides a service by supporting a set of remote procedures in a well-defined distributed application. A *client* is a user of those services.

The first major structural change to NFS compared to prior versions is the elimination of ancillary protocols. In NFS Versions 2 and 3, the Mount protocol was used to obtain the initial filehandle, while file locking was supported via the Network Lock Manager protocol. NFS Version 4 is a single protocol that uses a well-defined port, which, coupled to the use of TCP, allows NFS to easily transit firewalls to enable support for the Internet. As in WebNFS, the use of initialized filehandles obviates the need for a separate Mount protocol [RFC1813]. Locking has been fully integrated into the protocol – which was also required to enable mandatory locking. The lease-based locking support adds significant state (and concomitant error recovery complexity) to the NFS Version 4 protocol.

Another structural difference between NFS Version 4 and its predecessors is the introduction of a COMPOUND RPC procedure that allows the client to group traditional file operations into a single request to send to the server. In NFS Versions 2 and 3, all actions were RPC procedures. NFS Version 4 is no longer a "simple" RPC-based distributed application. In NFS Version 4, work is accomplished via operations. An *operation* is a file system action that forms part of a COMPOUND procedure. NFS Version 4 operations correspond functionally to RPC procedures in former versions of NFS. The server in turn groups the operation replies into a single response. Error handling is simple on the server – evaluation proceeds until the first error or last operation whereupon the server returns a reply for all evaluated operations.

We introduced the COMPOUND procedure to reduce network round trip latency for related operations, which can be costly over a WAN (for example, the Internet). The model NFS Version 4 uses implies the NFS layer engages more closely in the marshalling and unmarshalling of data, which complicates implementation. NFS Version 3 was designed to be easy to implement given an NFS Version 2 implementation. NFS Version 4 did not have that requirement. The only RPC procedures in NFS Version 4, in the strict sense, are NULL and COMPOUND, and their callback analogues.

Table 1. groups the operations (or in the case of NFS Version 2 and 3, RPC procedures) functionally for purposes of comparison. The comparison is a little unfair since the Network Lock Manager, Status Monitor and Mount protocol procedures needed by NFS Versions 2 and 3 are not shown. Significant changes occurred to data structures and semantics of existing operations, some of which are described below.

The NFS Version 4 introduction of the stateful operations OPEN and CLOSE is another major structural difference. NFS Versions 2 and 3 were essentially stateless. LOOKUP was the closest analogue to an open operation in earlier versions of NFS. However, a LOOKUP procedure did not create state on the server. The introduction of the stateful OPEN and CLOSE operations is required to ensure atomicity of share reservations as defined for Windows file sharing [CIFS], and to support exclusive creates. Additionally, the OPEN operation provides the server the ability to delegate authority to a client, allowing aggressive caching of file data and locking state.

The CREATE operation of NFS Version 4 differs from an NFS Version 3 CREATE in that it is only used to create special file objects such as symbolic links, directories, and special device nodes. To ensure correct share reservation semantics, the regular file CREATE procedure of NFS Versions 2 and 3 is replaced by the NFS Version 4 OPEN operation (with a *create* bit set). CREATE and REMOVE in NFS Version 4 subsumes the MKDIR and RMDIR directory functionality of prior versions of NFS.

NFS Version 4 servers depart from the semantics of previous NFS versions in requiring LOOKUP requests to cross mount points on the server. In NFS Version 4, a LOOKUP is very simple. It only sets the current filehandle to point at the file object resolved. Attributes (including the filehandle itself) can be obtained with a subsequent GETATTR operation in the same COMPOUND procedure. Additionally, as defined in WebNFS, LOOKUP takes a multi-component pathname.

Previous versions of NFS assigned special semantics to the directory entries "." and ".." NFS Version 4 assigns no special meaning to these names, and requires the client to explicitly use the LOOKUPP operation to obtain the filehandle of a parent directory.

The Weak Cache Consistency information (pre- and post-operation attributes) of NFS Version 3 has been removed. Instead, CREATE, LINK, OPEN, REMOVE, and RENAME return a data structure *change_info* (typically implemented as a modified time) that provides information on whether the directory underlying the object changed during the operation. The client can use this information to decide whether to flush cached directory information in the face of concurrent client modifications.

Underlying the NFS Version 4 protocol is mandated strong security via an extensible authentication architecture built on GSS-API. The client determines the authentication type required for a given file’s access using the `SECINFO` operation. Initial authentication flavors supported in this framework are Kerberos and LIPKEY. NFS Version 4 defines a Windows NT and Unix-compatible access control model.

The NFS Version 3 directory scanning operation `READDIRPLUS` procedure was dropped, and its functionality of providing attributes with each directory entry (including the filehandle) is now supported by the `READDIR` operation. This “bulk `LOOKUP`” functionality is used to initialize attribute caches when first scanning directories to reduce latency introduced by a (now unneeded) subsequent stream of `LOOKUP` operations.

Attributes of the file system underlying a file system object (for example, file system free space) exist in NFS Version 4 as attributes of the file system object itself. This replaces the NFS Version 3 procedures `FSSTAT`, `FSINFO` and `PATHCONF` with an NFS Version 4 `GETATTR` operation of the desired attributes.

As in NFS Version 3, file access rights are checked on the server, not the client. However, in NFS Version 4, file access rights are checked as part of an explicit `OPEN` operation instead of the NFS Version 3 `LOOKUP` and `ACCESS` procedure sequence. In retrospect, the introduction of the separate `ACCESS` procedure to handle access checking in conjunction with an initial `LOOKUP` (associated with a client application opening a file) hurt performance by introducing further unwanted network latency. The explicit `ACCESS` operation is retained in NFS Version 4 to support the UNIX *`access(2)`* programming interface which does not require the file to be opened.

NFS Version 4 supports file system replication and migration, but details of server-to-server file system transfers are undefined.

Generalized file attributes are extensible through the addition of named attributes.

File names in operations that use them are UTF-8 encoded UCS strings [UTF8] to enable internationalization.

Table 1. NFS operations by version - at a glance

Version 2	Version 3	Version 4
NULL	NULL	NULL
Compound operations		
		COMPOUND
		NVERIFY
		VERIFY
		Reserved Operation 2
OPEN/CLOSE operations		
		OPEN
		OPENATTR
		OPEN_CONFIRM
		OPEN_DOWNGRADE
		CLOSE
Delegation operations		
		DELEGPURGE
		DELEGRETURN
		SETCLIENTID
		SETCLIENTID_CONFIRM
Client callback procedures for delegation		
		CB_NULL
		CB_COMPOUND
		CB_GETATTR
		CB_RECALL
Locking operations		
		LOCK
		LOCKT
		LOCKU
		RENEW
Filehandle operations		
		PUTPUBFH
		PUTROOTFH
		GETFH
		RESTOREFH
		SAVEFH
Security operations		
	ACCESS	ACCESS
		SECINFO
Traditional file operations		
LOOKUP	LOOKUP	LOOKUP
		LOOKUPP
GETATTR	GETATTR	GETATTR
SETATTR	SETATTR	SETATTR
LINK	LINK	LINK
READDIR	READDIR	READDIR
	READDIRPLUS	
READLINK	READLINK	READLINK
CREATE	CREATE	CREATE
MKDIR	MKDIR	
	MKNOD	
REMOVE	REMOVE	REMOVE
RMDIR	RMDIR	
RENAME	RENAME	RENAME
SYMLINK	SYMLINK	
READ	READ	READ
WRITE	WRITE	WRITE
	COMMIT	COMMIT
STATFS	FSSTAT	
	FSINFO	
	PATHCONF	
Never implemented		
ROOT		
WRITECACHE		
18 ops	22 ops	42 ops

3. File system model and sharing

A *file system* is an implementation of a single file name space containing files, and provides the basis for administration and space allocation. Associated with each file system is a *file system identifier*, or *`fsid`*, which is a 128-bit per-server unique

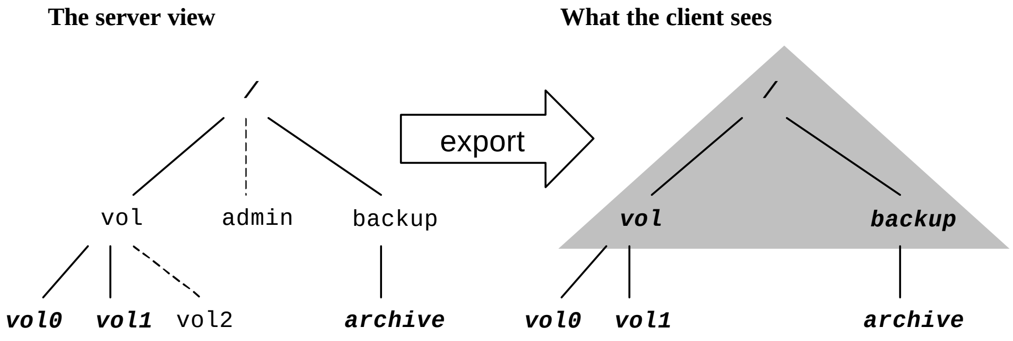

Figure 1. The Pseudo-file system

■ The Pseudo-file system constructed by the server

identifier. A *file* is a single named object consisting of data and attributes, residing in a file system. A *regular file* is a simple byte stream – not a directory, symbolic link or special (device) file. A *filehandle* uniquely identifies a file on a server (and consequently in a file system on that server).

In all versions of NFS, a server contains one or more file systems that are exported to clients.

However, in NFS Version 4, a server presents a single seamless view of all the exported file systems to a client. A client can move up and down the name space, traversing directories without regard to the structure of the file systems on the server. The client can notice file system transitions on the server by observing that the *`fsid`* changes. Removing the requirement that a client mount the different exported file systems of a server separately rendered the NFS Version 2 and file system attribute procedures useless. The server now reports file system attributes such as the file system free space for the specific file system underlying a file system object.

The client accesses the exported file systems of the server by using the `PUTROOTFH` operation to load the filehandle of the root of the file systems tree into the current file handle for subsequent operations.

3.1. Exporting file systems

An NFS Version 4 server exports file systems similarly to prior versions of NFS. The export operation makes available only those file systems, or portions of file systems, desired to be shared with clients. Further, the export operation allows the administrator to specify the acceptable security flavors by which a client can access a given exported file system.

3.2. Pseudo-file systems

> The subject may appear an insignificant one, but we shall see that it possesses some interest.
> Darwin, *The Formation of Vegetable Mould…*

On most operating systems, the name space describes the set of available files arranged in a hierarchy. When a system acts as a server to share files, it typically shares (or "exports") only a portion of its name space, excluding perhaps local administration and temporary directories.

Consider a file server that exports the following directories:

```
/vol/vol0
/vol/vol1
/backup/archive
```

The server provides a single view of the exported file systems to the client as shown in Figure 1.

In NFS Version 4, a server's shared name space is a single hierarchy. In the example illustrated in Figure 1., the export list hierarchy is not connected. When a server chooses to export a disconnected portion of its name space, the server creates a *pseudo-file system* to bridge the unexported portions of the name space allowing a client to reach the export points from the single common root. A pseudo-file system is a structure containing only directories, created by the server having a unique *`fsid,`* that allows a client to browse the hierarchy of exported file systems.

The client’s view of the pseudo-file system is limited to those paths that lead to exported file systems. Because `/vol/vol2` and `/admin` are not exported in this example, they do not appear to the client during browsing operations as shown in the client’s view in Figure 1.

4. The `COMPOUND` procedure

NFS is an RPC-based distributed application. Previous versions of the NFS protocol were defined only in terms of remote procedure calls. This approach has the significant limitation that each RPC call defines a single request-response transaction between the client and server incurring a minimum network latency cost for each transaction. A client may actually be required to transmit a series of related requests on the network to accomplish a single client operation.

NFS Version 4 introduces the `COMPOUND` RPC procedure. The `COMPOUND` procedure groups multiple related *operations* into a single RPC packet. The RPC response to a `COMPOUND` procedure contains the replies to all the operations. Because of the simplicity of error handling (evaluation of the operations stops on first error), it *may* be unwise to attempt grouping unrelated operations into a single `COMPOUND` procedure.

4.1. An example

The following denotations represent NFS transactions in this paper. We represent a simple client RPC request in NFS Versions 2 and 3 by:

```
→ LOOKUP
```

We represent a simple server RPC response by:

```
← LOOKUP OK
```

We represent a `COMPOUND` client RPC request in NFS Version 4, which contains one or more operations, by:

```
⇒ PUTROOTFH
  LOOKUP
  GETFH
```

We represent a `COMPOUND` server RPC response in NFS Version 4, which contains one or more replies to previous operations, by:

```
⇐ PUTROOTFH OK
  LOOKUP OK
  GETFH OK
```

Note the direction of the arrows in each example.

We represent side effects of operations in NFS Version 4 in the following way:

```
⇐ PUTROOTFH OK ↓CURFH
```

to suggest storing the current state of evaluation of the `COMPOUND` procedure.

The following example illustrates not only the use of the `COMPOUND` procedure, but also the elimination of the Mount protocol and *portmapper* through the use of a well-known port (2049). Consider the traffic generated over the network by the following simple commands on a Solaris (UNIX) system:

```
mount bayonne:/export/vol0 /mnt
dd if=/mnt/home/data bs=32k count=1
            of=/dev/null
```

to mount a remote file system and read the first 32KB of the file.

Using NFS Version 3, the following sequence results:

```
→ PORTMAP C GETPORT (MOUNT)
← PORTMAP R GETPORT
→ MOUNT C Null
← MOUNT R Null
→ MOUNT C Mount /export/vol0
← MOUNT R Mount OK
→ PORTMAP C GETPORT (NFS)
← PORTMAP R GETPORT port=2049
→ NULL
← NULL
→ FSINFO FH=0222
← FSINFO OK
→ GETATTR FH=0222
← GETATTR OK
→ LOOKUP FH=0222 home
← LOOKUP OK FH=ED4B
→ LOOKUP FH=ED4B data
← LOOKUP OK FH=0223
→ ACCESS FH=0223(read)
← ACCESS OK (read)
→ READ FH=0223 at 0 for 32768
← READ OK (32768 bytes)
```

The sequence above contains the simplified output from an actual network trace. Each of the 11 pairs of request and response transactions represents a network round trip.

The following traffic would result in an NFS Version 4 network:

```
⇒ PUTROOTFH
  LOOKUP “export/vol0”
```

```
   GETFH
   GETATTR
⇐  PUTROOTFH OK ↓CURFH
   LOOKUP OK ↓CURFH
   GETFH OK
   GETATTR OK
⇒  PUTFH
   OPEN “home/data”
   READ at 0 for 32768
⇐  PUTFH OK ↓CURFH
   OPEN OK ↓CURFH
   READ OK (32768 bytes)
```

Although an implicit “mount” occurred, the SECINFO is not needed. The SECINFO operation is only needed when the client attempts access with the wrong security flavor and a NFS4ERR_WRONGSEC error is returned.

In the above example, the number of round trip requests for the same application in NFS Version 4 compared to prior versions is reduced from 11 to two request and response transactions.

A client can aggressively use the COMPOUND procedure to pre-load caches on initial reference. Callaghan prototyped a super LOOKUP in the Java client that emitted the following sequence on initial access:

```
⇒  PUTFH
   LOOKUP “image”
   GETFH
   GETATTR
   ACCESS
   READ at 0 for 32768
```

A client may be restricted through its file system architecture in the generation of complex sequences.

4.2. Properties of the COMPOUND procedure

The set of operations in a COMPOUND procedure is not atomic. That is, no assumptions can be made as to whether conflicting operations occurred to file system objects referenced in a COMPOUND procedure between successive operations.

Error handling is simple on the server. If an operation fails in a COMPOUND procedure, evaluation halts and the remaining operations are not processed. Replies are returned to the client up to and including the error reply for the failed operation.

Most operations require a filehandle and may produce a filehandle as a result. In NFS Version 4, however, most operations do not explicitly have a filehandle as an argument or result. Instead, the server maintains a single filehandle, the current filehandle, as the argument for those operations. To initially load the current filehandle the operations PUTFH, PUTROOTFH and PUTPUBFH are used. The SAVEFH operation stores an additional filehandle for use by the LINK and RENAME operations (which require two filehandles for the source and target directories). RESTOREFH retrieves the saved filehandle.

5. Multi-component LOOKUP

The multi-component LOOKUP allows a client to resolve a full path name in one operation. The client can detect mount point crossing by inspecting the *fsid* of the directory containing the object to be resolved and the *fsid* of the resolved object. A UNIX client that detects a mount point crossing can explicitly mount the separate file systems for reporting space allocation information to the user. A Java client doesn’t care.

A client can enter partial information for intermediate nodes, filling in details with additional operations to the server when referenced.

Consider the following example. The command:

```
ls /export/home/beepy
```

can result in the following initial sequence of operations:

```
⇒  PUTROOTFH
   LOOKUP “export” “home” “beepy”
   GETFH
   GETATTR
```

NFS Version 4 requires that symbolic links be resolved relative to the client’s name space. If beepy is a symbolic link, the LOOKUP will fail with an NFS4ERR_NOTDIR error:

```
⇐  PUTROOTFH OK ↓CURFH
   LOOKUP FAILED
```

The client must then resolve the pathname component by component – still doable in a single COMPOUND procedure. An equivalent sequence – still in a single COMPOUND request is:

```
⇒  PUTROOTFH
   LOOKUP “export”
   GETFH
   GETATTR
   LOOKUP “home”
   GETFH
   GETATTR
   LOOKUP “beepy”
   GETFH
   GETATTR
```

The benefit of this sequence, besides loading the client attribute cache for interior directory nodes, is that the client receives a partial result from which to proceed in final pathname resolution:

```
⇐ PUTROOTFH OK ↓CURFH
   LOOKUP OK ↓CURFH
   GETFH OK
   GETATTR OK
   LOOKUP OK ↓CURFH
   GETFH
   GETATTR OK
   LOOKUP FAILED
```

This optimization would require more sophisticated error recovery on the client.

6. Important data structures

The following data structures are fundamental building blocks of NFS Version 4.

6.1. Filehandles

A *filehandle*, as in previous versions of NFS, is a per server unique identifier for a file system object that is opaque to the client. As in previous versions of NFS, filehandles that are equal refer to the same file system object. But no assumptions can be made by the client if the filehandles differ. In prior versions of NFS, procedures returned a filehandle explicitly in the results structure. In NFS Version 4, operations set an object called the *current filehandle* as a side effect, for use by subsequent operations in a single `COMPOUND` procedure. A client uses the `GETFH` operation to fetch the current filehandle.

There are two special filehandles: the *root* and the *public* filehandles. These filehandles are assigned to the current filehandle with the `PUTROOTFH` and `PUTPUBFH` operations. A client uses `PUTROOTFH` to gain initial access to the filehandle of the common root of all exported file systems on the server, as in the following sequence:

```
⇒ PUTROOTFH
   LOOKUP “export” “home”
   GETATTR

⇐ PUTROOTFH OK ↓CURFH
   LOOKUP OK ↓CURFH
   GETATTR OK
```

The public filehandle identifies the portion of the server name space used with WebNFS as described in [RFC2054, RFC2055]. Unlike the root filehandle, the public filehandle may be bound to an arbitrary file system object. It may be that the root and public filehandles are the same.

6.1.1. Persistent vs. volatile filehandles

In NFS Versions 2 and 3, filehandles returned by the server were persistent. The client could count on the filehandle always referring to same file. The server would typically generate an opaque persistent filehandle by including a unique `inode` number, the `inode`'s generation count, and device number (`fsid`) of the disk partition that the filehandle's object was allocated on. If the underlying file object was deleted and replaced with a file object of the same name, the change in generation count maintained by the server would result in a new filehandle being generated – and invalidating any existing filehandles held by clients. When the server received a request from the client that included a filehandle, it was straightforward to resolve the underlying file object from the device number and `inode` number.

This model worked well for most UNIX-based servers, but did not work for non-UNIX systems that relied solely on a file's pathname for identification, or for any local file system that did not have a persistent equivalent to a compact `inode` number (for example, the High Sierra file system for CD-ROMs).

NFS Version 4 introduces the concept of *volatile filehandles*. For volatile filehandles, a client must cache the mapping between path name and file handle, and regenerate the (possibly different) filehandle upon filehandle expiration. When a filehandle expires, the client gets an `NFS4ERR_FHEXPIRED` error on the next access and must flush any cached information that refers to that filehandle.

The intent is that volatile filehandles expire only upon certain events, such as:

- when an open file is closed
- when the file system the filehandle belongs to is migrated
- when a client renames a file in some file systems (as is the case with Linux NFS Version 2 and 3 servers today)

The weakest form of volatile filehandles allows expiration at any time. This can be risky for a client, such as when a second client removes a file, and creates a new one with the same name. A client that has the original file open would regenerate the volatile file handle and then access the new (unexpected) data resulting in corruption. Volatile file handles this weak are best reserved for isolated scenarios where a user knows they alone are accessing the file system or the file system is read only.

6.2. Client ID

A client first contacts the server using the `SETCLIENTID` operation, in which it presents an opaque structure identifying itself to the server, together with a verifier. The opaque structure uniquely identifies a particular client. A *verifier* is a unique, non-repeating 64-bit object generated by the client that allows a server to detect client reboots. On receipt of the client's identifying data, the server will return a 64-bit *`clientid`*. The *`clientid`* is unique and will not conflict with those previously granted, even across server reboots.

The *`clientid`* is used in client recovery of locking state following a server reboot. A server after a reboot will reject a stale *`clientid`*, forcing the client to re-establish a *`clientid`* and locking state.

After a client reboot, the client will need to get a new *`clientid`* to use to identify itself to the server. When it does so, using the same identity information and a different verifier, the server will note the reboot and free all locks obtained by the previous instantiation of the client.

6.3. State ID

A *stateid* is a unique 64-bit object that defines the locking state of a specific file.

When a client requests a lock, it presents a *`clientid`* and a unique-per-client lock owner identification to identify the lock owner. A *lock owner* is the thread id process id or other unique identifier for the application owning a particular lock on a client. On granting the lock, the server returns a unique 64-bit object, the *`stateid,`* to be used by the client in subsequent operations as a shorthand notation to the lock owner information now stored on the server. This not only prevents another client from accessing a file in a manner that conflicts with the locks that are held, it also prevents unwanted replay by a broken router of I/O requests with a previous *`stateid`* (which can corrupt the locking state). A side effect of the *`stateid`* is that it also provides a positive acknowledgement to the server that all locks held by the client are still valid, allowing an active client to avoid explicit lease refresh.

7. `OPEN` and `CLOSE`

> To make vertue of necessite.
> Chaucer, *The Canterbury Tales*

Apart from the Network Lock Manager, NFS Versions 2 and 3 were essentially stateless protocols (other than for necessarily persistent file objects on the server). This presented problems in implementing the functions of file locking and file sharing (with Windows operating system semantics) required for correct operation of client applications. Further, aggressive client caching with well-defined semantics was impossible.

NFS Version 4 introduces an `OPEN` operation that provides an atomic operation for file lookup, creation and share reservation. To provide correct share reservation semantics, an NFS Version 4 client must use the `OPEN` to obtain the initial filehandle for a file. Windows requires the ability to atomically create a regular file with a share reservation - the `OPEN` operation (with a *create* bit set) provides these semantics.

The `CLOSE` operation releases the state accumulated by an `OPEN`.

8. Caching and delegation

NFS has never implemented distributed cache coherence, nor supported concurrent write-sharing in the absence of locking, and NFS Version 4 does not change that. However, client-side caching is essential to good performance. NFS has always supported client caching – albeit with restrictions and a loss of strict cache coherence.

NFS Version 4 differs from previous versions of NFS by allowing a server to *delegate* specific actions on a file to a client to enable more aggressive client caching of data and to allow caching of locking state for the first time. A server cedes control of file updates and locking state to a client for the duration of a lease via a *delegation*.

8.1. Client-side caching

NFS Version 4 file, attribute, and directory caching resembles that in previous versions. Attributes and directory information are cached for a duration determined by the client. At the next use after the end of a predefined timeout, the client will query the server to see if the file system object has changed.

When opening a regular file, the client validates cached data for that file. The client queries the server to determine if the file has changed. Using this information, the client determines if the data cache for the file should be kept or flushed. When the file is closed, the client writes any modified data to the server. This technique of close-to-open consistency [Pawlowski94] has provided sufficient consistency for most applications and users.

If an application wants strict serialized access to file data, share reservations or file locking of specific file data ranges should be used.

Previous versions of NFS avoided the use of client-side data caching when record locking was in effect. Version 4 defines rules that allow data caching during locking while maintaining cache integrity. `COMPOUND` operations allow fetching the modified time for a file after obtaining a record lock, without additional latency, simplifying the implementation of these rules.

8.2. Open delegation

In NFS Version 4, when a file is only being referenced by a single client, responsibility for handling all of the `OPEN` and `CLOSE` and locking operations may be *delegated* to the client by the server. This eliminates `OPEN` and `CLOSE` requests, allows locking requests to be resolved locally, and eliminates normal NFS client periodic cache consistency checks – reducing over-the-wire traffic and associated latency. Since the server on granting a delegation guarantees the client that there can be no conflicting `OPEN` operations, the cached data is assumed valid. The server may also allow the client to retain modified data on the client without flushing at `CLOSE` time, if it can be guaranteed that sufficient space will be reserved on the server ensuring that subsequent `WRITE` operations will not fail due to lack of space.

When many clients share a file, in the absence of writing, the server may delegate the handling of read-only `OPEN` operations to multiple clients. This allows `OPEN` and `CLOSE` operations to be avoided. Since such a delegation will only persist in the absence of writers, the client is assured that cached data is valid, without periodic consistency checks to the server.

A lease is associated with a delegation. If the lease expires, the delegation will be revoked, just as with locks.

Delegation allows common patterns of limited sharing and read-only sharing to be dealt with efficiently, avoiding extra latency associated with frequent communication with the server. When these patterns no longer obtain, the delegation is revoked and normal client-side caching logic is used.

8.3. Client callbacks

Revocation of delegation requires the client to update state on the server to reflect changes made by the client as part of the delegation, and then return the delegation to the server. Upon return of the delegation, the server will centrally manage `OPEN` and locking operations.

Revocation is accomplished by making a callback. A callback is an RPC from the server to the client to inform it of server actions. Because callbacks may have problems transiting firewalls, callbacks are not required for proper operation of the protocol. A server will test whether a client can respond to callbacks by making an initial `CB_NULL` request to the client. If a client fails to respond, the server will not delegate authority to that client.

8.4. Delegations vs. Windows OpLocks

Delegation has many similarities to Opportunistic Locks (OpLocks) used by CIFS [Borr], and was inspired by the benefits which that mechanism provides. The differences between them reflect the different histories of the two protocols and the problems they solve.

Delegations differ from OpLocks in that a delegation is an optimization that is solely up to the server while OpLocks are requested by the client. The ability to delegate depends on a network configuration that the server can verify, plus specific sharing patterns.

When OpLocks are lost or not available, CIFS sends all operations to the server while NFS can fall back to its standard modes of (periodically checked) client-side caching when delegations are unavailable. This makes delegation less critical a feature, but delegation – when possible – provides many performance benefits, particularly when applications are doing frequent file locking operations.

Delegations can persist beyond the `OPEN` operation which gave rise to them, like Batch OpLocks in Windows, allowing subsequent `OPEN` operations to be cached on the client. Delegated files can be shared by many applications on a single client with the proper state for all transferred back to the server upon delegation revocation.

9. Locking

NFS Version 4 locking is similar to the adjunct Network Lock Manager (NLM) protocol used with NFS Versions 2 and 3, but it is tightly coupled to the NFS protocol to better support different operating system semantics and error recovery.

A major failing of the NLM protocol was the detection and recovery of error conditions. The design assumed that the underlying transport was reliable and preserved order. With NLM, an unreliable network easily resulted in orphan locks on the server. In addition, if a client crashed and never recovered, locks could be permanently abandoned, preventing any other client from ever acquiring the lock.

9.1. Leases

The key change in NFS Version 4 locking is the introduction of leases for lock management.

A *lease* is a time-bounded grant of control of the state of a file, through a lock or delegation, from the server to the client. During a lease interval a server may not grant conflicting control to another client. A lease confers on the client the right to assume that a lock granted by the server will remain valid for a fixed (server-specified) interval and is subject to renewal by the client. The client is responsible for contacting the server to refresh the lease to maintain the lock.

The expiration of a lease is considered a failure in the communications between the client and the server, requiring recovery. If the lease interval expires without a refresh from the client, the server assumes the client has failed and may allow other clients to acquire the same lock. If the server fails, on reboot the server waits a duration equal to a lease interval for clients to reclaim the locks that they may still hold, before allowing any new lock requests.

Leases or token-based state management exists in several distributed file systems [Kazar90, Macklem94, Srinivasan].

Most operating systems demand that a lock is irrevocable once acquired by an application. Unlike leases used to manage cache consistency where leases are kept short to prevent unnecessary delays in normal operations, the lock lease intervals can be substantially longer, reducing the number of lease refreshes required, one of the primary drawbacks of a lease-based protocol.

In addition, the lease protects against a loss of the locking state by the client. A client exists in two states: either all the locks held from a given server are correct or all are lost. A refresh of *any* lock by the client validates all locks held by the client to a particular server. This reduces the number of lease refreshes by the client from one per lock each lease interval, to one per client each lease interval, eliminating another drawback of a lease-based protocol.

9.2. Mandatory locking

Better interoperability with non-Unix operating systems is an important goal of NFS Version 4. A key feature of the Windows operating systems, and available on some Unix operating systems, is mandatory locking - the ability to block I/O operations by other applications on a file that contains a record lock. The NLM protocol provided only for advisory locking which allowed cooperating applications to synchronize I/O operations, but did not block other applications from performing I/O operations to the file. To handle this additional semantic, the concept of a *`stateid`* was added to NFS Version 4.

9.3. Share reservations

To provide better interoperability, NFS Version 4 fully supports share reservations. A *share reservation* grants a client access to open a file and the ability to deny other clients open access to the same file. A share reservation is similar to a file or record lock, except that its granularity is always on an entire file, and its lifetime equals the duration of the file open. Normal file and record locks do not interact with share reservations - a share reservation is distinct from a record lock in that it only governs the ability to open a file.

For example, an application may open a file for read access and acquire a share reservation denying other subsequent opens that request write access. The NLM protocol supported clients that use this style of lock to cooperate amongst themselves, but it did not enforce it between non-cooperating clients. More importantly, a share reservation was not tied into other operations that implicitly open a file, such as `CREATE`. This exposes a race condition where one client could create a file, and before the second operation to acquire a share lock denying other clients access is received, another client acquires a conflicting reservation. The addition of an explicit `OPEN` operation correctly supports share reservations.

The `OPEN` operation takes as parameters the traditional desired access of read or write and, in addition, allows the application to deny read or write access to other applications. The server response contains a *`stateid`* that is used by the server to enforce share reservations. A corresponding `CLOSE` operation allows a client to free the held share reservations.

9.4. Sequence IDs

The most problematic part of network locking is dealing with lock requests that arrive out of order or are replayed. As an example, a client issues a sequence of lock, unlock, and lock requests. If a misbehaved router replays a previous unlock request other clients may acquire a conflicting lock and corrupt data. The RPC layer's transaction id will defend against many of these replay errors, but the server duplicate request caches are frequently not large enough to handle even modest windows of time [Juszczak]. Locking requests by an application in virtually all operating systems are strictly ordered, defining a well-known state of the file. This requires that a server in a distributed file system also process the locking requests in the required strict order.

NFS Version 4 adds to every lock and unlock operation a monotonically increasing sequence number to provide at-most-once semantics. The server maintains for each lock owner the last sequence number and the response sent. If a second request is received with the last sequence number, the response is replayed under the assumption that the previous response was lost. If an earlier sequence number is received then an error is returned as it must be a replay of a previously received response. A sequence number beyond the next sequence number is also rejected.

10. Attributes

The attribute model for NFS Version 4 is different from prior versions in providing a mechanism for extensibility. NFS Version 4 defines three types of attributes:

- Mandatory
- Recommended
- Named

Mandatory and recommended attributes are defined in terms of a bit vector to allow efficient implementation of operations that return or manipulate those attributes. A mask defines those attributes that are to be manipulated – with unset bits representing attributes to be ignored.

10.1. Mandatory attributes

Mandatory attributes represent the baseline attributes that must be supported or emulated by every implementation. Mandatory attributes include:

- Object type
- Filehandle expiration type
- Change indicator
- Size
- UNIX LINK support
- UNIX SYMLINK support
- *`fsid`*
- Lease duration

10.2. Recommended attributes

The recommended attributes include:

- ACL
- Archive bit
- Case insensitive
- Case preserving
- Change owner restricted
- No file name truncation beyond maximum
- Filehandle
- File ID
- Hidden
- Maximum file size
- Maximum number of links
- Maximum filename size
- Maximum read size
- Maximum write size
- MIME type
- UNIX mode bits
- Owner string
- Group string
- Modify time
- Create time
- Access time
- Space available to user
- File system free space
- File system total space

- Space used by object

ACLs are a special recommended attribute and are described below in the section on security.

10.3. Named attributes

NFS Version 4 introduces named attributes for the first time. The model for named attributes is simple. Associated with each file system object is a hidden directory containing all its named attributes. The data associated with the named attributes is an uninterpreted (by NFS) stream of bytes. A client would access named attributes in the following way:

- The `OPENATTR` operation sets the current filehandle to the named file attribute directory for the file object
- `READDIR` and `LOOKUP` operations retrieve file handles for the various named attributes associated with the original file system object.

Named attributes require support on the server, and are a feature of common file systems like Windows NTFS.

11. Security model

NFS relies on the underlying security model of RPC for its security services. A variety of authentication flavors have been defined for use by NFS going back to the Diffie-Hellman public key authentication scheme defined for use with NFS Version 2 [Taylor]. However, no model other than the weakly authenticated UNIX permission scheme was ever widely adopted, limiting the use of NFS in hostile networks (for example, universities).

While NFS Version 3 introduced the `ACCESS` procedure in part to support flexible ACL-based access control, no agreement was ever reached on a common ACL format to allow heterogeneous access control.

In the area of security, NFS Version 4 improves over NFS Versions 2 and 3 by:

- mandating the use of strong RPC security flavors that depend on cryptography
- negotiating the security used via a system that is both secure and in-band
- using character strings instead of integers to represent user and group identifiers
- supporting access control that is compatible with UNIX and Windows
- removing the Mount protocol.

11.1. GSS-API framework

NFS is based on ONCRPC [RFC1831] and leverages its security architecture, recently bolstered by the addition of a security flavor based on the Generic Security Services API (GSS-API), called RPCSEC_GSS [RFC2203]. RPCSEC_GSS is a *security flavor* allocated under the same flavor number space as the commonly used AUTH_SYS flavor; AUTH_SYS is flavor number 1, RPCSEC_GSS is flavor number 6. The flavors between 1 and 6 represent efforts such as [Taylor] to improve RPC security that became obsolete due to advancements in attacks based on brute force [EFF] and better cryptanalysis [LaMacchia].

RPCSEC_GSS differs from AUTH_SYS and other traditional flavors in two ways:

- First, RPCSEC_GSS does more than authentication. It is capable, albeit at considerable expense of CPU execution time [Eisler96], of performing integrity checksums and encryption of the entire body of the RPC request and response. Hence, RPCSEC_GSS is a security flavor, and not just an authentication flavor.
- Second, because RPCSEC_GSS simply encapsulates the GSS-API messaging tokens – it merely acts as a transport for mechanism-specific tokens for security flavors like Kerberos. Adding new security mechanisms (as long as they conform to GSS-API) does not require re-writing significant portions of NFS or any other ONC RPC-based application.

11.2. Mandated strong security

All versions of NFS are capable of using RPCSEC_GSS. The difference is that while an implementation can claim conformance to NFS Versions 2 and 3 without implementing support for RPCSEC_GSS, a conforming NFS Version 4 implementation must implement RPCSEC_GSS. Furthermore, conforming NFS Version 4 implementations must implement security based on Kerberos Version 5 (in this paper, simply Kerberos) [RFC1510] and LIPKEY [Eisler00], each of which are GSS-API conforming security mechanisms.

11.2.1. Kerberos versus LIPKEY

Kerberos divides user communities into realms. Each realm has an administrator responsible for maintaining a database of principals (users). Each realm has one master Key Distribution Center (KDC), and one or more slave KDCs that give users tickets to access services on specific hosts in a realm. Users in one realm can access services in another realm, but it requires the cooperation of the administrators in each realm to develop trust relationships and to exchange per-realm keys. Hierarchical organization and authentication of realms can reduce the number of inter-realm relationships.

Kerberos has been used on other distributed file systems, such as the Andrew File System [Howard], the Open Software Foundation's Distributed File System [Kazar], NFS Version 2 and 3 [RFC2623], and most recently, Microsoft's CIFS (Windows 2000) [Microsoft00]. Kerberos is an excellent choice for enterprises and work groups operating within an Intranet, since it provides centralized control, as well as single sign on to the network.

But NFS Version 4 is also designed to work outside of intranets on the global Internet. Kerberos does not work well on the Internet. The user would need the cooperation of his local system administrator to negotiate a trust relationship with the administrator of the remote realm.

The Low Infrastructure Public Key (LIPKEY) system provides an SSL-like model and equivalent security for use on the Internet. LIPKEY is a GSS-API security mechanism using a symmetric key cipher and server-side public key certificates.

The LIPKEY user experience is similar to that of HTTP over the Secure Sockets Layer (SSL). A user is prompted for a user name and password. These are encrypted with a 128-but symmetric session key. The session key is encrypted with the server’s public key and all are sent to the server. The client authenticates the server by comparing the latter’s certificate with a list of trusted Certification Authorities.

11.3. Why not SSL?

NFS Version 4 does not use SSL [SSL]. The primary issue with SSL is that it does not work over connectionless protocols like UDP, whereas NFS does. The second problem is that as mentioned previously, RPC has its own security architecture – it is unclear how to cleanly merge SSL and RPC security. RPCSEC_GSS provides equivalent security, yet is compatible with flavors like AUTH_SYS.

11.4. Kerberos in Windows 2000 vs. UNIX

As noted in [Ts'o], Windows 2000's Kerberos has some incompatibilities with most other Kerberos implementations. Windows 2000 uses the pre-authentication field in Kerberos messages to encode a proprietary representation of the privileged access groups (PAGs) that a user belongs to. This way, when a Kerberized client talks to a Kerberized-server, the server knows immediately what groups the user belongs to. This is both an efficient and non-interoperable scheme, which is exacerbated by no published documentation on the format of the PAG list, and what the PAG entries mean.

Most Kerberized servers outside of Windows 2000 would do something different. For example, an NFS server in the UNIX space would map the principal name to the UNIX user identifier, and the UNIX user identifier to the list of groups associated with the user. It is no less efficient to do it this way, because it is possible to compute the mappings upon user addition to the directory services domain that the NFS server lives in. This approach also has the virtue of being completely inter-operable with non-UNIX clients.

The effect of PAGs on NFS is that if a Windows 2000-based NFS Version 4 client or server uses PAGs, then it will not interoperate with a non-Windows 2000-based server or client. Otherwise, there are no issues with the Windows 2000 and non-Windows 2000 nodes on the network sharing the same Kerberos key space.

11.5. Negotiating security

NFS Version 2 had no way to negotiate security, which meant that if an NFS server exported a file system with something other than AUTH_SYS, there was no way for it to tell the client. Unless the client mounted the file system with an explicit mount option for different security, the mount attempt would fail.

NFS Version 3 enhanced the Mount protocol to include a list of security flavors that the client could use to mount the file system. The problem with this approach is that the Mount protocol itself was not secure. While in theory, the Mount protocol could use RPCSEC_GSS, in practice, Mount servers were not required to support RPCSEC_GSS.

NFS Version 4 deals with negotiation of security by including a new `SECINFO` operation that allows a client to ask what security the server requires for a given file object. The SECINFO operation's arguments and results are secured using one of the mandatory security flavors. The results of a SECINFO call define the RPC security flavors that should be used, and for each flavor any required additional information. For example, if SECINFO specifies that AUTH_SYS can be used, no additional information is needed. However, if SECINFO specifies to use RPCSEC_GSS, because RPCSEC_GSS is merely a security mechanism switch more information is needed. The client and server will then negotiate the Object Identifier of the GSS-API mechanism, what quality of protection to use, and whether to use authentication, integrity (checksummed arguments and results), or privacy (encrypted arguments and results – full user data encryption).

11.6. String identifiers

NFS Versions 2 and 3 represented users and groups via 32 bit integers. The NFS protocol uses user and group identifiers in the results of a get attribute (GETATTR) operation and in the arguments of a set attribute (SETATTR) operation. Using integers to represent users and groups requires that every client and server that might connect to each other to agree on user and group assignments. Not only is this impractical across the Internet, but problematic for some large enterprises. Some feel that a secondary issue is that 32 bits to represent users is not large enough.

NFS Version 4 represents users and groups in the form:

user@domain

or

group@domain

where domain represents a registered DNS domain, or a sub-domain of a registered domain. By leveraging the global domain name registry and delegating user and group identifier control, NFS Version 4 does not require IANA to develop yet another global registry to guarantee uniqueness.

One issue with using string names, instead of integers, is that UNIX systems like Solaris will still be using integers in the underlying file systems stored on disk. This requires mapping string names to integers and back. Since NFS clients and servers have done something similar with security flavors like RPCSEC_GSS and AUTH_DH [Taylor] that use string names for principals and not integers, we did not see a risk from removing integer based identifiers from the protocol.

11.6.1. UUIDs

We did consider Universal User Identifiers (UUIDs) instead of strings. However, UUIDs still have the translation issue, since they are 128 bits long versus 32 bits for UNIX identifiers. Furthermore, in situations where a client receives a GETATTR result with an untranslatable identifier, it was felt that a string like ted@eisler.com would be more useful than a string of 128 bits. We anticipate that UNIX implementers might consider adding a *stat(2)* system call variant that returns the file system's native string representations if available.

11.7. Access Control Lists

An Access Control List, or ACL, is simply a list that describes which users and groups get access to a file with what type of access (for example, read versus write). NFS Versions 2 and 3 do not have support for an ACL attribute, although there are several proprietary protocols for manipulating ACLs over NFS based on the POSIX Draft ACL specification. Such ACL support never saw wide use, perhaps due to the proprietary nature of the protocols and that the POSIX specification was never standardized.

NFS Version 4 includes ACL support based on the Windows NT model and not the POSIX model. The reasons are that compared to the POSIX model, the NT model is both richer, and widely deployed.

The richness of the NT model is seen in that an Access Control Entry (ACE) within an ACL can be one of four types: ALLOW, DENY, AUDIT, or ALARM. ALLOW and DENY simply means the ACE allows or denies the specified access to the entity attempting access. AUDIT means if the entity in the ACE attempts the specified access, log the attempt. ALARM generates a system dependent alarm if the entity in the ACE attempts the specified access. The POSIX model does not support AUDIT and ALARM.

One major difference between the NT and POSIX ACL models prevents NT from being a strict superset of the POSIX. In the NT model, the first ACE in the ACL that denies or allows access corresponding to the principal, or the principal's group making the request, determines if access is allowed. In the POSIX model, there are two kinds of ACEs: user entries and group entries. In the POSIX model, the user identifier is checked against the user entries first, and if the access is not unambiguously granted or denied, then the user's group identifiers are each checked against the group entries in the ACL. We feel that in practice this subtlety is unimportant.

There do exist systems today with POSIX ACLs that are incompatible with the ACLs defined for NFS Version 4. An NFS Version 4 server on such a system could continue to compute a user's access to a file with an incompatible POSIX ACL per the POSIX draft. As long as the ACL on the file does not change, there is no issue. When a client changes the ACL via the `SETATTR` operation, the server can replace the incompatible POSIX ACL with an NFS Version 4 compatible ACL as long as it assures that:

- the resulting ACL is not more permissive than the pre-existing POSIX ACL
- the resulting ACL is not more permissive than what the client intended.

11.8. Removing the Mount protocol

Unlike NFS Versions 2 and 3, NFS Version 4 has no Mount protocol. As a byproduct, this closes a security hole. Suppose there exists an exported directory called

`/export/alice/safe/A.`

Suppose the permissions on

`/export/alice/safe`

do not allow anyone but `safe`'s owner, Alice, access, but the permissions on `/export/alice/safe/A` are wide open. An NFS Version 2 or 3 client would normally be allowed to get a filehandle for `/export/alice/safe/A` and mount it, thus allowing a second party wrongful access.

Since NFS Version 4 has no way to distinguish mount attempts from other accesses, any client but Alice that attempts to get a filehandle for `/export/alice/safe/A` will be denied.

12. Migration and replication

To improve availability, NFS Version 4 has added features to support file system migration and replication.

A file system can migrate to a new server and the clients notified of the change by means of a special error code. A client is informed of the new location by means of the *`fs_locations`* file attribute. It may then access the file system on the new server transparently to applications running on the client.

The *`fs_locations`* attribute may also designate alternate locations for a (read-only) file system. If a client finds a file system unresponsive or performing poorly, it may choose to access the same data from another location. If a server implementation is concerned about the persistence of filehandles in the face of migration, it can vend volatile filehandles. The client will re-`LOOKUP` open files using saved pathname components on switching to a new server.

13. Minor versioning

This is the second major revision of NFS. In the past, NFS has been extended by overloading the semantics of existing procedures – without recourse to a formal protocol revision. Unfortunately, this sometimes hurt interoperability. One goal of the NFS Version 4 effort was to provide a framework for minor versioning of the protocol to facilitate rapid, simple evolution.

Minor versioning is left mostly undefined in the base NFS Version 4 protocol. A Reserved Operation 2 exists to provide minor version negotiation in a future minor revision. The `COMPOUND` arguments also include a minor version field (currently 0). Via the reserved operation, a client will query the server for minor versions supported – negotiating capabilities in a similar fashion to today’s version binding in RPC. Minor version negotiation is client driven. A minor version 0 server (the current protocol definition) identifies itself as only supporting version 0 by returning `NFS4ERR_NOTSUPP` – operation not supported - on attempts to invoke Reserved Operation 2.

The base specification (minor version 0) has some recommended rules for future work groups on managing the creation of a minor version. For example, allowing extension through the addition of additional attributes, but avoiding deletion of attributes existing in previous minor versions.

14. Modifications for use on the Internet

In the area of suitability for the Internet, NFS Version 4 improves over NFS Versions 2 and 3 by:

- requiring TCP as a transport

- defining COMPOUND operation to reduce round-trip latency
- defining a global user identifier name space
- mandating strong security based on a public key scheme
- enabling operation through firewalls

14.1. TCP is mandatory

The NFS Version 4 specification requires that any transport used provide congestion control. The easiest way to do this is via TCP. By using TCP, NFS Version 4 clients and servers will be able to adapt to known frequent spikes in unreliability on the Internet [Martin].

14.2. Reduced round trip latency

As illustrated in the examples of section 4.1, the COMPOUND procedure enables clients to pack more operations in a single request, thus significantly reducing round trip latency.

14.3. Global user name space

As described in 11.6, user and group identifiers are string names allocated relative to DNS domain names. Because the identifiers are completely generic, with no bias toward UNIX, NT, or any other operating system, the consumer need not be impacted if the service provider changes platforms, nor is the service provider impacted if the consumer changes platforms.

14.4. Mandatory security

As described in 11.2., NFS Version 4 clients and servers must support LIPKEY, a public key scheme that has similar properties to SSL. Both SSL and LIPKEY share properties that make them suitable for the Internet, namely that customers and vendors can get together without prior establishment of complex trust relationships.

The e-commerce market place has proven to be quite dynamic. If another security technology replaces the simple public key approaches of SSL and LIPKEY, the flexibility of GSS-API will ease the introduction of this new security mechanism.

14.5. Firewall friendly

To access an NFS server, an NFS Version 2 or 3 client must contact the server's *portmapper* to find the port of the Mount server. It contacts the Mount server to get an initial file handle. Then it contacts the *portmapper* to get the port of the NFS server. Finally, the client can access the NFS server.

This creates problems for using NFS through firewalls, because firewalls typically filter traffic based on well known port numbers. If the client is inside a firewalled network, and the server is outside the network, the firewall needs to know what ports the *portmapper*, Mount server, and NFS server are listening on. The Mount server can listen on any port, so telling the firewall what port to permit is not practical. While the NFS server usually listens on port 2049, sometimes it does not. While the *portmapper* always listens on the same port (111), many firewall administrators, out of excessive caution, block requests to port 111, from inside the firewalled network to servers outside the network.

NFS Versions 2 and 3 are not practical to use through firewalls.

NFS Version 4 solves the issue by eliminating the Mount protocol, and mandating that the server will listen on port 2049. This means that NFS Version 4 clients do not need to contact the *portmapper*, and do not need to access services on floating ports, making firewall configuration as simple as configuration for HTTP.

15. A common Internet file system

> One ring to rule them all,
> Tolkien

NFS Version 4 lends itself to several applications on the Internet.

15.1. An open download protocol

The Internet is rapidly becoming the primary means for distributing large files containing installable software, documents, and multi-media. Most downloads use the File Transfer Protocol (FTP), or HTTP. For slow links, large file downloads have an almost certain chance of aborting, with no recourse for the user but to start over again. While NFS is designed to be a file access protocol, because NFS allows the clients to read files from arbitrary offsets, it is a superior file transfer protocol. If the TCP connection breaks due to timeout or other reasons, the client can simply re-connect and continue (transparently to the user). With the use of LIPKEY, the client and server can protect the transfers from third party eavesdropping or tampering.

15.2. Consumer backup and restore

The cost of disk space on personal computers seems to be approaching US$1 (or 1.04€ or ¥107) per gigabyte. With the capability to store more data, the odds of a user losing data are increasing. Outside the home, data management policies are in place to ensure that valuable data is not lost due to a failure in the storage system. These policies include backup of data to tertiary storage, and the use of redundant arrays of disks or file servers. Within the home, it is impractical to expect the average consumer to implement formal data management. While we are seeing the emergence of low-end appliances for storing data redundantly, that these appliances are co-located with the user's primary data violates the principle of having off-site backups.

Several web sites today provide file backup and restore services. By definition, these web sites are off site. As high bandwidth links like DSL and cable modem become available to users, it becomes increasingly practical to backup larger amounts of data, obviating the need for on-site backups at home.

So far, these services are based on HTTP and FTP, which suffer from the same problems as file download for large file transfer. Again, NFS Version 4, secured via LIPKEY, offers a superior approach, providing strong authentication and privacy.

15.3. The Internet disk

Combining high-bandwidth persistent connections like DSL with NFS Version 4 delegation and sophisticated caching allows one to envision a time when users will prefer that the master copies of their data always exist on the service provider - who can better deal with the complexity of reliable data management.

For example, in the morning, before work, the user can access his data, which results in a transparent download of a subset of it to local storage, and manipulate it locally. Before going to work, the user "saves" it. When the user arrives at work, he will be able to access the same version of the data he was working on at home, because either his NFS Version 4 capable desktop at home has synchronized its dirty cache with the server, or the server will revoke the delegation to gain access to the latest data. The user at the office will be blocked from accessing his data until the server has a consistent copy.

16. Future work

[Pawlowski] described several follow-on tasks for NFS Version 3. Of those tasks, NFS Version 4 addresses strong security, while it does not provide support for concurrent write sharing (though we introduce delegations for improved caching performance), nor does it support disconnected operation. Changes to the export model and allowing mount point crossing when browsing from a single server root partially address consistent name space construction.

Curiously missing from the analysis in 1994 is recognition of the growing importance of support for file sharing on the Internet – which the design NFS Version 4 strongly reflects.

Given that track record of predictions, let's take a stab at presenting expected future work in the NFS Version 4 space.

16.1. IETF standardization

At the time of this writing, the working draft of the NFS Version 4 protocol specification has been submitted to the Internet Engineering Steering Group for consideration as a Proposed Standard – the first formal step towards the goal of achieving Internet Standard acceptance [RFC2026]. Specifications intended to become Internet Standards evolve through a set of maturity levels known as the "standards track". These maturity levels - *Proposed Standard, Draft Standard,* and *Standard* - reflect movement through the IETF standards process. While achieving Proposed Standard designation does not require implementation experience, we chose to prototype the specification to prove out concepts.

The construction of two independent, interoperable conforming implementations based on the specification are required to achieve Draft Standard status. Some changes may occur between Proposed Standard and Draft Standard status, but these are not expected. A Draft Standard is normally considered to represent the final specification – any changes made to the protocol beyond this reflect specific (otherwise insoluble) problems. Internet Standard achievement follows widespread experience with the Draft Standard and its implementations.

16.2. Minor versioning

Details of minor version negotiation, and change coordination for minor versioning, remain for future versions of the working group. Reserved operation 2 provides the ability to evolve NFS Version 4. Some suggested rules for future efforts in minor versioning appear in the draft specification.

16.3. Performance

The reduction in network latency with the use of the COMPOUND procedure comes at the cost of additional complexity in operation coding and decoding on the client, and increased complexity in handling error returns. More experience is needed in this area to understand the costs.

The attribute model and the use of a bit mask to describe attributes of interest to be fetched by the client generated much discussion. The trade-off of possibly reduced work on the server in loading only those attributes of interest is pitted against the increased decode complexity (and branching) in the implementation to handle a variable attribute return. The costs of the attribute model will be explored during further implementation.

16.4. Migration and replication

A server in NFS Version 4 can inform a client when multiple copies of a file system exist, or when a file system has moved. The client uses this information to adapt to changing network conditions and file system relocation. This provides a framework for migration and replication.

NFS Version 4 does not address server-to-server file system migration protocols or the issues of maintaining replica consistency and migration atomicity. It remains for future working groups to define. Until then, vendor-specific solutions may arise.

16.5. Single system image or name space

[Kazar, Howard, Microsoft99] describe approaches to providing a shared consistent name space that hides server details of data location from users. An NFS Version 4 server hides some details of data location by presenting a per-server single image of all exported file systems to a client. There is interest in providing a general scheme for a global, server independent name space within the context of NFS Version 4.

16.6. High performance locking

NFS Version 4 maintains lock ordering and supports mandatory blocking locks, but these features are based on a polling model. Fast lock cycling is critical to application locking performance, and this may be a weakness in our model and an area for future redesign.

16.7. SFS benchmark

The SPEC organization's SFS benchmark is a standard for measuring the performance of NFS implementations [Robinson]. An NFS Version 4 version of the benchmark remains to be done.

17. Resources for developers

The primary site for NFS Version 4 information is:

http://www.nfsv4.org

Pointers to relevant sections of the Internet Engineering Task Force site:

http://www.ietf.org

can be found there.

The CITI group at the University of Michigan is developing an open source reference implementation, and their work can be accessed at:

http://www.citi.umich.edu/projects/nfsv4/

18. Acknowledgements

The NFS Version 4 working group in the IETF contributed immensely not only to the specification, but to the discussion around the changes to the architecture and semantics of the protocol. Spencer Shepler was the editor of the NFS Version 4 protocol specification through its life in the working group. EMC, Hummingbird Communications Ltd. Network Appliance Inc., Sun Microsystem, Inc., and the University of Michigan/CITI research group participated in the first interoperability testing. Gordon Waidhofer often acted as the working group's conscience.

19. Bibliography

[Borr] Borr. A., "SecureShare: Safe UNIX/Windows File Sharing through Multiprotocol Locking," 2nd USENIX Windows NT Symposium. August 3-4, 1998. http://www.netapp.com/tech_library/3024.html

[CIFS] http://msdn.microsoft.com/workshop/networking/cifs/default.asp

[Cthon] Sun Microsystems, Inc., "Sun Enterprise Authentication Mechanism 1.0 Interoperability Notes," 1999. http://www.connectathon.org/seam1.0/

[EFF] Electronic Frontier Foundation, John Gilmore (Editor) (1998). "Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design," O'Reilly & Associates, ISBN 1565925203.

[Eisler96] Eisler, M., Schemers, R., and Srinivasan, R., "Security Mechanism Independence in ONC RPC," Proceedings of the Sixth Annual USENIX Security Symposium, 1996, pp. 51-65.

[Eisler00] Eisler, M. (2000), "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM," a work in progress to be published as an RFC by the Internet Engineering Task Force.

[Howard] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J., “Scale and Performance in a Distributed File System,” ACM Transactions on Computer Systems 6(1), February, 1988.

[Juszczak] Juszczak, C., “Improving the Performance and Correctness of an NFS Server,” Proceedings of the USENIX Winter 1989 Conference.

[Kazar] Kazar, M.L., Leverett, B., et.al, “Decorum File System Architectural Overview,” Proceedings of the USENIX Summer 1990 Conference.

[Jaspan] Jaspan, B., “GSS-API Security for ONC RPC,” 1995 Proceedings of The Internet Society Symposium on Network and Distributed System Security, pp. 144-151.

[LaMacchia] LaMacchia, B. A., and Odlyzko, A. M. (1991). "Computation of Discrete Logarithms in Prime Fields, " Designs, Codes and Cryptography," pp. 47-62. http://www.farcaster.com/papers/crypto-field/index.htm

[Macklem] Macklem, R., “Not Quite NFS, Soft Cache Consistency for NFS,” Proceedings of the USENIX Winter 1994 Conference.

[Martin] Martin, R. P., Culler, D. E., “NFS Sensitivity to High Performance Networks,” http://www.cs.rutgers.edu/~rmartin/papers/snfs.ps

[Microsoft99] Microsoft Corp., “Microsoft Distributed File System.” http://www.microsoft.com/ntserver/nts/downloads/winfeatures/NTSDistrFile/AdminGuide.asp

[Microsoft00] Microsoft Corp., "Step-by-Step Guide to Kerberos 5 (krb5 1.0) Interoperability," January 26, 2000. http://www.microsoft.com/windows2000/library/planning/security/kerbsteps.asp

[MIT] Massachusetts Institute of Technology (1998). "Kerberos: The Network Authentication Protocol." The Web site for downloading MIT's implementation of Kerberos Version 5, including implementations of RFC 1510 and RFC 1964. http://web.mit.edu/kerberos/www/index.html

[Mogul] Mogul, J. C., “Recovery in Spritely NFS,” DEC WRL Research Report 93/2, Digital Equipment Corp. Western Research Lab. http://www.research.digital.com/wrl/techreports/abstracts/93.2.html

[Pawlowski] Pawlowski, B. Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz, D., “NFS Version 3 Design and Implementation.” Proceedings of the USENIX Summer 1994 Technical Conference. http://www.netapp.com/ftp/NFSv3_Rev_3.pdf

[RFC1094] Sun Microsystems, Inc., “Network Filesystem Specification,” RFC 1094, *the NFS Version 2 protocol specification.* http://www.ietf.org/rfc/rfc1094.txt

[RFC1510] Kohl, J., and Neuman, C. (1993). "The Kerberos Network Authentication Service (V5)," RFC 1510. http://www.ietf.org/rfc/rfc1510.txt

[RFC1813] Callaghan, B., Pawlowski, B. and Staubach, P., “NFS Version 3 Protocol Specification,” RFC 1813, June 1995. http://www.ietf.org/rfc/rfc1813.txt

[RFC1831] Srinivasan, R., “RPC: Remote Procedure Call Specification Version 2,” RFC 1831, August 1995. http://www.ietf.org/rfc/rfc1831.txt

[RFC1832] Srinivasan, R., “XDR: External Data Representation Standard,” RFC 1832, August 1995. http://www.ietf.org/rfc/rfc1831.txt

[RFC1964] Linn, J. (1996). "The Kerberos Version 5 GSS-API Mechanism," RFC 1964. http://www.ietf.org/rfc/rfc1964.txt

[RFC2026] Bradner, S., “The Internet Standards Process – Revision 3,” RFC 2026, October 1996. http://www.ietf.org/rfc/rfc2026.txt

[RFC2054] Callaghan, B., “WebNFS Client Specification,” RFC 2054, October 1996. http://www.ietf.org/rfc/rfc2054.txt

[RFC2055] Callaghan, B., “WebNFS Server Specification,” RFC 2054, October 1996. http://www.ietf.org/rfc/rfc2055.txt

[RFC2203] Eisler, M., Chiu, A., Ling, L., “RPCSEC_GSS Protocol Specification,” RFC 2203, September 1997. http://www.ietf.org/rfc/rfc2203.txt

[RFC2339] The Internet Society, Sun Microsystems, Inc., “An Agreement Between the Internet Society, the IETF, and Sun Microsystems, Inc. in the matter of NFS V.4 Protocols,” RFC 2623, June 1999. http://www.ietf.org/rfc/rfc2339.txt

[RFC2623] Eisler, M., “NFS Version 2 and Version 3 Security Issues and the NFS Protocol’s Use of RPCSEC_GSS and Kerberos V5,” RFC 2623, June 1999. http://www.ietf.org/rfc/rfc2623.txt

[RFC2624] Shepler, S., “NFS Version 4 Design Considerations,” RFC 2624, June 1999. http://www.ietf.org/rfc/rfc2624.txt

[Robinson] Robinson, D., “The Advancement of NFS Benchmarking: SFS 2.0,”
Proceedings of the 13th Systems Administration Conference - LISA '99
http://www.usenix.org/events/lisa99/robinson.html

[Sandberg] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., “Design and Implementation of the Sun Network Filesystem,” Proceedings USENIX Summer 1985.

[Shepler] Shepler, S., Beame, C., Callaghan, B., Eisler, M., Noveck, D., Robinson, D., Thurlow, R., "NFS Version 4 Protocol," a work in progress to be published as an RFC by the Internet Engineering Task Force, 2000. http://www.nfsv4.org

[Taylor] Taylor, B., Goldberg, D. (1986). "Secure Networking in the Sun Environment," Proceedings of the 1986 Summer USENIX.

[Ts'o] Ts'o, T. (1997), "Microsoft 'Embraces and Extends' Kerberos V5," ;login: - USENIX News. http://www.usenix.org/publications/login/1997-11/embraces.html

[Srinivasan] Srinivasan, V., Mogul, J. C., “Spritely NFS: Implementation and Performance of Cache Consistency Protocols,” DEC WRL Research Report 89/5, Digital Equipment Corp. Western Research Lab. Also in Proc. Of the Twelfth ACM Symposium on Operating System Principals. http://www.research.digital.com/wrl/techreports/abstracts/89.5.html

[UTF8] The Unicode Consortium, http://www.unicode.org