CHAPTER 1: NFS BASICS AND PROTOCOLS

Files are the basic storage unit of most computers. Easy access to one's files is an assumed part of day-to-day operations. In the Internet age, a user's files may reside on one machine but be accessed from another machine. Remote access to files is absolutely essential. Unfortunately, new complications arise with remote access:

- How to ensure seamless access to remote files as if they were local
- How to secure file data over public networks
- How to access files when networks or servers misbehave or are down
- How fast can you access your files over a network
- And more...

Over the past two decades, several file systems have been developed to allow remote access to files. The *Server Message Block* (SMB) protocol, used by Windows machines, allows machines to share access to remote folders and printers. The *Andrew File System* (AFS), invented at Carnegie-Mellon University, is a complex, high-performance file system that makes extensive use of caching to perform well. The *Coda* file system was invented for use in environments with a highly variable quality of network connectivity. In particular, the Coda file system can handle disconnected operations—computers that disconnect from the network for an arbitrary length of time and then reconnect again. Coda synchronizes files between disconnected clients and servers after reconnection.

The *Network File System* (NFS) is the most widely used network-based file system. NFS's initial simple design and Sun Microsystems' willingness to publicize the protocol and code samples to the community contributed to making NFS the most successful remote access file system. NFS implementations are available for numerous Unix systems, several Windows-based systems, and others. While not perfectly interoperable, heterogeneous NFS systems can exchange data together homogeneously with little user notice. For the most part, users are rarely aware that their files are being served over the network; they do not have to change their behavior or any programs they used before.

In this chapter, we introduce NFS. We begin by describing the basic operating principles; then we describe the many components that make up this complex system. We follow this with the details of the NFS protocols and how they evolved. We end with an example illustrating how to configure and use NFS.

NFS Versions

There are several revisions—or protocols—of NFS. Version 3 of NFS (NFSv3) is rapidly becoming the default version on most Linux systems. This version is available in the latest Linux 2.2 and 2.4 kernels. For the purposes of this book, we are assuming that readers are using one of these kernels and therefore they are using NFSv3; we will not be discussing NFSv2 at great length. To use the latest versions of NFS with Linux, a special user-level set of utilities is needed; these can be found in the `nfs-utils` package, version 0.2.0 or newer. To find out how to retrieve and install these new versions on Linux, see Chapter 7, "Building and Installing the Linux Kernel and NFS Software."

The specification for the latest version of NFS, version 4, is finalized but relatively new. At least one prototype implementation of this protocol version exists for Linux, but it is far from stable. It will be a long while before it becomes the default version of NFS for Linux. NFSv3 became stable on Linux more than six years after its original debut, and this was for a moderately complex improvement over the NFSv2 protocol. Compared to NFSv3, NFSv4 is significantly more complex. Therefore, we do not assume that users will be using NFSv4 soon, and we only describe NFSv4 briefly in Chapter 6, "NFS Version 4."

The first implementation of NFS on Linux was in a user-level daemon called `unfsd`. Since then, NFS implementations under Linux have moved into the kernel, where they perform faster and more reliably. To distinguish these kernel-based implementations from user-based implementations, the former were prefixed with `k`—so it would be `knfsd` instead of `unfsd`, or just `nfsd`. Other parts of NFS have also been moved into the kernel, as we see later in this chapter. However, to simplify our discussions, we have assumed that NFS is implemented in the kernel, and we have dropped the `k` prefix from those services.

> 🕮 We do not discuss or use the older user-level NFS daemon (`unfsd`). We strongly recommend that administrators move away from that older, slower, and unreliable NFS server.

Overview of NFS

In this section, we overview the primary ideas that make up NFS. There are several such ideas:

- Remote procedure calls
- Retransmissions of messages

- Idempotent operations
- A stateless server
- File handles to identify files
- Caching on the client
- Maintaining Unix file system semantics

Remote Procedure Calls

Remote Procedure Calls (RPCs) are a programming paradigm that allows a local process to call a function that is implemented by a remote process as if the local process were calling its own function. RPCs allow the NFS system to be split into two parts: a component that runs on the client (or calling) host and a component that runs on the server (or called) host. An NFS client can call file system functions—implemented as RPCs—on the server as if the functions, and hence the files, were local.

Retransmissions and Idempotent Operations

In a network-based communication, software systems must handle lost packets and messages. The RPC system was designed to resend RPC messages automatically, at a given configurable interval, for up to a configurable number of retries. Since NFS is based on RPC, it supports *retransmissions* of protocol messages automatically. If the network is unreachable (because it got disconnected for a period of time or because the NFS server crashed and is rebooting), an NFS client will resend its messages several times before giving up. This ensures that NFS continues to function even when transient problems occur.

The flip side of the retransmission ability is the danger of duplicate messages. In any network, especially one that uses the *User Datagram Protocol* (UDP), duplicate packets could occur. If a duplicate RPC message is sent to an NFS server, the NFS server will try to process that message again, possibly performing the same operation repeatedly. Therefore, it is important that the NFS protocol and its operations all be *idempotent*—executing them again does not change the outcome. The decision to make sure all NFS operations are idempotent figured heavily into the protocol design and into the key design feature of the NFS server, its statelessness, which is discussed next.

A Stateless Server

One important aspect of a distributed file system design is its handling of network failures and how well it recovers from host crashes. This point is crucial for the consistency and reliability of data. In local file systems, such as EXT2, a user has a strong guarantee that the data just saved in a `write` system call is actually saved to disk. Should the machine crash, the data will remain on disk.

Consider the case of a client-server statefull distributed file system. A client writes some data using the `write` system call. The data goes over the wire from the client to the server. To improve performance, a statefull server might cache that data in memory and return a successful return code to the client. The server can schedule to save the data onto physical media at a later time. Since the client received a successful return code, it believes that the data must have been saved to stable media—a usual assumption when the `write` system call returns successfully. However, if the server crashes after signaling the client that `write` succeeded but before the server had a chance to synchronize the data to stable media, that data would be lost. When that happens, the client's assumption of a successful write becomes incorrect.

There are several ways to improve the reliability of statefull servers while maintaining high performance. However, a much easier way to ensure reliable crash recovery is to make the server *stateless*; such a server keeps no state, therefore avoiding any loss of data during crashes. With a stateless server, the client can be assured that when a remote NFS operation succeeds, the remote file system is guaranteed to be consistent, even if the remote server crashes.

The assumption that NFS servers are stateless simplifies the NFS server code and the NFS protocol significantly. To ensure data consistency, the NFS client host just waits for a previously available remote server that has crashed to reboot. This wait state often locks all the processes that were performing operations on the remote server, which results in the infamous but necessary error message "NFS server not responding—still trying." Once the server comes back up, the NFS client that was waiting for the server to come back up can resume its operation.

One of the unfortunate results of the statelessness of the NFS server is poor performance. The server must write to disk all data it receives from clients before it returns successfully to the client. This synchronous writing operation is one of the largest drawbacks of NFS, and it impacts its performance significantly. Below we discuss several solutions that were invented to improve performance through client-side caching.

Note that the NFS client is not stateless and it does keep cached data in order to improve performance. If the client crashes, some data may be lost, but that is no different from a client host crashing in the middle of a write operation to a local disk. This is very different from when a server crashes; when this happens, many clients' data could be lost all at once, and those clients are now left in an inconsistent state of making wrong assumptions about the validity of their data.

File Handles

NFS clients and servers handle many files at once. Often, a client may issue many system calls on a single file, each call turning into one or more NFS operations over the wire. The client and server must agree about which files both should apply a series of operations to. For that reason, NFS servers issue *file handles* to clients. File handles are unique identifiers that the server generates for each file that a client uses.

These identifiers are anywhere from 32 to 64 bytes of data in length. File handles are opaque to the client—it does not know what the individual bytes in each handle are for. Only the server understands file handles. NFS servers generally encode enough information on the local disks to allow the servers to find out the exact file name that a remote client wishes to access. Servers usually encode the following pieces of information in each file handle:

- A file system identifier, an index number of a mounted local file system
- The inode number of the file within the file system
- An inode generation number, described below, under “Maintaining Unix Semantics”
- Other information as they see fit, usually listed in the C header file `/usr/include/linux/nfsd/nfsfh.h`

Recall that the server is stateless; therefore, it cannot keep an association (state) that maps file handles to actual files on the server’s disks. The server generates its own file handles and encodes in them everything that it needs to find out the exact file that any given file handle was generated for. Clients receive these file handles from the NFS server, and they must not modify them. Instead, the clients must send back the same file handles the server provided when these same clients wish to access the same files. When NFS servers get the returned file handles, even (and especially) after a server crash, they can decode the file handle and tell exactly which file the client wishes to apply the particular operation to. This is possible because the file handle encodes information that uniquely identifies the exact file within a given file system that resides on the server.

If an NFS client passes a file handle to an NFS server that the server is unable to decode, the server will refuse to use that file handle; the NFS server will return an error code back to the client telling it that the file handle is *stale*. Stale file handles happen most often when a file system on a remote file server is moved, the file system is reformatted or restored, or the mount point for the file system changed. See Chapter 5, “NFS Diagnostics and Debugging,” for how to handle stale file handles.

Client-Side Caching

As we just mentioned above, servers are stateless but clients are not. To improve performance, clients cache file data and file attributes.

NFS clients and servers employ a special I/O daemon (`rpciod`) in order to cluster multiple write operations from the client. When several consecutive write buffers accumulate, `rpciod` will issue a single large write request to the NFS server. This can improve performance manyfold, especially when a client process performs many small writes or appends to the same file. Note that client-side caching and buffering via `rpciod` does not violate the statelessness of the server, but it does increase the chance of greater data loss should the client crash.

NFS clients also cache attributes. This is intended to speed up operations that look up file attributes (such as the `lstat` system call) and operations that change file attributes (such as `chmod` or `chgrp`). This behavior also has annoying side effects since changed file modes do not propagate immediately from one client to a server, and certainly, they do not propagate immediately from a server to another client.

This behavior also has security implications. Consider two clients, A and B, that read the same file, possibly at the same time. Though client A might change the mode of the file so it is more restrictive and should no longer be readable by client B, client B might still cache the previous more permissive modes. This allows client B to continue to access a file that it is no longer supposed to.

Maintaining Unix Semantics

Several complications arise from Unix's particular file system semantics. These semantics must be maintained accurately so that user processes could mix access to local and remote files without any noticeable difference.

Inode Generation Numbers

The first such semantic issue has to do with inode numbers, the file index numbers that help locate the data of the file on disk. When a file is removed, its inode is also removed. That inode number, however, can be reused at a later date to create a new file. Inode numbers can be recycled shortly after becoming available, or much later, depending on the system usage and the operating system.

NFS servers encode the inode number of a file in the file handle so that they can identify the exact file that a client wants to access. Consider what happens if that file was removed, possibly on the server itself or by another client, and then another file was created that had the same inode number. If the NFS client host retained its file handle and tried to use it, the NFS server would find the new file with the same (old) inode number and would incorrectly think that it was the file that the client wanted to use. This scenario can result in serious data corruption.

For that reason, NFS servers add another piece of information into the mix that makes up the NFS file handle: an *inode generation number*. This number is usually an integer that gets incremented each time the same inode number is reused, or it is a time stamp of the file's creation time. Either way, it is an additional piece of information that helps the stateless server correctly identify the exact file that is being used.

Hidden *.nfs* Files

Another quirky piece of Unix file system semantics is the ability to open a file, unlink (delete) it, and still be able to access the file's data as long as the file remains open. This particular behavior was originally intended to allow programs to access temporary unnamed files in a way that guaranteed that the physical storage for those files was reclaimed as soon as the process using them closed the files' descriptors. Unfortunately, this behavior was used more often by attackers of Unix systems determined to hide their intentions by using unnamed files.

Since the NFS server is stateless, it is not allowed to keep any state on the current status of files. In particular, the server cannot tell when a file is opened or closed (these two operations are not even part of the v2 and v3 NFS protocols). Therefore, the NFS client-side code has to handle this case specially. The client side can do so because it does know when a file has been unlinked while being opened. When the client sees this unlink operation, it issues an NFS call to *rename* the file to a hidden file whose name starts with `.nfs` and ends with a sequence of characters and digits that guarantee the uniqueness of that special dot file.

The NFS client code remembers the renamed file name. When the client sees the close operation, it then issues an NFS call to actually delete the `.nfs` file, therefore removing its physical storage.

This behavior preserves this odd Unix semantics as closely as possible. Occasionally, clients can crash and leave behind `.nfs` files that the client no longer knows about. These stale files must be cleaned. For that reason, most systems add a daily cron job to look for and delete old `.nfs` files.

File and Record Locking

Modern Unix systems provide several system calls that allow a process to lock a whole file or a smaller region of a file while accessing it. This is most often used to guarantee that only one process at a time can be modifying a shared file. Unfortunately, proper support for locking requires that the operating system maintain a state about the locked files. Since the actual files reside on the server, locking information (state) must be maintained on the server. This state must be maintained on the server to guarantee that clients from different hosts do not all try to write to the same file at the same time.

So, it appears that NFS server hosts have to keep some state, but the NFS server itself (`nfsd`) was designed to be stateless. To reconcile these two opposing needs, the original designers of NFS opted to create a separate locking daemon (`lockd`) that NFS clients must communicate with to guarantee consistent locking behavior.

Components of the Network File System

The NFS system is complex and contains many components that interact with each other over special protocols. The different components use various configuration and state files. Figure 1.1 shows the main components of NFS and the primary configuration files. On the upper part of the figure we see an NFS server, and on the lower part an NFS client host. Each host has its own kernel-level services: *eXternal Data Representation* (XDR), *Remote Procedure Call* (RPC), NFS client or server, I/O daemon, and locking daemon. Each host also has its own user-level services. Both kernel-level and user-level services depend on the function that the host has: an NFS client or an NFS server. In addition, there are special configuration files (usually in `/etc`) that are used on each host based on its function. Note that if a host is both an NFS server and an NFS client, it will have to run the services on both parts of Figure 1.1.

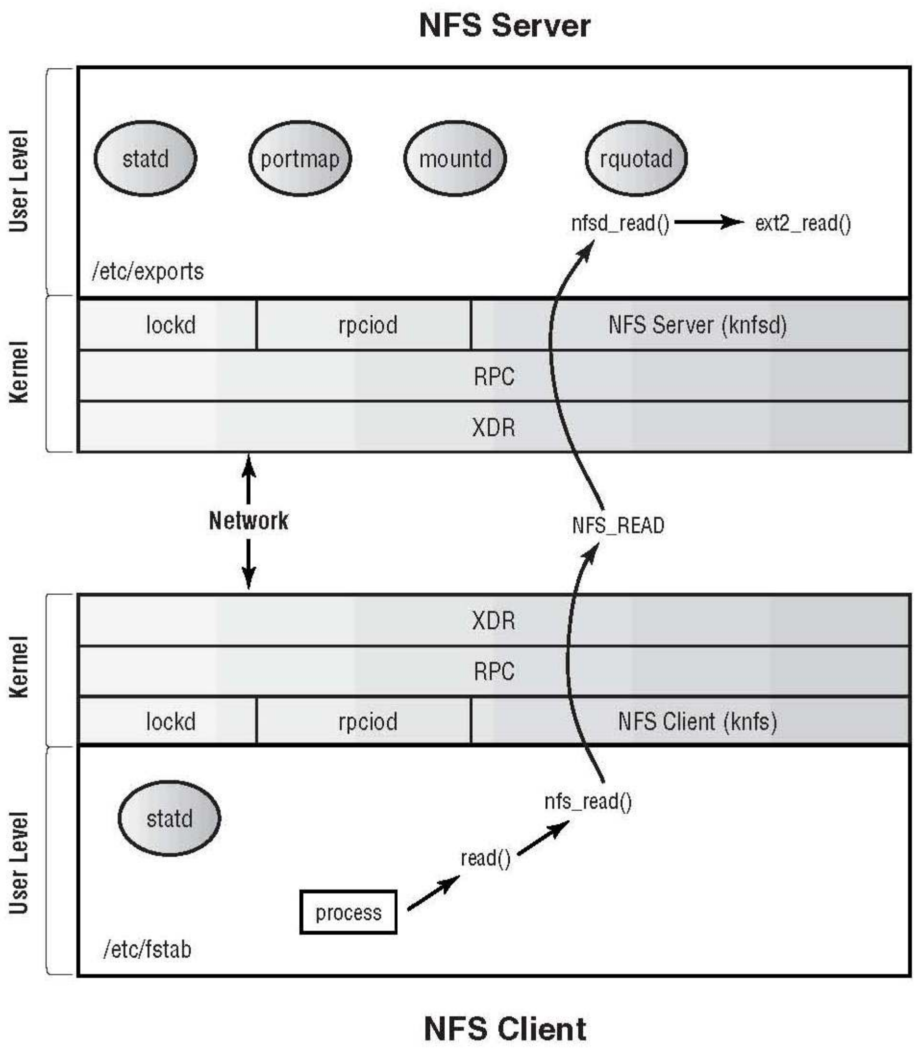

Figure 1.1: The components of an NFS system [F0101.eps]

Figure 1.1 also shows the flow of data for a simple read operation, when an NFS client wants to read a file from an NFS server. We describe that data flow at the end of this chapter.

In the next sections, we begin by describing each component of an NFS system. We discuss them in dependency order, describing a component first before other components that use it. Once we have described each component's function, we can put them all in context and show you how the overall NFS system works.

XDR: eXternal Data Representation

The basic need of a distributed file system such as NFS is to exchange data across a network between heterogeneous machines. Most CPUs use one of two ways to represent data in memory:

Big-endian

A *big-endian* machine stores the most-significant byte of a machine word on the left side of a register. For example, the integer 2 as represented in hexadecimal on a big-endian machine would be 0x00000002.

Little-endian

A *little-endian* machine stores the least-significant byte of a machine word on the left side of a register. On such a machine, the integer 2 (in hexadecimal) would be represented as 0x02000000.

Since there are two such representations for data, it is imperative that when a big-endian machine exchanges data with a little-endian machine, they both agree on the representation of the data being exchanged. If they did not agree on a common format, these machines could corrupt data. For this reason, the Internet standard for exchanging data over the network is in *network order*, which is big-endian. This means that little-endian machines, such as Intel-based ones, must convert data to big-endian format before sending it over the wire, and they must convert it back to little-endian (which is their own *host order*) when they read data from the network. This process must be done very carefully and consistently.

The *eXternal Data Representation* (XDR) system was designed to simplify data exchange among networked hosts. XDR can encode any arbitrary data in a uniform manner and prepare it to be sent over the network. XDR can also take an arbitrary data stream off the network and decode it into its original data units.

XDR comes with libraries and functions that can encode and decode many primitive types: integers, longs, floating-point numbers, strings, bytes, and more. Programmers using XDR can build more and more complex encoding functions using basic ones. For example, they can create an XDR encoding function for a complex data structure that contains simpler native types, and then they can create an even more complex XDR function that builds on top of that.

XDR offers a powerful method to encode arbitrarily complex data structures in a uniform network order and ensures that when the data arrives on the other end, it can be decoded into the exact data structure that the sender encoded. The ability to transfer data structures across a network is the first necessary component of sharing files over a network.

RPC: Remote Procedure Call

XDRs provide a mechanism to exchange data uniformly across a network. *Remote Procedure Calls* (RPCs) provide a method for calling functions on another host. Most programming languages allow you to call functions in your own program or in libraries that the program links with; these languages also allow you to pass data to the function and to return data back from the function. Together, XDR and RPC allow you to do the same—only on remote hosts.

RPCs allow programmers to provide shared services in a single location designated as the server. Many clients can access those services as easily as they would if they were calling a function in their local C library. With RPCs, code and services can be shared more easily between hosts.

The RPC system also allows for implementation versioning. A server can provide multiple implementations of a given service, and clients making RPC calls can choose the implementation they want to use. This allows network-wide services to evolve naturally over time (even in an incompatible manner), while maintaining backward compatibility; therefore, newer RPC clients can use the latest versions of the RPC service, while older ones can use older (yet compatible) versions of the RPC service. This versioning ability will prove to be very useful to the evolution of the NFS protocol.

Refer back to Figure 1.1. Here we see that in order for two machines to exchange information, they must go through the RPC layer, which in turn uses the XDR layer to encode the data. When an RPC client calls an RPC server, the client has to provide the server with at least this basic information:

- The name of the remote host
- The program number of the remote service, uniquely identifying the RPC service
- The version of the service (a number)
- The number of the exact remote procedure to execute on the remote service
- The arguments and data to pass to the remote service
- Placeholders for the arguments and data to get back from the remote service

The Portmapper

Now that we have the ability to create many RPC services, all versioned, and each with their own set of procedures, we have to decide how they should be advertised to clients. Traditionally in Unix, each service gets assigned a port number on which to listen: Telnet on port 23, FTP on port 21, SSH on port 22, SMTP on port 25, and so on. This agreement on the port number is important; without it, clients would not know how to contact the service. Unfortunately, a limited number of port numbers is available (generally only 65535), and even fewer are available to privileged (root) services—the first 1024 ports. Therefore, if we are to support many possible RPC services, we cannot dedicate a port number to each such service; if we did, we would quickly run out of available ports.

The *portmapper* solves this problem. Each RPC service has its own program number, a large number that is either assigned to the service or can be designated by the programmer of the service. The portmapper maps the RPC program number to the actual port the service runs on.

The portmapper, however, can do more than that. When an RPC program wishes to make its services available to clients to contact, it registers its service with the portmapper. It provides the portmapper with the following information:

- The program number that it uses for the service
- The version of the service that it supports
- The transports that it supports: UDP, TCP, or both

RPC servers can register multiple services, versions, and transports to use by registering each tuple (<*service*, *version*, *transport*>) individually with the portmapper. The portmapper records this information so that when an RPC client contacts it to call an RPC service, the portmapper can tell it what port the service listens on. Then the RPC client can contact that service directly. The only requirement for RPC clients to contact the portmapper is that the portmapper itself must have a preassigned port number: 111.

Getting RPC Information

The portmapper may be the mother of all RPC services, but it is also an RPC service itself. As such, it has an implementation version and transports that it supports. To find out this information about any RPC service, including the portmapper, use the `rpcinfo` program. Listing 1.1 shows the five forms of typical use for the `rpcinfo` command.

Listing 1.1: Usage of the `rpcinfo` program

```
Usage: rpcinfo [ -n portnum ] -u host prognum [ versnum ]
       rpcinfo [ -n portnum ] -t host prognum [ versnum ]
       rpcinfo -p [ host ]
       rpcinfo -b prognum versnum
       rpcinfo -d prognum versnum
```

The first two forms contact a remote *host* and query the NULL (first) procedure of *prognum*. This form is intended to test if a remote service is up and responding using UDP (-u); the second form tests using TCP (-t). You can optionally specify the version number (*versnum*) of the service you wish to verify.

Normally, rpcinfo will contact the remote portmapper to find out which port number the service listens on. If you know that port number, you can specify it using the -n option. For example, to test if a remote NFS server (RPC program number 100003) named aladdin supports protocol version 3 using UDP, run the command as seen in Listing 1.2.

Listing 1.2: Testing the availability of an RPC service

```
[ezk]$ rpcinfo -u aladdin 100003 3
program 100003 version 3 ready and waiting
```

The second form of usage for rpcinfo is the most often used one—listing the RPC services that run on a given host. To list the RPC services on a host, run rpcinfo -p as seen in Listing 1.3.

Listing 1.3: Listing the RPC services running on a host

```
[ezk]$ rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100004    2   udp    981  ypserv
    100004    1   udp    981  ypserv
    100004    2   tcp    984  ypserv
    100004    1   tcp    984  ypserv
    100007    2   udp    995  ypbind
    100007    2   tcp    997  ypbind
    100024    1   udp    964  status
    100024    1   tcp    966  status
    100011    1   udp    896  rquotad
    100011    2   udp    896  rquotad
    100005    1   udp   1037  mountd
    100005    1   tcp   1024  mountd
    100005    2   udp   1037  mountd
    100005    2   tcp   1024  mountd
    100005    3   udp   1037  mountd
    100005    3   tcp   1024  mountd
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100021    1   udp   1038  nlockmgr
    100021    3   udp   1038  nlockmgr
    100021    4   udp   1038  nlockmgr
    100021    1   tcp   1025  nlockmgr
    100021    3   tcp   1025  nlockmgr
    100021    4   tcp   1025  nlockmgr
    100001    3   udp    941  rstatd
    100001    2   udp    941  rstatd
    100001    1   udp    941  rstatd
    100008    1   udp   1040  walld
    300019    1   tcp    975  amd
    300019    1   udp    976  amd
```

Listing 1.3 shows the RPC services on this host, many of which are related to NFS and will be described in the sections that follow.

The third form of usage for `rpcinfo` sends an RPC broadcast command for a given service and version of that service. It returns the IP address and name of every host that responds. For example, to find out which hosts support the `nlockmgr` service version 4, run the command as seen in Listing 1.4.

Listing 1.4: Querying for all hosts that support an RPC service

```
[ezk]$ rpcinfo -b 100021 4
172.29.1.65 lorien.dev.example.com
172.29.1.66 jigglypuff.dev.example.com
172.29.1.67 kendo.dev.example.com
172.29.1.57 toreador.dev.example.com
172.29.1.3 aladdin.dev.example.com
172.29.1.126 corvair.dev.example.com
172.29.1.87 atlast.dev.example.com
172.29.1.61 rewind.dev.example.com
```

The fourth and last form of usage for `rpcinfo` removes an association of program number (RPC service) and version number from the portmapper. Only the superuser can execute this command because once the association is removed, that service can no longer be used!

📖 Many RPC programs use a different service name from the actual program name. Most often, a program such as `/usr/sbin/rpc.mountd` is represented by the RPC portmapper as `mountd`. Throughout this chapter, we will refer to the service name of the program using the short form, and to the actual program name using the longer form that begins with the `rpc.` prefix.

The Portmapper RPC Protocol

For a given version, each RPC service can support any number of procedures. These procedures are individual functions that RPC clients can call. Being an RPC service itself, the portmapper is no different. The following procedures are defined for the (latest) portmapper protocol, version number 4:

PMAPPROC_NULL

Every RPC program has this NULL procedure (also called *procedure 0)* because its number is zero, which is sometimes known as the *ping* procedure. This procedure does not take or return arguments. Its purpose is for RPC clients to call and find out if a service exists and is running for a given version of a protocol.

PMAPPROC_SET

The SET procedure registers an RPC program number, version number, and protocol transport type with the portmapper.

PMAPPROC_UNSET

This procedure deregisters an RPC service from the portmapper.

PMAPPROC_GETPORT

The GETPORT procedure returns the port number for a given RPC program number and version.

PMAPPROC_DUMP

The DUMP procedure returns the list of all services registered with the portmapper. It is used with `rpcinfo -p`.

PMAPPROC_CALLIT

This procedure is used to make the portmapper call a procedure of *another* RPC service on the same host. Normally, you would not need to use this procedure; you would call the RPC service directly. On Linux, the CALLIT procedure is limited to UDP only, it performs no authentication, and it does not return any error information. It is therefore of limited use, and not very secure.

The Mount Daemon

XDR, RPC, and the portmapper are generic parts of the RPC system on which NFS is based. The *mount daemon* (`mountd`) is the first RPC service we cover that is specific to NFS. This server implements the MOUNT protocol. It retrieves a list of *exported* directories from a configuration file called `/etc/exports`. This configuration file describes which directories this NFS server allows remote NFS clients to access. NFS clients contact `mountd` to request initial access to an NFS volume. The `mountd` daemon checks the list of currently exported volumes against the credentials of the NFS client and responds, either allowing or denying access.

If the daemon grants access, it returns a *root file handle* to the NFS client. This file handle is the NFS system identifier for the top-level directory of the exported volume. This piece of information is important to the NFS client. Using the root file handle, an NFS client can, for example, request listing of the entries in that directory, read or write files within that directory, or get handles for additional files or subdirectories. In other words, without the root file handle, an NFS client cannot begin to access any files that are in that volume.

The *rpc.mountd* Program

The `rpc.mountd` program is the user-level RPC daemon that processes MOUNT requests from clients. Administrators list volumes to export in the file `/etc/exports`. A special tool, `exportfs`, maintains a current list of exported volumes in the file `/var/lib/nfs/xtab`. The `rpc.mountd` server reads the current export list from that file. We show an example using these files later in this chapter, and we detail their format and options in Chapter 2, “Configuring NFS.”

The `rpc.mountd` daemon supports the command-line options seen in Listing 1.5.

Listing 1.5: Usage of the `rpc.mountd` program

```
Usage: rpc.mountd [-Fhnv] [-d kind] [-f exports-file] [-V version]
           [-N version] [--debug kind] [-p|--port port] [--help]
           [--version] [--exports-file=file]
           [--nfs-version version] [--no-nfs-version version]
```

`-d` or `--debug`

Turn on debugging. All debugging messages are logged via `syslog` to the LOG_DAEMON service.

`-F` or `--foreground`

Do not daemonize `rpc.mountd`. Instead, the server remains running in the foreground. This is useful if you wish to debug it using a debugger such as `gdb`.

`-f ARG` or `--exports-file ARG`

By default, `rpc.mountd` reads export information from `/etc/exports` when the daemon starts up. With this option, it reads the export information from a file specified by *ARG*. Note that `rpc.mountd` also reads export information from `/var/lib/nfs/xtab`; we describe these files in greater detail in Chapter 2.

`-h` or `--help`

Prints the help message shown in Listing 1.5.

`-N` *`ARG`* or `--no-nfs-version` *`ARG`*

The daemon is capable of returning NFS root file handles for multiple versions of NFS. If, however, you do not wish the daemon to support a certain version of NFS, you can specify it with this option, as *ARG*. This is most often useful when you do not wish to use a buggy or unstable implementation of the NFSv3 protocol:

```
[root]# rpc.mountd --no-nfs-version 3
```

`-V` *`ARG`* or `--nfs-version` *`ARG`*

This option is the opposite of the `-N` option: it forces `rpc.mountd` to offer the version as specified in *ARG*.

`-p` *`ARG`* or `--port` *`ARG`*

Usually, `rpc.mount` will bind to a port that is randomly assigned to it by the portmapper. If you wish to force `rpc.mountd` to use another port, specify it in *ARG*.

`-v` or `--version`

Print the version of the daemon and exit.

The MOUNT Protocol

The latest version (3) of the MOUNT protocol implemented by `rpc.mountd` supports the following RPC procedures:

MOUNTPROC_NULL

This is the ping procedure for testing the `rpc.mountd`'s responsiveness.

MOUNTPROC_MNT

An NFS client passes the name of a directory it wishes to mount. If the client is authorized to access that directory, this procedure returns the root NFS file handle for that directory to the NFS client. Also, `rpc.mountd` updates the remote mount table file `/var/lib/nfs/rmtab`.

MOUNTPROC_DUMP

Return the full list of remotely mounted file systems, as listed in the `/var/lib/nfs/rmtab` file. This list includes the names of remote hosts that mount this NFS server's file systems, and the names of the directories that they mount.

MOUNTPROC_UMNT

Delete an entry from `/var/lib/nfs/rmtab`, which is a pair of hostname and directory name that the remote host mounts.

MOUNTPROC_UMNTALL

Delete all entries for `/var/lib/nfs/rmtab` for the NFS client host that calls this procedure.

MOUNTPROC_EXPORT and MOUNTPROC_EXPORTALL

Return the full list of all exported file systems and the names of the machines that are allowed to mount these file systems.

MOUNTPROC_PATHCONF

Return the POSIX *pathconf* information to the client. This information includes file system parameters about the server, such as the maximum allowed length for a path name, the maximum length for a file name, etc.

The NFS Locking Daemon

Unix supports locking files or portions of files (such as record locks) to ensure that no two people can attempt to write the same part of the file. This guarantees data consistency. On a single host, the kernel maintains information about user processes that acquired file locks and ensures the appropriate handling of lock requests and file access while files are locked. The key part in ensuring consistent locking is that there is a central authority—the kernel—that must arbitrate lock and write requests.

In a distributed file system such as NFS, many clients may wish to lock the same remote file. The most logical choice for the central lock arbitration authority is the NFS server itself because only it has access to the actual locked file. To maintain this locking information, however, the NFS server would have to track which client locked which file or a portion of a file. However, the NFS server was designed to be stateless, and thus it cannot maintain any state. For this situation, the solution Sun came up with was to add another RPC protocol to handle locking operations.

The RPC *Locking Daemon*, `rpc.lockd`, implements the *NFS Lock Manager* (NLM) protocol. The NLM protocol was designed to support multiple clients wishing to lock files consistently through NFS. NFS clients make RPC calls to the local `rpc.lockd`, which communicates with the remote `rpc.lockd` to ensure consistent locking of files. The `rpc.lockd` server uses another daemon, `rpc.statd`, which implements a status monitoring service; we describe `rpc.statd` below.

> 🕮 At this stage, astute readers might begin to wonder why there are so many components of the NFS system; for instance, why is this locking protocol not integrated together with the NFS server `nfsd`? If you are asking this question, you would be right, but it took more than 15 years to do just that: integrate all of these extra protocols into the NFS protocol. See Chapter 6, "NFS Version 4." You will not be disappointed.

The *rpc.lockd* Program

In older Linux systems, `rpc.lockd` was not implemented at all. In later kernels, such as those distributed with Red Hat 7, a kernel-level thread implementation of this service is included; it is sometimes called `klockd`. In Linux 2.2.18 and 2.4 kernels, the `klockd` thread is spawned from the `knfsd` thread on demand.

The `rpc.lockd` program takes no arguments or switches. It starts the NLM RPC service, if the kernel had not already started it. There are several ways to find out if your kernel supports the NLM service in the kernel. First, you can run `modprobe lockd` and then check to see if a `lockd` module is listed in the output of `lsmod`. Second, you can run `ps axf` and look for an output such as seen in Listing 1.6.

Listing 1.6: Process listing output for in-kernel NFS services

```
737 ?        SW    22:22 [nfsd]
745 ?        SW     0:00  \_ [lockd]
746 ?        SW     0:09      \_ [rpciod]
```

Whenever you see processes listed in square brackets, it usually (but not always) indicates that these are kernel threads (*kthreads*). *Kernel threads* are independent programs that run in the kernel: they are fast since they are in the kernel, and they are independent so they can execute actions on their own, separately from the main kernel itself. Listing 1.6 shows three such kthreads. The main one is `nfsd`, which automatically spawns `lockd` as needed, which automatically spawns `rpciod` (described below).

The NFS Lock Manager Protocol

The latest version (4) of the NLM protocol implemented by `rpc.lockd` supports the following RPC procedures:

NLMPROC_NULL

This is the ping procedure for testing the `rpc.lockd`'s responsiveness.

NLMPROC_TEST

This procedure tests to see if a lock is available to the NFS client that wants to lock a file.

NLMPROC_LOCK

This procedure creates a lock for a range of bytes within a file (which could subsume the entire file if desired). If the server is unable to provide the lock because another client may hold it, then the client can wait for the lock. The client can then CANCEL the request or wait until it gets the GRANTED response; both of these choices are described next.

NLMPROC_CANCEL

When a client asks to lock a file and the file is locked by another client, the requesting client must block waiting for that lock to be released. Sometimes, the waiting client may opt to cancel that request and do something else instead. This procedure cancels a pending request for a lock.

NLMPROC_UNLOCK

This procedure removes a lock that was previously given for a range of bytes within a file.

NLMPROC_GRANTED

This procedure is initiated by the server and sent back to the client (via a callback) to tell the client that a request for a lock has been granted.

NLMPROC_SHARE

The remaining four procedures are used primarily by MS-DOS clients utilizing PC-NFSD. This procedure creates a share reservation for a file on an MS-DOS system, essentially locking an entire file while it is in use by MS-DOS.

NLMPROC_UNSHARE

This procedure releases a share reservation for a file on an MS-DOS system.

NLMPROC_NM_LOCK

This procedure establishes a non-monitored lock on a file. It is used primarily on single-threaded systems, such as MS-DOS, that cannot utilize the Network Status Monitor protocol. Clients using this procedure are responsible for clearing out server locks.

NLMPROC_FREE

If a client that used a non-monitored lock crashes, it may leave locks on the server that must be cleaned up. This procedure instructs a server to remove all locks for the calling client.

The NFS Status Daemon

As we described above, the `rpc.lockd` daemon coordinates locks between NFS clients and an NFS server. We explained how `rpc.lockd` maintains state about who holds locks, on which files, and for what byte ranges in the file. The problem is what happens when the host that maintains lock state, the NFS server running `rpc.lockd`, reboots and loses that state information. If it lost information about locks, NFS clients that acquired those locks could no longer use them. The solution Sun came up with was to add yet another RPC protocol to handle the recovery of locking information.

The *Network Status Monitor* (NSM) protocol was originally designed as a general purpose active state sharing service for RPC systems. In practice, however, only `rpc.lockd` makes use of this service—for maintaining state about NFS locks. Furthermore, over time, the protocol had actually devolved to include passive state sharing messages.

To ensure that this state information is not lost upon server reboot, `rpc.statd` saves its state information on disk, in files under the directory `/var/lib/nfs/sm`. Each file in that directory is named after the client that holds any locks.

If an NFS server crashes and reboots, when it comes back up, `rpc.lockd` asks `rpc.statd` for any known locks. If there were any locks recorded in `/var/lib/nfs/sm`, then the NFS server's `rpc.statd` and `rpc.lockd` daemons provide that information to their counterparts on each NFS client that held any locks. This is done before the NFS server is fully ready to serve file access requests in order to ensure that all NFS clients are back in sync with any NFS servers that granted them locks.

The *rpc.statd* Program

The `rpc.statd` program supports only one startup option. By default, this program daemonizes (backgrounds) itself. If you specify the `-F` option, `rpc.statd` will remain running in the background. This is most useful when you are debugging the daemon using tools such as `gdb`.

The Network Status Monitor Protocol

The latest version (1) of the NSM protocol implemented by `rpc.statd` supports the following RPC procedures:

SM_NULL

This is the ping procedure for testing the `rpc.statd`'s responsiveness.

SM_STAT

This procedure tests if a given host is being monitored. This procedure is part of the original active monitoring design and may not be fully implemented in `rpc.statd`.

SM_MON

This procedure tells `rpc.statd` to begin monitoring a given host. The procedure is used by `rpc.lockd` before granting the very first lock to the host. That way, if the NFS server host crashes, `rpc.statd` will be able to inform the client about these locks when the server comes back up.

A process using this procedure must supply a callback routine to invoke when the status of the monitored host has changed. This way a client that has asked to monitor a host can be informed by a remote RPC client when any change in status had taken place. This is used with the SM_NOTIFY procedure described below.

SM_UNMON

This procedure tells `rpc.statd` to stop monitoring a given host. It is usually used by `rpc.lockd` after releasing the last lock of that client host.

SM_UNMON_ALL

This procedure tells `rpc.statd` to stop monitoring all hosts. It is usually not used by `rpc.lockd`.

SM_SIMU_CRASH

If `rpc.lockd` crashes on the client, all lock information on the client is lost. When `rpc.lockd` is restarted, it sends a message to the local client's `rpc.statd` to inform the status daemon that the host had crashed and is now back up. This message simulates a crash from the client's point of view: `rpc.statd` informs all NFS servers that the lock state had been lost by sending them the NLM_UNLOCK RPC message. This procedure is an artifact of poor past designs and implementations of this service: this message had to be sent when some services of the NFS system failed.

This procedure is not too practical on recent Linux systems since their `rpc.lockd` service is implemented in the kernel, and it is thus not that likely to crash as often as user-level daemons might. If the kernel module for `rpc.lockd` crashes, a more severe situation requiring a complete host reboot may occur—no need to simulate that.

SM_NOTIFY

When `rpc.statd` crashes on a host and then comes back up, it must inspect the state of monitored hosts it recorded in `/var/lib/nfs/sm`. For each host listed there that has asked for status monitoring, this client sends this NOTIFY message to the remote host. When a remote `rpc.statd` receives such a notification, it invokes the callback procedure registered with the original SM_MON request prior to this client's crash. This series of notifications and callbacks is intended to restore the state of all status monitoring to its original condition before the crash.

The NFS Remote Quota Daemon

The `rpc.rquotad` daemon implements the RQUOTA protocol. It is currently used by only one program: `quota`. This program displays quota information about one or more users. With NFS in place, the `quota` program can also contact remote `rpc.rquotad` servers to retrieve quota information for users of remote file systems.

📖 The `rpc.rquotad` daemon does not enforce quotas nor is it used to provide quota information for the parts of NFS that do enforce quotas:

the NFS server itself. On Linux, the NFS server always enforces quotas whether `rpc.rquotad` runs or not.

The `rpc.rquotad` program itself is very simple. It takes no arguments and is usually started at boot time from the `/etc/rc.d/init.d/nfs` startup script.

Similarly, the RQUOTAD protocol is very simple. The latest version (2) of the RQUOTAD protocol implemented by `rpc.rquotad` supports the following two RPC procedures:

RQUOTAPROC_GETQUOTA

This procedure returns the list of all available quotas from the remote server, including those that are not activated (in use at the time).

RQUOTAPROC_GETACTIVEQUOTA

This procedure returns only the list of active in-use quotas.

The NFS I/O Daemon

Since the NFS server is stateless, it may not keep any information that could be lost if the server crashed. This means that when an NFS client writes data to the server, the server must write it to stable storage immediately. Unfortunately, that synchronous write is slow. Furthermore, the server may have to update additional metadata, which causes further synchronous writes of disk blocks. This problem seriously affected performance of early NFS servers. The solution to this problem was to add yet another statefull server to the NFS system that improved the performance of writes over NFS.

Traditionally, the `rpciod` server runs on the NFS client. It interfaces with the rest of the kernel and collects write requests to remote NFS servers. Instead of sending each write request to the remote NFS server, the `rpciod` server collects them and sends writes in larger, but less frequent batches. In particular, `rpciod` looks for consecutive writes to a file that can be combined into a larger sequential write. When `rpciod` has gathered enough writes, it sends them as one large NFS write request. This saves lots of network bandwidth and extra work for the NFS server.

With `rpciod`, local NFS users can perform writes and those writes will return immediately with a successful return code. However, note that the data has now been buffered on the local host and will be lost if the local host crashes. The assumption with the design of `rpciod` was that if the local host crashed, much more data could be lost anyway, and that the loss of buffered data was a reasonable compromise that yielded in greatly improved performance. Surely it is better than an NFS server crashing and losing lots more data that many clients believe to have been written to stable storage.

In the latest versions of Linux, the `rpciod` service is implemented as a kernel thread and is invoked automatically by the in-kernel NFS server or client as needed.

The NFS Client Side

Now that we have described all other components of NFS, we begin discussing the actual NFS components. We start with the NFS client-side code. In the next section, we cover the NFS server-side code and the NFS protocols.

The NFS *client-side* code had always resided in the kernel in Linux. More recent versions of Linux also support newer NFS protocols as well as TCP transports. Your Linux system must support client-side NFS to be able to mount remote NFS servers. To find out if your system supports the client-side NFS, see if `nfs` is listed in `/proc/filesystems`. If it is not, run `modprobe nfs` to check to see if NFS support is available as a loadable kernel module, and then check `/proc/filesystems` again. If you cannot find `nfs` listed in `/proc/filesystems`, your client host does not support NFS. Go to Chapter 7, “Building and Installing the Linux Kernel and NFS Software,” for details on how to add the right NFS support to your Linux system.

The main function of the NFS component on the client side is to translate system call requests into their NFS protocol RPC messages and send these messages over to the remote NFS server. The NFS client side also coordinates with the local I/O daemon (`rpciod`), the locking daemon (`rpc.lockd`), and the status-monitoring daemon (`rpc.statd`).

The NFS Server

The *NFS server*, `nfsd`, is at the heart of the NFS system. It listens for RPC requests from remote hosts and interprets them according to the NFS protocol. It sends responses back to clients using RPCs. It also communicates with other components that run on the NFS server host: the locking daemon `rpc.lockd`, the status daemon `rpc.statd`, and the I/O daemon `rpciod`.

📖 The term *NFS server* can have multiple meanings. Some consider it to mean the whole host that serves files over the NFS protocol. Some will use the term to refer to the whole NFS system: client hosts and server hosts sharing files. Others will assume that the NFS server is only the actual component or program that implements the NFS protocol: `rpc.nfsd` if in user-level, and `nfsd` or `knfsd` if running in the kernel. In this book, we endeavor to use terms that clearly distinguish these cases.

The *rpc.nfsd* Program

The *`rpc.nfsd`* supports only one option and also takes one argument. By default, `nfsd` listens on the reserved port 2049. If you want it to listen on a different port (for testing or security reasons), say 2050, run it as `rpc.nfsd -p 2050`.

When a client contacts `nfsd`, the server processes that client's request until completion. While the server processes that client request, other clients cannot contact the server. To solve this problem, multiple instances of the NFS server are usually started. For Linux, these instances are kernel threads. That way, the NFS server can process several requests at once. By default, Red Hat starts eight `nfsd` threads. If you have a particularly busy NFS server, and you wish to start 30 threads, you can run `rpc.nfsd 30` on that server.

NFS Version 1
Why did the first NFS protocol start at 2? Was there ever an NFS protocol version 1? Yes, there was one. It was a prototype NFS server used internally by Sun Microsystems at their labs. It was never released to the public. Most traces of any documentation or sources it had appear to have been lost over the past two decades. It was the same time that the RPC system was being developed as well. As Sun engineers were developing NFS, their first prototype changed enough that they decided to give it a new number. The main use of having two NFS protocol versions at that time was to make sure the RPC system and the NFS server were capable of handling multiple versions of the same RPC service, and that they could fall back to older versions if newer ones did not exist.

NFS Version 2

The first NFS protocol ever released publicly (in the early 80s) was version 2. Many of the procedures of this protocol have corresponding system calls. Recall that the key unit that describes a file is the NFS file handle. In the descriptions of the 18 protocol procedures below (including the NULL procedure) in this version, we will emphasize the purpose of the file handle where appropriate:

NFS_NULL

This is the ping procedure for testing to see if the `nfsd` is up and responding for a given protocol version.

NFS_GETATTR

This procedure gets the attributes of a file, such as the owner, group, and mode bits.

NFS_SETATTR

This procedure sets the attributes of a file, similar to what `chmod`, `chgrp`, and `chown` can do.

NFS_ROOT

This procedure is obsolete—it was never used or implemented. Its original intent was to return the root file handle of a file system, but this functionality was moved to the MOUNT protocol's MOUNTPROC_MNT function.

NFS_LOOKUP

This is one of the key procedures and it is invoked more often than other procedures. You give this procedure two key pieces of information: a file handle for a directory, and the name of a file you wish to find in that directory. The LOOKUP procedure checks to see if the file exists in that directory. If it does not, you get back an error code. If it does exist, this procedure returns a new file handle for the file just looked up. You can use this new file handle, for example, as an argument to the READ procedure in order to read data from that file.

NFS_READLINK

This procedure returns the value of a symbolic link, or what it points to. The returned value is usually an arbitrary string that the NFS client has to process one component at a time—an action also called *traversing a symlink*.

NFS_READ

This procedure reads a number of bytes from a file, given the file's handle, a start offset to read, and the number of bytes to read. You will notice that nowhere in the NFS protocol are there explicit procedures for opening, seeking into, or closing a file; this is due to the server's statelessness and the need for each procedure to be idempotent. A normal Unix kernel keeps state after you `open` a file; this state is captured by a file descriptor data structure that includes a pointer to the current read head within a file. That way, a `read()` system call can find the point where it left off on the last read and just read the next few bytes. But with NFS, we cannot keep state. That is why each NFS_READ call must specify the exact offset to begin reading from.

NFS_WRITECACHE

This is the second and last obsolete procedure in NFSv2. Its original intent was to improve the performance of the NFS server by allowing the server to cache data without writing it to disk (at a risk of data loss should the server crash). The idea was that the NFS server could cache data, and that the NFS client would know about it, and because of this, it would not assume that the data was written to stable storage. Then, the client would issue a WRITECACHE procedure to ensure that the data got written to disk. That way, the server could cache data and the client could force a bulk write of the data to disk.

This procedure was not implemented in NFSv2 for several reasons. First, it would have made the NFS server statefull. Second, the I/O daemon provided an alternate method for improving performance by clustering writes on the client's side. Third, it was thought that the NFSv2 protocol would evolve and this procedure could be implemented in the next version of the protocol. As it turned out, this procedure was not really implemented in NFSv3 either, and it was not until NFSv4 that serious performance issues were addressed at the protocol level.

NFS_WRITE

This procedure is very similar to the READ procedure above: it writes a number of bytes to a file from a given offset. Again, this procedure is idempotent for the same reasons that the READ procedure is.

NFS_CREATE

This procedure is akin to an `open` or `creat` system call. It creates a new file in a directory specified by the directory's file handle and returns the new file handle for the new file.

NFS_REMOVE

This procedure deletes a file from a directory, just as the `unlink` system call does.

NFS_RENAME

This procedure renames a file just as the `rename` system call does.

NFS_LINK

This procedure creates a hard-link to an existing file, just as the `link` system call does.

NFS_SYMLINK

This procedure creates a symbolic link, just as the `symlink` system call does.

NFS_MKDIR

This procedure creates a new directory, just as the `mkdir` system call does.

NFS_RMDIR

This procedure deletes an existing directory, just as the `rmdir` system call does.

NFS_READDIR

This procedure reads a number of entries in a directory, just as the `readdir` system call does. Note that the procedure does not necessarily return all of the entries in a directory. It may only return a subset of those and an indicator to the client that the client must pass back to the server. That opaque indicator helps the NFS server find where to resume reading directory entries (again, because the server must not keep any state).

NFS_STATFS

This procedure provides statistics on a remote file system, such as its size, how much of it is used, and how much remains. This is similar to the `statfs` system call and is used by programs such as `df`.

For the most part, the NFSv2 protocol returns to clients error codes that resemble normal system call return codes. A zero indicates success. Other error codes include NFSERR_PERM (permission denied), NFSERR_NOENT (no such entry), and so on. The most popular error code that was newly added specifically for NFS was NFSERR_STALE. This one is returned when the NFS server is unable to decode the file handle that the client passed to it, a condition known as a *stale file handle*. A stale file handle occurs most often when an NFS server's file system is reformatted or mounted differently, but it could also occur as a result of network corruption or even a security break-in.

Like any protocol designed for the first time, NFSv2 was not without its own ambiguities and problems. One of the most serious ambiguities was in the precise meaning of the file handle—the file identifier that NFS servers create and pass to clients. Clients must not try to interpret the file handle's internal meaning (byte by byte). However, the protocol did not address the relationship between identical file handles and identical files. Specifically, it was not made clear if two identical file handles from the same server represent the same file, a hard-linked file by another name, or otherwise. It was also not made clear if two different file handles must represent different files. These and other concerns were addressed in future revisions of the NFS protocol.

NFS Version 3

More than a decade after the initial release of the NFS protocol, the next version was released by Sun. During that decade, NFS has gained immense popularity and was deployed on numerous systems. Experienced users and vendors alike were demanding a new protocol that would address serious problems with the NFSv2 protocol. With NFSv3, Sun made a good effort in addressing some of the most serious performance and security deficiencies in the protocol. The major changes in the protocol and its implementation included the following:

- Support for TCP transports as well as UDP. NFSv2 used only UDP because TCP was deemed too slow and costly at the time. Since TCP is a reliable transport and UDP is not, however, the NFSv2 protocol had to build its own reliability mechanism on top of UDP, which complicated its implementation. With TCP, much of that complication was removed.
- Support for 64-bit file systems. NFSv2 only handled 32-bit file systems, which limited the maximum file size to 2GB. NFSv3 greatly increases the maximum file size that can be used over NFS, more than eight billion gigabytes!
- Longer file handles. NFSv3 doubled the size of the file handle to 64 bytes. This was primarily done to make it more difficult for attackers to guess or fake file handles.
- Since NFSv2 used UDP, it limited the maximum number of bytes that could be transferred at once to 8KB. NFSv3 extends this range to 64KB. Ironically, the UDP specification allows up to 64KB bytes in a packet, but the original RPC implementation used by NFSv2 limited packet sizes to 8KB.
- To improve security, NFSv3 supports Kerberos authentication.
- To improve performance, new operations were created and file attributes are automatically returned on most calls, greatly reducing the number of times that some of the more popular procedures had to be invoked.

- To further improve performance, the NFSv3 server is allowed to cache data. This requires that NFSv3 clients know of this and are able to ask the server to commit cached data to stable storage.

The NFSv3 protocol has 22 procedures (including the NULL procedure), which we have listed below. Most of the procedures did not change. When a procedure remained basically the same, we did not describe it at length. Two unused procedures in NFSv2 were removed: ROOT and WRITECACHE.

NFSPROC3_NULL

Test if the `nfsd` for version 3 is up and running.

NFSPROC3_GETATTR

Get the attributes of a file.

NFSPROC3_SETATTR

Set the attributes of a file, such as owner and mode bits.

NFSPROC3_LOOKUP

Find a file in a directory. This procedure was improved by returning the mode of the directory in which it looked up the file. That way a client could tell if the mode bits of the directory had changed remotely, especially if they had changed so that the user is no longer allowed access.

NFSPROC3_ACCESS (new to NFSv3)

Test for access to a file, even if the file resides on a server that does not use traditional Unix mode bits, such as ACLs. Version 2 of the NFS protocol was very Unix centric, and assumed that all access to files was controlled by traditional Unix mode bits.

In addition, this procedure avoids problems with root users accessing files over NFSv2; in NFSv2, those root users could have their UID (0) mapped to nobody (for security reasons). But in NFSv2 this UID mapping was not handled consistently and could result in partial failures to access files by root. The new ACCESS procedure took care of that because NFS clients can test explicitly if access is granted before trying to read or write files.

NFSPROC3_READLINK

Return the value of a symbolic link.

NFSPROC3_READ

Read bytes from a file. This procedure was improved to better handle reading near the end of the file, where a read request may return fewer bytes than it was asked for.

NFSPROC3_WRITE

Write bytes from a file. This procedure was changed in two ways. First, it can indicate to clients that fewer bytes were written than were requested. Second, the write could be performed asynchronously, meaning that the data may remain cached in the server and a success status code is returned to the NFSv3 client.

NFSPROC3_CREATE

Create a new file. This procedure was improved to allow the creation to fail if the file already existed, a condition that could happen if multiple NFS clients try to create the same file at the same time.

NFSPROC3_MKDIR

Create a new directory.

NFSPROC3_SYMLINK

Create a symbolic link to a file.

NFSPROC3_MKNOD (new to NFSv3)

Create special files such as block and character devices. In NFSv2, creation of these files was done as special cases of the CREATE procedure.

NFSPROC3_REMOVE

Delete a file.

NFSPROC3_RMDIR

Delete a directory.

NFSPROC3_RENAME

Rename a file.

NFSPROC3_LINK

Create a hard-link to a file.

NFSPROC3_READDIR

Read a number of entries for a directory. This procedure was improved in NFSv3 to extend the number of bits used to describe the offsets of directory entries in a directory, thus allowing for better interoperability with other systems.

NFSPROC3_READDIRPLUS (new to NFSv3)

Improve the performance for some of the most popular operations. Often, when users list the contents of a directory, they run `ls -l`. To accomplish this in NFSv2, first the contents of the directory had to be retrieved (READDIR) and then the attributes of each entry had to be found (GETATTR). Most READDIR operations in NFSv2 were followed by a flurry of GETATTR requests.

The READDIRPLUS improves the performance of this common operation. In one message, it returns a list of entries in a directory as well as the attributes for each. This greatly reduces server and network load.

NFSPROC3_FSSTAT (renamed in NFSv3)

Provides the same statistics on a remote file system as the older STATFS procedure performed in NFSv2, such as its current and maximum size, total and current number of available inodes (which was not always available in NFSv2), etc.

NFSPROC3_FSINFO (new to NFSv3)

Helps interoperability between different systems. It returns to the caller information about the remote file system, such as whether the file system supports symlinks and hard links, what the preferred and maximum read and write sizes are, and more.

NFSPROC3_PATHCONF (new to NFSv3)

Returns file system parameters about the server, such as the maximum allowed length for a path name, the maximum length for a file name, etc. (added to comply with POSIX standards and especially the `pathconf` system call).

NFSPROC3_COMMIT (new to NFSv3)

Instructs an NFS server to flush all cached data onto stable storage. Since the WRITE procedure in NFSv3 allows asynchronous writes, this COMMIT procedure became necessary so that NFS clients could ensure that the data they sent was written reliably. Together, the asynchronous WRITE and COMMIT procedures help to improve the performance of NFS in version 3.

On a quiet local area network with little stress on NFS servers, NFSv2 may work just as well as NFSv3. But you will most likely prefer NFSv3 if one or more of these conditions exist:

- You want your Linux system to interoperate with many other systems.
- You want to increase the reliability of your NFS service, through the use of a better-defined protocol.
- You want better performance for busy servers.
- Your networks exhibit high latency or performance that varies constantly.

The next version of the NFS protocol, NFSv4, greatly improves over NFSv3 and changes it significantly. For one, the server is no longer stateless and other RPC protocols (MOUNT, RQUOTA, locking and status) are now integrated into the protocol. The NFSv4 standard is now complete, but it is still new. A few prototype implementations exist, including one for Linux. We discuss NFSv4 in more detail in Chapter 6, "NFS Version 4." Next, we show a full example of the operation of NFS with existing protocols.

An Example

Now that we have discussed all of the various components of the NFS system individually, as well as the protocols used, we are ready to examine how the system works in practice. We show this through a simple example: a client named `moon` that wishes to access a file system named `/home` from a server named `earth`.

Start by setting up the NFS server. First, make sure that all the right programs are running. The steps to start the various NFS services on the server are in Listing 1.7.

Listing 1.7: Enabling and starting NFS server services

```
[root]# chkconfig nfslock on
[root]# chkconfig nfs on
[root]# /etc/rc.d/init.d/nfslock restart
[root]# /etc/rc.d/init.d/nfs restart
```

Next, we have to allow the client host to mount the NFS server's file system. Do this by configuring a file named `/etc/exports`, as seen in Listing 1.8.

Listing 1.8: An example NFS server /etc/exports file

```
/home       moon(rw)
```

The `/etc/exports` entry from Listing 1.8 tells the NFS server to allow access to the `/home` file system to a host named `moon`. Right after the hostname, in parentheses, we list the permission options for host `moon`. The `rw` indicates that client host will be allowed to write to that file system.

Finally, ensure that the various NFS programs are aware of the changes to the `/etc/exports` file by running the `exportfs` program, as seen in Listing 1.9.

Listing 1.9: Exporting file systems to an NFS client

```
[root]# exportfs -rv
```

The `exportfs` program informs `rpc.mountd` of any changes in the list of allowed file systems to export. Verify that the NFS server is running by checking with `rpcinfo`, as seen in Listing 1.10.

Lisitng 1.10: Checking that all NFS server services are running

```
[ezk]$ rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp    957  status
    100024    1   tcp    959  status
    100011    1   udp    728  rquotad
    100011    2   udp    728  rquotad
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100021    1   udp   1026  nlockmgr
    100021    3   udp   1026  nlockmgr
    100021    4   udp   1026  nlockmgr
    100005    1   udp   1027  mountd
    100005    1   tcp   1024  mountd
    100005    2   udp   1027  mountd
    100005    2   tcp   1024  mountd
    100005    3   udp   1027  mountd
    100005    3   tcp   1024  mountd
```

At this stage the NFS server is set up. Now configure the NFS client. First, ensure that the client-side NFS software is available. Assuming that support for the NFS file system is either compiled into the kernel or available as a loadable module, then run the commands seen in Listing 1.11 to ensure that `nfs` appears in the listing of `/proc/filesystems`.

Listing 1.11: Checking for NFS client-side support

```
[root]# modprobe nfs
[root]# grep nfs /proc/filesystems
nodev   nfs
```

The next stage is to configure the local client's configuration files so that the remote file system is mounted at boot time. Add an entry to `/etc/fstab` to do this, as seen in Listing 1.12.

Listing 1.12: An example NFS client-side `/etc/fstab` file

```
# device     mountpoint  fs-type  options  dump  fsckorder
earth:/home  /home       nfs      rw       0     0
```

Use the `/etc/fstab` entry in Listing 1.12 to configure the NFS client to mount the `/home` partition from server `earth` onto the `/home` directory (mount point) of the local host, and then use the mount options that allow the client to read and write the remote file system. To test that the entry worked, you can either reboot your client or simply run the mount command as seen in Listing 1.13.

LIsting 1.13: Mount all NFS entries in `/etc/fstab`

```
[root]# mount -a -t nfs
```

Note that you could also mount the remote file system by hand, as seen in Listing 1.14, without the need to specify it in `/etc/fstab`.

Listing 1.14: One-time mounting an NFS server by hand

```
[root]# mount -t nfs earth:/home /home
```

After running the `mount` command, you can run `df` to see that the entry is indeed mounted, and then you could inspect the `/home` directory on the host `moon` to see what contents it has:

```
[ezk]$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/sda1               879078    820825     12840  98% /
earth:/home            6224742   5352649    554811  91% /home
[ezk]$ ls -F /home
ezk/    martha/   lost+found/
```

At this point, you can access files in `/home` as if they were local. If you wish to unmount the file system, run the command as seen in Listing 1.15.

Listing 1.15: Unmounting an NFS file system

```
[root]# umount /home
```

You will notice that unmounting an NFS file system is no different from unmounting any other file system. That is because NFS integrates so seamlessly with the rest of the system. Despite this seamlessness, NFS performs many actions in the background.

We now describe what happens when you try to access a file from a remote NFS server, say, by reading the contents of `/home/ezk/.profile`. Refer back to Figure 1.1, which shows the flow of data and operations for a read operation like the one below:

1. A user process calls the read system call.
2. The system call is translated in the kernel into an nfs_read function and the NFS client-side code is invoked.

3. The NFS client encodes the information and calls the RPC procedure NFS_READ.
4. The RPC information is encoded via XDR and transmitted over the network to the NFS server.
5. The server host receives the encoded data and decodes it via XDR.
6. An RPC message is constructed and passed on the NFS server nfsd.
7. nfsd calls nfsd_read.
8. nfsd_read is translated into a disk read via ext2_read.
9. The data is read from the disk and passed back to nfsd.
10. nfsd passes the data back in the reverse order of all previous steps until in reaches the user process on the client.

In Sum

The Network File System allows users to share files among many machines seamlessly: users do not know if they are accessing files locally or remotely. NFS was designed to interoperate with many other systems so that users could get their files no matter where they were physically stored. That is one of NFS's greatest strengths and it is what contributed to its popularity—it is the most widely used remote-access file system.

The inner workings of NFS are complex. The system is broken into components that run on the client side and components that run on the server side. Both sides are layered on top of Remote Procedures Calls, which in turn are layered on top of XDRs, which is a method for encoding data in a consistent manner that can be interpreted identically by all clients and servers regardless of the operating system they run on. To add to this, primarily because of the original stateless design of the NFS server, there are many other components that are part of NFS. These include the following: an I/O daemon for improving performance, locking and status daemons for handling file and record locks, a portmapper for brokering all RPC services, a mount daemon for authenticating mount requests, and a quota daemon for providing quota information to NFS clients. Finally, the NFS system uses several configuration and state files. In this chapter, we only described a few. We describe all configuration and state files in more detail in the next chapter.

Despite its complexity, NFS is relatively simple to use. Administrators list file systems to export in one file, and file systems to mount in another configuration file. Restarting the NFS client's and server's programs and mounting remote file systems then allows users to access those remote volumes. The greatest benefit is to users—who notice no difference if the files are local or remote.

In this chapter, we described the various components of NFS and its protocols. We also showed a simple example of its basic use. In the following chapters, we show NFS's operation and configuration in greater detail. We will also cover important issues, such as performance optimizations, security, how to debug and correct problems in your NFS system, and how to add NFS support into your Linux system.

CHAPTER 6: NFS VERSION 4

Sun engineers designed NFSv2 in the early 1980s, opening the protocol to other vendors to use, and even distributing a reference implementation. NFS has become a huge success since that time; it is the most popular distributed file system in use. Many vendors support NFS protocol versions 2 and 3 today. With such success and increased exposure, however, also came criticism regarding NFS's performance, security, flexibility, and interoperability with other systems.

When designing NFSv3 a decade later, Sun invited a few industry participants. The goals for developing NFSv3 were to complete the design and reimplementation in under a year and to address only the most important issues without making fundamental changes to the NFS protocol. These goals were achieved with the understanding that a further update of the protocol would be needed.

Because of NFS's popularity, it became apparent that the next revision of the protocol would be more significant than NFSv3 and needed to involve the larger Internet community in its design. A formal standardization process was needed. The *Internet Engineering Task Force* (IETF), a body of the *Internet Society*, had been very successful in producing quality protocol definitions for various Internet technologies. The IETF had in place a precise procedure for involving the community in the long process of proposing, defining, refining, and finally, publishing a document as a standard. These published standards are called *Request For Comments* (RFCs) and are available from `www.ietf.org`. The standardization process itself is a standard defined in RFC-2026, "The Internet Standards Process – Revision 3."

The IETF was the perfect choice to lead the charge for NFSv4. However, Sun Microsystems had long held the "NFS" trademark. So before the IETF could begin this work, Sun had to relinquish control over the NFSv4 specification and its successors. This decision is described in RFC-2339, "An Agreement Between the Internet Society, the IETF, and Sun Microsystems, Inc. in the matter of NFS V.4 Protocols."

The IETF then formed a *working group* (WG) for NFSv4, invited anyone who wanted to participate in the discussions, and ensured that the many working groups could meet at least three times a year during IETF meetings to report progress and discuss future work.

NFSv4 is very new. This chapter discusses the overall goals of the protocol and the new ideas or concepts that it introduces. We cover the new protocol messages that are exchanged in both directions between the NFS client and the server as well as the error codes that NFSv4 defines.

This chapter assumes a familiarity with NFSv2 and NFSv3 as described in previous chapters and especially in Chapter 1, "NFS Basics and Protocols." The discussion here focuses on the differences between NFSv4 and previous versions.

Finally, we report on the state of the NFSv4 implementation for Linux. Since the implementation is only at its prototype stage, there is little that can be discussed regarding system administration activities related to NFSv4. Nevertheless, a general understanding of this new protocol and its operation should give system administrators familiar with previous versions a feel for the effort involved in configuring and administering NFSv4.

Design Goals

Any good large-scale project begins with a requirements document to guide the project along. The IETF's NFSv4-WG started by declaring the overall goals of the project. These goals were outlined in RFC-2624, "NFS Version 4 Design Considerations." Published in June 1999, RFC-2624 states that the goals of the NFSv4-WG are to create a distributed file system that focuses on the following items:

Performance

File system performance should be good especially when used over wide area networks (WANs). The protocol should handle the particular problems of the Internet: longer latencies, higher congestion, and increased packet loss when compared to a local area network. Previous NFS protocols always assumed that NFS would be used in a local area network, so when it was used on a WAN, it exhibited poor performance.

Scalability

NFSv2/NFSv3 servers typically can handle at most several hundreds of clients. A single NFSv4 server should be able to handle many thousands of clients.

Accessibility

The protocol should be easier to access through various network appliances such as firewalls, load-balancers, or caching devices, and application proxies such as those using SOCKS.

Reliability and high-availability

To improve reliability, the protocol should allow a client to use multiple replicas of file servers and to cache data in a coherent manner. See the sidebar "Load-Balancing NFS Mounts?" in Chapter 10, "Automounter Maps."

Strong security

The protocol should include strong security and built-in negotiation of security features. It should include a better security mechanism than the typical Unix UID/GID model, which includes simple bit-mode permissions.

Cross-platform interoperability

The protocol should be flexible enough for different systems to interoperate. It should not favor one operating system over another: Unix, MS-DOS and Windows, MacOS, VMS, etc., should all be treated equally.

Internationalization (I18N)

The protocol should allow clients and servers from different parts of the world to work with each other properly, especially if the server and client use a different character set or language.

Extensibility

The protocol should be extensible enough to evolve without a costly redesign for a new major version release. (Four versions of NFS is enough, thank you; we'd rather avoid a fifth.)

Much work proceeded after NFSv4's design goals were laid out. The WG worked diligently to detail the protocol's proposed specification, while various groups began implementing prototypes. These prototypes were tested for functionality and interoperability during several gatherings of interested parties at *Connectathon* meetings (`www.connectathon.org`). This event is designed to be a marathon of sorts, bringing together implementers of various protocols to test how well their systems implement a protocol and also to expose any deficiencies in the protocol.

Creating precise, accurate, and functionally complete protocols has always been very difficult. Often, protocols were ratified as formal specifications without much practical experience. Later—sometimes years after the fact—serious flaws in some protocols were uncovered but could not be corrected because the protocols had already been widely deployed. The Connectathon meetings proved very useful in preventing such problems. For the first time in the history of NFS, many groups worked together to create a detailed, comprehensive protocol. Prototypes were built and tested, and experiences from these tests helped further refine the protocol's description, leading to better prototypes. This process was repeated until everyone was satisfied that the protocol was adequately stable and that it addressed the goals outlined in RFC-2624.

A year and a half later, in December 2000, the NFS version 4 protocol specification moved from a *Draft Internet Standard* to a *Proposed Internet Standard*, and RFC-3010 was born. The next stage is to have the larger Internet community comment on this proposed standard. If it is accepted, it will become an Internet standard and be assigned a number in the IETF's "STD" series.

Overview

The IETF design and development process has created a new protocol with NFSv4. To support the stated design goals, NFSv4 substantially departs from previous NFS protocols in several ways. For example, NFSv4 servers maintain state; functions that previously occurred in separate "stateful" protocols are now integrated into the NFS server. In addition, security has been enhanced. Special filehandles and new filehandle types have been developed. New file attributes and concepts such as delegations, leases, and callbacks have been added to the protocol. Also, the protocol allows future developers to extend its functionality and includes facilities for clients and servers to agree on a common set of functions that they support. These extensive changes make NFSv4 a new challenge for even experienced NFS administrators. To put the protocol's new procedures in proper context, the next several sections overview the major features of NFSv4.

A Stateful Server

The NFSv4 server is no longer stateless. Chapter 1 showed how, while the NFSv2 and NFSv3 servers were indeed stateless, many other components were not. These extra protocols were folded into a single NFSv4 protocol: MOUNT, NLM, NSM, RQUOTA, and even the PCNFS protocol.

This consolidation of protocols helped to design a coherent interaction between clients and servers. Furthermore, now the NFSv4 server uses a single well-known port (2049) for all communication; this eases its transit across firewalls and proxies.

NFSv4 contains explicit OPEN and CLOSE operations. The OPEN operation provides clients with a single point where various open-time semantics can be controlled in a consistent manner, such as opening and locking a file atomically. The CLOSE operation helps clients to inform servers when the latter can discard the state that was associated with an opened file.

Security

Several mechanisms exist for securing network traffic. Two of the more popular ones are *Secure Socket Layer* (SSL) and *Internet Protocol Security* (IPSEC). The problem with these methods is that they only work with connection-oriented transport protocols such as TCP. They do not work with UDP, and NFS has to be able to use both UDP and TCP.

Over the years, Sun Microsystems has developed an RPC security mechanism called *RPCSEC*. This is a secure method of exchanging network data but at the RPC layer, which is above the transport layer. Sun's efforts helped to evolve a general-purpose RPC-layer security protocol called RPCSEC_GSS, or the *Remote Procedure Call Security — Generic Security Services*.

RPCSEC_GSS was chosen as the security mechanism for use with NFSv4. Its advantage is that it works at the RPC layer and thus can support both UDP and TCP. Furthermore, it can be implemented for older versions of NFS.

The main features of RPCSEC_GSS that made it attractive for the NFSv4 designers were its ability to handle private keys such as those used in Kerberos 5. It supports public keys, encrypts data, includes strong authentication, and supports several security mechanisms, all of which improve the security of NFS.

NFSv4 added a special operation that allows clients to query a server for the methods of security that the server supports. This helps clients and servers negotiate automatically the types of security mechanisms that they wish to use. Such features need not be hard-coded into an implementation or have to require a complex and manual configuration. Servers and clients can implement any number of existing (or future) security methods, and the NFSv4 protocol is flexible enough to determine the best match of security features between the NFS clients and servers.

Compound Operations

Past analysis of NFSv2 traffic had shown that one of the most popular sequences of NFS messages was a READDIR followed by many GETATTR messages. This happened when users ran the common command to list a directory's contents: `ls -l`. This particular sequence was optimized in NFSv3 by the addition of a new protocol procedure called READDIRPLUS. This procedure saved the NFS client from having to exchange many RPC messages by returning the contents of a directory along with attributes for each entry—all in one RPC message.

Further analysis of NFS traffic suggested that many other sequences of procedures could be optimized. However, rather than create new procedures for each such sequence, NFSv4 introduces the concept of *compound operations*. An NFSv4 client can compose a compound operation by listing a series of NFS operations in one RPC message.

There are two advantages to compound operations. First, they save a lot of network traffic since only one RPC message is exchanged. Second, they alleviate some of the need for frequent protocol changes. Many past suggestions and some improvements to older NFS protocols involved the creation of new procedures that combined other procedures in different ways. With the compound procedure, protocol extensibility is improved.

📖 NFSv4 actually defines only two primary procedures: NULL and COMPOUND. All other *operations* are defined in terms of the COMPOUND procedure. A client using even a single operation encapsulates it inside a COMPOUND procedure. See the upcoming section "Protocol Procedures and Operations."

One typical example of what NFSv4 clients might do would be to LOOKUP, OPEN, READ, and then CLOSE a file. This would let a client read a whole (presumably small) file in one message. These four operations would be combined into one COMPOUND procedure and sent to the NFSv4 server. The server interprets COMPOUND procedures by evaluating each composed operation in order until an error occurs or the end of the compound procedure is reached. The server then returns to the client the last success or error code, as well as the results for all of the successful operations.

The server evaluates each compound procedure independently, so that the same client or multiple clients can send different compound procedures and there is no confusion as to their interpretation. One way that the NFSv4 server achieves this is by using a *saved filehandle* for evaluating compounded operations. Evaluation of the operations in a compound procedure requires a temporary filehandle to avoid interfering with the primary filehandle that the server uses for the client, which is called the *current filehandle*. Additional details about filehandles are provided in upcoming sections.

File System Model

In NFSv4, a file system on the server is still represented as a hierarchy of files and folders. However, this is now decoupled from actual local file systems that exist on the server. Previous versions of NFS could export only one file system at a time, or a portion thereof. NFSv4 is able to create logical file system volumes that are composed of several physical file systems or their subdirectories.

The NFSv4 server is expected to provide *glue* between physical file systems where gaps in the namespace may exist. For example, if an NFSv4 server exports a single logical volume that combines `/usr` and `/usr/local/bin`, the server may have to create a glue filehandle for the `local` component, so that an NFS client can traverse from `/usr` down to `local/bin` transparently.

Filehandles

The NFSv4 filehandle, just as with all previous versions of NFS, is a unique file identifier that is constructed by the server and is opaque to the client. In previous versions of NFS, only one filehandle existed. This filehandle had to be exchanged many times between clients and servers. In addition, there were several ambiguities about the precise meaning of the filehandle on some systems. NFSv4 expands the definition of the filehandle by creating several filehandles, several types for each filehandle, and by creating new ways to save on repeated exchanges of these filehandles over the network.

Several special filehandles exist in NFSv4: current, saved, root, and public.

The Current Filehandle

Unix operating systems maintain a *current working directory* (CWD) for each running process on the system. This CWD is the default location from which all file access using relative path names is assumed to begin. Similarly, NFSv4 defines a *current* filehandle as the default filehandle that the server keeps on behalf of the client. That way the client does not need to include a filehandle for all operations; this saves on network use.

NFSv4 clients can reset this current filehandle using the PUTROOTFH operation, just as Unix users can `cd` to a new directory and begin working off of there.

The Saved Filehandle

The *saved* filehandle is another filehandle available on NFSv4 servers. It is used in various operations as temporary filehandle storage while manipulating the current filehandle.

For example, the COMPOUND procedure, described above, saves the current filehandle in the saved filehandle location while processing compound operations. The current filehandle typically gets restored from the saved filehandle when the compound operation finishes.

The Root Filehandle

The *ROOT* filehandle designates the root (top-level directory) of the exported volume. Recall that for the client, an exported volume appears as one directory hierarchy; on the server, this could be any combination of several physical file systems or subdirectories. That’s why the root filehandle is often thought of as the logical head of the conceptual top-level filehandle of a directory tree: the server may have to create the root filehandle dynamically and there may not be an actual directory on the server that directly corresponds to this root filehandle.

In previous NFS protocols, clients authenticated to the server using a separate MOUNT protocol. This is not needed in NFSv4 since clients authenticate directly to the server. When an NFSv4 client authenticates itself to a server, the client may instruct the server to set the value of the current (default) filehandle to that of the volume's root filehandle, using the PUTROOTFH operation. This operation sets the top-level directory entry from which the client can begin to access the server's files. Afterward, the client could begin traversing the entire directory tree of that volume using lookup operations and others.

The Public Filehandle

The *PUBLIC* filehandle is a special filehandle used by the server to authenticate clients, in lieu of the older MOUNT protocol. The server is responsible for the exact definition of the public filehandle and what file system objects it may be associated with.

Typically, the PUBLIC filehandle is a zero-length or all-zero filehandle. This special value cannot be a regular filehandle. When a server receives such a filehandle, it knows that a client is trying to authenticate for the first time. The server is then responsible for determining if the client should be allowed the access and, if so, for sending the client back a successful response. The client can then get the ROOT filehandle of the file system just authenticated for, or simply set to begin file access from the root of that file system (using the PUTROOTFH operation).

NFSv4 clients cannot assume anything about this association; they can, however, send a LOOKUP request using the predesignated public filehandle to an NFSv4 server—to request first-time access to the file server's volumes. Note that servers can choose to change the public filehandle's definition and hand it to a select set of clients; this can be useful in order to avoid causal attempts to attack an NFSv4 server by probing for access using the typical PUBLIC filehandle.

WebNFS
WebNFS is a protocol designed by Sun Microsystems as an extension to NFSv2 and NFSv3. Its main goal is to allow access to NFS servers across the Internet without the need for an explicit MOUNT protocol and without changing the existing protocols.
WebNFS achieves this goal by defining a special *PUBLIC FILEHANDLE* to be used with the NFS_LOOKUP procedure. The PUBLIC filehandle is defined as containing all zeros in NFSv2 and as having a zero length in NFSv3. A WebNFS client contacts a remote NFS server directly at port 2049 using standard-style Internet URLs and provides the remote NFS server with the PUBLIC filehandle. Upon receiving this filehandle, the remote NFS server authenticates the NFS client just as the MOUNT protocol does. The successful response from the NFS_LOOKUP call includes an actual filehandle to use for normal NFS operations.

Traditional NFS path traversal looked up each directory in a long path name for the component under that directory. This resulted in a series of NFS_LOOKUP requests sent over the network. A good Internet-wide protocol should endeavor to reduce the number of network messages used. To improve performance over the Internet, WebNFS also supports multicomponent lookups. A client sends a whole path name to a server; the server interprets the complete path name and returns a filehandle for the final component.
The design of and the experimentation with WebNFS provided some useful feedback on the feasibility of NFS use over the Internet. The lessons learned from WebNFS were incorporated in the NFSv4 protocol.

Filehandle Types

Past NFS protocols assumed that filehandles were persistent: they must be valid at all times until the client is done with them. This became a problem for two reasons. First, some operating systems—especially non-Unix ones—lacked the information necessary to encode filehandles consistently. For example, their file systems may not use unique and constant inode numbers, or the file system ID could change after a reboot. Second, vendors who wanted to implement NFS replication (read-only), migration, or load-balancing faced serious obstacles since a filehandle on one server was not valid on another server, even for the same file.

NFSv4 defines two main types of filehandles: persistent and volatile. *Persistent* filehandles have the same semantics as filehandles in older NFS protocols. *Volatile* filehandles, on the other hand, may become invalid at any point. When that happens, the server returns the typical "stale filehandle" error code to the client. If the file system was renamed or migrated, the client can then query the file server for the new location of that file system. The client is then responsible for retrieving an updated filehandle for the file in question.

NFSv4 also supports expiration times on volatile filehandles. A server can issue timed filehandles to clients and be assured that clients could not use them after a certain period of time. The flexibility of filehandles in NFSv4 is neatly captured by the five different types that can be encoded as a type field bitmask in the filehandle itself:

FH4_PERSISTENT

This bit indicates that the filehandle is persistent.

FH4_NOEXPIRE_WITH_OPEN

This filehandle cannot expire while a client has the file open.

FH4_VOLATILE_ANY

This filehandle could expire at any time, especially during file system renaming or migration.

FH4_VOL_MIGRATION

This filehandle will expire during file system migration.

FH4_VOL_RENAME

This filehandle will expire during file system renaming.

One additional change to filehandles in NFSv4 was a semantic one. NFSv4 clarified the relationship between filehandles and files when it came to their equality or inequality. Two identical filehandles must represent the same exact file storage, even for files that are hard-linked. Two different filehandles must represent two different files.

File Attributes

Older versions of NFS defined a fixed set of mostly Unix-centric file attributes. There was no easy way to support non-Unix file servers or create new attributes. NFSv4 includes a flexible mechanism for supporting many platforms as well as creating new attributes.

NFSv4 defines three types of file attributes: mandatory, recommended, and named. We discuss these attribute types next.

Mandatory Attributes

Mandatory attributes are those that all servers and clients must define. The set of mandatory attributes is intended to be kept very small and forms a basis on which all NFSv4 systems can interact with each other.

Mandatory attributes include the type of a file (e.g., regular file, directory, symlink, etc.), the size of the file, a unique identifier for the file, and more. Some attributes are specific to a whole file system, such as a flag indicating whether the file system supports symbolic or hard links.

Recommended Attributes

Recommended attributes typically represent differences between various operating systems and file systems that need not be available in all NFSv4 implementations. An NFSv4 client has to query an NFSv4 server and find out which attributes (if any) are supported by both sides. The intent of this type of attributes is to allow servers to implement as many of them as possible, as long as it is simple to do so. Attributes that are difficult to implement should not be simulated or half done: it is better for a feature not to exist than to work inconsistently.

Recommended attributes include the "archived" and "hidden" file bits of MS-DOS file systems, Access Control Lists (ACLs), whether the file system is case-insensitive, quota information (thus subsuming the older RQUOTA protocol), and more.

Together, mandatory and recommended attributes represent the total set of file and file system attributes that exist on most systems used today. Both sets of attributes are defined and a bitmask is allocated for them, one bit per attribute.

Access Control Lists

Access Control Lists (ACLs) exist in some operating systems and file systems. They allow more flexible control than the simplistic Unix UID/GID model for who can access file system resources. Unfortunately, ACLs have been implemented in a variety of non-interoperable ways by different vendors. One of NFSv4's goals is to provide a flexible cross-platform way of specifying and using ACLs.

One special recommended attribute is the ACL attribute. This attribute specifies an array of *access control entries* (ACEs). ACEs define an access type, the files or directories it should apply to, and the actual access set (a single owner, a group, everyone, anonymous users, dial-up users, and more). Users, for example, could be represented in a universal fashion (such as an e-mail address). There are four types of possible ACEs:

`ALLOW`

Grant access as defined in the ACE.

`DENY`

Deny access as defined in the ACE.

`AUDIT`

Log any attempt to access any file or directory that uses the access method specified by the ACE.

`ALARM`

Generate an alarm on attempt to access any file or directory that uses the access method defined by the ACE.

Named Attributes

Named attributes allow NFSv4 to evolve and be extended easily. A named attribute has a simple string name and a value that is an arbitrary sequence of bytes. One use for these attributes is in the creation of application-specific file attributes. For example, a compression system could associate the type or compression algorithm used with the file, or a source-control system could attach version numbers to source files.

Named attributes are accessible to an NFSv4 client as an attributes directory for each file that has named attributes. The attributes directory contains files whose names are the named attributes' names and whose contents represent the values of the named attributes. In this way, named attributes can be listed, created, read, and modified easily with normal directory browsing tools (such as `ls` and `cat`).

Named attributes may themselves have attributes, even named ones. This allows for the creation of a whole hierarchy of attributes for a given file.

File Locking

File and byte-range locking methods used to be in a separate protocol, the Network Lock Manager (NLM), and included complex callback procedures. In addition, the PCNFS protocol included share reservation messages used typically by Windows-based systems. NFSv4 folds all locking and share-reservation support right into the protocol and does away with most of the locking-related callbacks. Since locks require that the server maintain state about the locked files, a simpler lease-based model was taken. This system allows for a wide range of locking semantics to be supported reliably—anywhere from advisory read-only locks to mandatory single-writer locks.

When the server creates any new state on behalf of a client, it provides the client a lease for that state for a period of time. This period of time is defined by the server globally for a given client. The client is responsible for renewing the lease, either explicitly by calling a RENEW procedure or implicitly by accessing the resource for which the lease was provided (such as reading from or writing to a file that is locked).

If the client does not renew the lease, the server may discard the state associated with that lease. A client that tries to access files with expired leases will get an error code.

Client Caching

NFSv4's primary goal is to provide good performance over the Internet. To that effect, the protocol minimizes the amount of communication that is needed. One method already mentioned is the COMPOUND procedure. Another obvious technique is caching. Previous versions of NFS also employed client-side caching to improve performance.

File data is cached by default. With such extensive caching, the chances for caches to get out of sync are increased. Older versions of NFS did not include cache consistency mechanisms; NFSv4, on the other hand, supports and describes cache consistency methods in detail. The client is responsible for ensuring cache consistency when it opens or closes a file. Applications that wish to enforce cache consistency at all times (and possibly bypass the cache) have to lock the byte ranges of the file in question.

Delegations

One important contribution to improving performance in NFSv4 is the concept of a *delegation*. An NFSv4 server can, upon opening a file, grant an NFSv4 client a read or write delegation, as well as several other types of delegations such as locks. This allows the client a measure of independence while manipulating the file. Clients can open, lock, read, write, and close files during a delegation—all without having to use costly network resources to communicate with the server.

With a read delegation, the server guarantees the client that no other client can write to that file during the lifetime of the delegation. With a write delegation, the server guarantees the client that no other client can read or write to that file.

Delegations, however, have a limited lifetime. Furthermore, delegations may be revoked or recalled by the server when it believes that a granted delegation conflicts with a more important request from a different client. In a marked departure from previous NFS protocols, NFSv4 servers may *themselves* initiate calls to NFS clients in order to recall a delegation. This reverse procedure-calling path is known as a *callback path*. If the callback path does not exist between the server and the client, servers will not grant that client any delegations. We describe the NFSv4 callback procedures later, under the section “Callback Procedures.”

One possible problem with delegations, or for that matter, any state that either client or server maintains, is what happens when either party fails. If the server crashes, it will lose knowledge of any delegations it gave to clients. If a client crashes, it will lose any information about locks it held before. The problem is exacerbated by the Internet, where networks are less reliable and can become partitioned for a period of time. To address these situations, NFSv4 defines a method of *delegation recovery*. A server may, at any time, contact the client and ask it to relinquish its delegation. This could be useful, for example, for a server that crashed and came back up, to consolidate all of its known delegations before resuming regular file system activity.

Internationalization (I18N)

Older versions of NFS only supported the U.S. ASCII character set for file names. If NFSv4 is to cross international boundaries, the protocol must support character sets used throughout the world. This is particularly important because with NFSv4, a client could be accessing files that reside on a server in a different country, possibly one with a very different language and alphabet. Such a client would have to understand both the local and remote language character sets and possibly translate between the two sets, a process known as *internationalization* (I18N).

One proposed idea for I18N was for NFSv4 servers and clients to negotiate a locale before exchanging file names. Afterward, both parties could agree on a small subset of characters to use. This small set could be encoded in just 7 bits for most languages. However, several complications arose during the design of NFSv4, especially with multicomponent lookups where each pathname component could use a different locale. The idea of using a locale with each NFS operation was dropped in favor of a simpler method, albeit one that could consume more network bandwidth.

NFSv4 uses universal 16-bit or even 32-bit character sets, sometimes known as Unicode sets, or UTF-8. Sixteen-bit Unicodes have been determined to support all characters of all known languages. They are unique and simple and clearly identify the language to which the particular character belongs. Thirty-two-bit codes may be needed to support additional language character sets—perhaps when the intergalactic space station is completed (we may also have to revise I18N to support purely telepathic species).

Normalization is defined as the process by which a client and server agree on a base set of characters and then only use that set, thus saving on network bandwidth. For example, most NFSv4 clients and servers within the U.S. can easily fall back to using the ASCII character set. The first minor version of the NFSv4 protocol (version 0) does not specify how normalization should take place, nor does it require it. This is intended to be addressed in future revisions. Therefore, at this stage, NFSv4 clients and servers have to be able to handle un-normalized characters. If they choose to, or require it, such clients or servers can normalize the UTF-8 characters they get as needed.

Minor Versioning

Past NFS protocol revisions were *major*; the protocol procedures were changed significantly and were incompatible with previous versions. Typically, major revisions are assumed to be needed once a decade (yes, start worrying about NFSv5 come 2010).

In another departure from previous NFS versions, NFSv4 allows for *minor* revisions such as NFSv4.1, NFSv4.2, etc. Minor revisions are assumed to be standardized quicker, in about 1 to 2 years.

Many features of NFSv4 allow users to extend it without a formal change: compound operations and named attributes, for example. The IETF, however, recommends that the community using NFSv4 periodically reevaluate the need for small revisions since, in all likelihood, certain new features might turn out to be sufficiently useful that they should be standardized. Such features as named attributes should be standardized and given an official number through the *Internet Assigned Numbers Authority* (IANA).

The rules for minor revisions of the NFSv4 (and all future major versions) are as follows:

- New primary procedures may not be added or removed.
- No existing compound operations may be removed.
- New compound operations may be added. Since these compound operations may result in a semantically different NFS protocol for different minor versions, clients must not use filehandles or any objects returned from a compound procedure where the client's minor version was different from the server's minor version.
- Existing attributes may not be changed or removed. New attributes may be added by appending them at the end of the current list of attributes.
- No existing data structures, bit flags, attributes, returned results, or error codes may be changed or deleted.
- Semantics of all existing operations, returned results, and error codes must remain as is.
- New error codes can be defined.
- Minor versions can define an operation, an attribute, or a bit flag as "mandatory to not implement." All existing structure for the operation is kept intact, but the operation or object is obsoleted. This allows the reintroduction of the operation or object at a later date and avoids the possibility of reuse of the operation's or object's freed slot.
- Features may be downgraded from mandatory to recommended or from recommended to optional, or they may be upgraded in the reverse (but only one upgrade level at a time).
- No new features can be added as mandatory in a minor revision. They can be introduced as optional or recommended and then upgraded to mandatory in a subsequent revision.
- Clients and servers supporting a minor revision must support all previous minor revisions for the same major release.

Protocol Procedures and Operations

At the heart of any protocol are the procedures that make up the specification. Table 6.1 lists the procedures and operations of NFSv4. This protocol has only two primary procedures: NULL and COMPOUND, which are procedure 0 and procedure 1, respectively. Procedure 2 is undefined and reserved for future expansion. The rest are protocol operations that can only be encapsulated in a COMPOUND procedure.

Table 6.1: NFSv4 Protocol Procedures and Operations

Number	Operation	Meaning
Procedure 0	NULL	No operation.
Procedure 1	COMPOUND	Compound operations.
Procedure 2	N/A	(For future expansion.)
Operation 3	ACCESS	Check access rights.
Operation 4	CLOSE	Close file.
Operation 5	COMMIT	Commit cached data.
Operation 6	CREATE	Create a non-regular file object.
Operation 7	DELEGPURGE	Purge delegations awaiting recovery.
Operation 8	DELEGRETURN	Return delegation.
Operation 9	GETATTR	Get attributes.
Operation 10	GETFH	Get current filehandle.
Operation 11	LINK	Create link to a file.
Operation 12	LOCK	Create lock.
Operation 13	LOCKT	Test for lock.

Operation 14	LOCKU	Unlock file.
Operation 15	LOOKUP	Look up file name.
Operation 16	LOOKUPP	Look up parent directory.
Operation 17	NVERIFY	Verify difference in attributes.
Operation 18	OPEN	Open a regular file.
Operation 19	OPENATTR	Open named attribute directory.
Operation 20	OPEN_CONFIRM	Confirm open.
Operation 21	OPEN_DOWNGRAD E	Reduce open file access.
Operation 22	PUTFH	Set current filehandle.
Operation 23	PUTPUBFH	Set public filehandle.
Operation 24	PUTROOTFH	Set root filehandle.
Operation 25	READ	Read from file.
Operation 26	READDIR	Read directory.
Operation 27	READLINK	Read symbolic link.
Operation 28	REMOVE	Remove file system object.
Operation 29	RENAME	Rename directory entry.
Operation 30	RENEW	Renew a lease.
Operation 31	RESTOREFH	Restore saved filehandle.
Operation 32	SAVEFH	Save current filehandle.

Operation 33	SECINFO	Obtain available security.
Operation 34	SETATTR	Set attributes.
Operation 35	SETCLIENTID	Negotiate clientid.
Operation 36	SETCLIENTID_CONFIRM	Confirm clientid.
Operation 37	VERIFY	Verify same attributes.
Operation 38	WRITE	Write to file.

Next, we describe the NFSv4 RPC messages shown in Table 6.1. For more details on each procedure, or for the precise source code definitions for these, see RFC-3010 and Appendix B, "Online Resources."

NULL

This is the standard ping procedure that takes and returns no arguments. It is intended to run very quickly because it is used more often by clients to test if a server is up and responding.

COMPOUND

The COMPOUND procedure combines several NFS operations into a single RPC message. As described above in "Compound Operations," the NFSv4 server evaluates each operation in the compound in order, until all operations have been evaluated or an error occurs. The server returns the last error status as well as the results for all intermediate operations.

ACCESS

This is the first operation in the NFSv4 protocol. It verifies that a user, as specified by the user's credentials, has access to the file or directory in question. The NFS client can specify the bitmask of access rights to check, so that the NFS server can check only for those rights.

CLOSE

This operation closes a file. The server releases any state information and share reservations that exist for the file. Clients are expected to release all locks held prior to closing a file. If they do not release locks, the server may try to free up the locks itself. If the server is unable to free all the locks, it returns an error message for the CLOSE operation.

COMMIT

This operation flushes all unwritten data to stable storage, for a given file. Clients may specify the starting offset and length of bytes (a range) within the file to flush. If the offset and length are both zero, the server will flush the entire file.

One typical problem with asynchronous writes is the possibility of a server crash between an asynchronous write and a commit operation. Servers return a special identifier to clients upon a successful asynchronous write, called a *write verifier*. The same write verifier is returned to the client upon a successful commit. If clients receive a different write verifier, they know that, somehow, the previously written data was lost and not written to stable media. This can often happen when a server reboots between a write and commit operation.

CREATE

This operation creates a non-regular file. To create regular files, clients must use the new OPEN operation. Clients must specify the type of object to be created: a directory, symbolic link, character device, or block device, etc.

DELEGPURGE

Typically, when a server delegates access to a client, and either one of them crash, both parties have to recover their previous information about the delegations they provided or received. However, there are cases when not all delegations need to be recovered after a client or server crash. If a client determines that a delegation it lost was no longer needed, it can use the DELEGPURGE operation to tell the server to remove all of the delegations pending recovery for that client. This can happen when a client does not need to store information relating to the delegation into stable storage locally on the client. By purging such delegations, servers can clear up some of the state they hold and even unblock pending operations for other clients (possibly for the same shared resource).

DELEGRETURN

Each delegation has a special *state identifier* (stateid) that the server hands back to the client. The DELEGRETURN operation simply returns to the server a delegation given its stateid. This operation is often the final stage in returning a delegation to a server that had recalled it from a client using the CB_RECALL callback operation (described below).

GETATTR

This operation returns the attributes for a given object. The client can specify the exact set of attributes it is interested in (because there could be many). The server returns a bitmap of the attributes for which it was able to get values, as well as the values for these attributes.

GETFH

This operation returns the current filehandle back to the client. This operation may be needed after a LOOKUP or CREATE since they do not automatically return new filehandles. See the section “Filehandles” above.

LINK

This operation creates a new name for a given file, known in Unix file systems as a hard-link. Of course, servers running on platforms that do not support hard-links will return an error code back to a client that requests the creation of a new link for an existing file.

LOCK

This operation requests a lock for a byte range in a file, specified using the starting offset and the length of the byte-range to lock. Clients can use special syntax to specify the locking of a file from a given offset through the end of the file, or the entire file. If a lock cannot be granted, the server returns as much information as it can about the conflict: the owner, offset, and length of the byte-range in the conflicting file. This way clients can tell which other client already has a lock on the file.

LOCKT

This operation does not set a lock; it tests to see if a lock exists for a file at a given offset and length. Note that the server does not guarantee that a different client may not acquire a lock shortly after a LOCKT.

LOCKU

This operation instructs a server to release (unlock) the locked resources specified by the given stateid for the file in question.

LOOKUP

This operation finds a file in a directory corresponding to the current filehandle. It supports multicomponent lookups, thus saving on repeated lookups for each component.

LOOKUPP

This operation specifically finds the parent directory of the current filehandle. Previous versions of NFS required special semantics to the LOOKUP procedure when looking up “`..`” or “`.`”. However, this *dot-dot* concept is Unix-centric and does not exist on all operating systems. Therefore, the LOOKUPP operation must be used to ensure cross-platform compatibility; servers would implement this operation according to their own specifics.

NVERIFY

This operation can be used primarily to verify the validity of a cache. Suppose a client wants to find out if a cached file changed and, if so, return the new data bytes of the file. The client can send a sequence of <LOOKUP, NVERIFY, READ>, where the NVERIFY is asked to check for the size attribute of the file and the last modification time as the client knows them. If the file has not changed, NVERIFY will return an error indicating that the attributes checked for are still the same. If, however, the file did change, then NVERIFY will not return an error, and the compound procedure would proceed to process the READ operation—thus reading the updated bytes of the file.

`OPEN`

This complex operation opens a regular file, possibly creating it. The result of an OPEN leaves a state on the server, a state that is normally released using a CLOSE operation. Clients must issue an explicit GETFH operation to retrieve the filehandle to CLOSE. The OPEN operation, just as with the `open(2)` system call, can create files exclusively if they do not exist, or reopen existing files. One additional complication of this operation is the client's ability to claim to the server that it may have already held a previous lock or delegation.

`OPENATTR`

NFSv4 supports extensible attributes, designated as a hierarchy of named attributes and their values. To access these attributes, clients perform the OPENATTR operation. This operation returns a filehandle for a special virtual directory that contains the named attributes of the given file system object. The client can then issue normal READDIR, LOOKUP, READ, and WRITE operations on the attribute directory—since it appears as any other normal directory. This provides a very flexible method for extending and manipulating the attributes of a file.

`OPEN_CONFIRM`

During the time that a client opens and uses a locked file, the client may generate a lot of additional state information that the server is obligated to maintain. Each time a client generates such additional state information, it will typically include a sequence ID number along with the information. To help NFSv4 servers save on the amount of memory and resources consumed by possibly many such pieces of state data, the client may periodically issue the OPEN_CONFIRM operation. The client passes a sequence ID number to the server; the server may then discard much of intermediate state information for that client's use of the file.

`OPEN_DOWNGRADE`

This operation is used to reduce a client's access to an open file. It may be necessary if a client that locked a file has the file opened multiple times and possibly is using different access or deny permissions. When one of the opened references is closed, the file's permanent access restrictions may be changed in such a way as to conflict with some of the remaining opened references. This procedure is useful to instruct a server to replace the access and deny bits of an opened file with those specified by the client.

PUTFH

This operation replaces the current filehandle with the handle specified by the client. It is useful, for example, when clients change to different directories or other operations where the clients wish to change the context for the following operations.

PUTPUBFH

This operation replaces the current filehandle with the one that represents the server's public filehandle. This operation is often the first NFSv4 operation used, to set the initial context for future operations. Recall that the public filehandle can be different from the root filehandle, as detailed in "Filehandles" above.

PUTROOTFH

This operation replaces the current filehandle with the one that represents the server's root filehandle. This operation is also often the first NFSv4 operation used, to set the initial context for future operations. See "Filehandles" above.

READ

This operation reads a number of bytes, starting at a given offset, for the file represented by the current filehandle.

READDIR

This operation returns a number of entries in a directory, starting from a given offset in that directory (which is often the last place where a previous READDIR left off). The client may specify a list of attributes to return for each entry. This operation in NFSv4 subsumes the capabilities of NFSv3's READDIRPLUS procedure.

READLINK

This operation reads the value of a symbolic link, or what the symlink points to. If the server does not support symlinks, it returns an error.

REMOVE

This operation deletes a file system object from the directory corresponding to the current filehandle. If that is the last reference to the object, the server may destroy any information associated with that file object. This operation can remove files, directories, or any other type of file.

RENAME

This operation renames a file. The old and new names of the file are given as UTF-8 strings. The old directory where the file resides is stored in the current filehandle, and the new directory location is stored in the saved filehandle.

RENEW

When clients receive a lease from a server, for example, a lease to write a file exclusively, that lease expires after the designated period of time. Clients must issue the RENEW operation as needed to renew their lease; otherwise the server may timeout the lease and free it.

RESTOREFH

This is one of several operations that are used to manipulate the various filehandles that servers can have; such operations include setting and copying values of filehandles between different handles. This operation copies the saved filehandle contents into the current filehandle.

SAVEFH

This operation copies the current filehandle into the saved filehandle location. It is in essence the opposite of the RESTOREFH operation.

SECINFO

This operation returns a list of valid available RPC authentication techniques that the server supports for a given filehandle. In this fashion, clients and servers can negotiate the most suitable forms of security for their needs.

SETATTR

This operation changes a set of attributes for a given file system object. The client provides the server with a bitmask of attributes to change along with their values.

SETCLIENTID

Each NFSv4 client is identified to a server using a *client ID*, which includes a possible callback path: an RPC program number and a port number. This operation tells a server that the client wants to use a different client ID for subsequent operations. It can be used, for example, by NFSv4 clients that act as proxies, caching devices, or load-balancers.

SETCLIENTID_CONFIRM

This operation confirms that a client identifier given in a previous SETCLIENTID operation is still valid. It may be necessary to use this operation if the server holds state for a given client ID because that state should not be released for a client that has changed its ID.

VERIFY

This operation is used to ensure that the attributes of a file are the same before proceeding with the next operation. For example, a client can issue a sequence <LOOKUP, VERIFY, PUTFH, REMOVE> to remove a file. However, the client issues the VERIFY before removing the file to check, for example, that the file's size and owner are what the client expects. If the attributes are not the same, the VERIFY operation would return an error and the file would not be removed: the compound procedure will terminate before completing all operations.

WRITE

This operation writes data to a regular file represented by the current filehandle. The client can specify the offset and length of bytes to write to the file.

An NFSv4 Implementation RFC
Past NFS protocols attempted to specify design and implementation information in the same RFC. Worse, often many implementation recommendations were not discussed at all, leading vendors to implement past protocols in incompatible ways.
The NFSv4-WG left many implementation details unanswered in RFC-3010. This was intentional, so that the main RFC would only concern itself with the design and specification of the NFSv4, not how it might be implemented. For example, for callbacks to work through firewalls and NAT boxes, these devices will have to be modified to be aware of NFSv4, so they can properly allow such access through the firewall and into a secure site. In addition, load-balancers will have to figure out how to migrate or replicate NFSv4 files for use with this new protocol. These implementation details, and many more, are the subject of an upcoming NFSv4-WG RFC—an *implementation* RFC—that the working group is scheduled to work on next.

Callback Procedures

Because of the complexity of NFSv4, the server may require access to the client. That is, the client itself must act as a server under certain conditions. The client's ability to service requests is done via the NFS4_CALLBACK program. This program is just another RPC program: it has a set of procedures and operations. However, there is no preassigned RPC program number and port number for the callback program. The client provides the server with its callback program number and port numbers via the SETCLIENTID operation.

In a similar fashion to the NFSv4 server program, the NFS4_CALLBACK client program also defines only two primary procedures: CB_NULL and CB_COMPOUND. All other callback operations are defined in terms of the CB_COMPOUND procedure. Table 6.2 lists all callback procedures and operations.

Table 6.2: NFSv4 Callback Procedures and Operations

Number	Operation	Meaning
Procedure 0	`CB_NULL`	No operation.
Procedure 1	`CB_COMPOUND`	Compound operations.
Procedure 2	N/A	(For future expansion.)
Operation 3	`CB_GETATTR`	Get attributes.
Operation 4	`CB_RECALL`	Recall an open delegation.

Next, we describe the NFSv4 callback RPC messages shown in Table 6.2. For more details on each procedure, or for the precise source code definitions for these, see RFC-3010 and Appendix B.

`CB_NULL`

This is the standard ping procedure. The server uses this procedure to verify that a callback path to the client exists.

`CB_COMPOUND`

This procedure is a wrapper for any number of operations. Its behavior is similar to the NFSv4 COMPOUND procedure.

`CB_GETATTR`

This is the first callback operation. When a client holds a delegation on a file, it can perform read and write operations that change the attributes of the file—such as its size. If another client asks the server for the attributes of that file, the server has to call the original client using the CB_GETATTR operation to find out the most up-to-date attributes of the file before responding to the second client.

`CB_RECALL`

Using this operation, a server asks a client to relinquish a delegation. The client has to process the recall as soon as possible and then return the delegation using the DELEGRETURN operation.

Error Messages

One of the ways to understand any protocol is to look at the error conditions that can occur. Many of the NFS error messages in this and previous protocols resemble standard error conditions such as those listed in the header file `/usr/include/asm/error.h`. In this section we summarize the error conditions of the NFSv4 protocol. Note that these errors can be returned by servers to clients, as well as by clients to servers (in response to a callback procedure).

NFS4_OK

The operation completed successfully.

NFS4ERR_ACCES

The caller does not have the proper permission to perform the operation requested.

NFS4ERR_BADHANDLE

The server could not decode the filehandle, possibly due to an internal consistency failure.

NFS4ERR_BADTYPE

The server does not support the type of object that the client tried to create.

NFS4ERR_BAD_COOKIE

The *cookie* (an internal identifier) of a READDIR operation is invalid.

NFS4ERR_BAD_SEQID

Each locking request must identify itself using a sequence number that is either the last one or a new one that is one more than the last one used. If a different sequence number is used, the server responds with this error.

NFS4ERR_BAD_STATEID

Each state that either clients or servers maintain has a unique identifier (stateid) that is exchanged between them. If a stateid was used that the other party does not recognize, it returns this error.

NFS4ERR_CLID_INUSE

A client tried to use the same client ID (via SETCLIENTID) that another client already uses.

NFS4ERR_DELAY

Occasionally, a client will send a request to a server and the server will begin processing it. However, the server may realize that the request is taking a long time to process; for example, when retrieving a file that has been stored on a backup tape as part of a Hierarchical Storage Management (HSM) system. In that case, the server responds with this error, telling the client that the request is taking longer and that the client should retry it later.

NFS4ERR_DENIED

Trying to lock a file failed. This could be a temporary condition and the client should try again later, since the other client who held the lock might release it.

NFS4ERR_DQUOT

The hard quota limits have been exceeded.

NFS4ERR_EXIST

The file already exists. This could happen when a client tries to create a file exclusively and the file already exists.

NFS4ERR_EXPIRED

The client's lease has expired.

NFS4ERR_FBIG

The server is unable to perform the operation because it would result in growing the file beyond the server's limits.

NFS4ERR_FHEXPIRED

A volatile filehandle has expired.

NFS4ERR_GRACE

When a server crashes and then restarts, there is a period of time when it is initializing and cannot respond to some operations. During that time and similar times, the server will return this error.

NFS4ERR_INVAL

The client passed an operation that included an invalid argument, or the server cannot support the operation requested.

NFS4ERR_IO

A hard I/O error (such as a failing disk) occurred.

NFS4ERR_ISDIR

A non-directory operation was applied to a directory.

NFS4ERR_LEASE_MOVED

The lease has moved or migrated to a new server.

NFS4ERR_LOCKED

The client tried to read or write a file that is locked by another client.

NFS4ERR_LOCK_RANGE

This error occurs when a client requests a lock for a portion of a file that overlaps a portion locked by another client.

NFS4ERR_MINOR_VERS_MISMATCH

The server does not support the minor revision of the protocol requested.

NFS4ERR_MLINK

The file has too many hard-links.

NFS4ERR_MOVED

The file system was moved or migrated to another server. The server can find out the new location for the file system by requesting the `fs_locations` attribute of the current filehandle.

`NFS4ERR_NAMETOOLONG`

The name of the file is too long.

`NFS4ERR_NODEV`

The device does not exist.

`NFS4ERR_NOENT`

The file or directory does not exist.

`NFS4ERR_NOFILEHANDLE`

The current filehandle has not been properly set.

`NFS4ERR_NOSPC`

The file system is full.

`NFS4ERR_NOTDIR`

The directory operation was applied to a non-directory.

`NFS4ERR_NOTEMPTY`

Cannot remove the directory because it is not empty.

`NFS4ERR_NOTSUPP`

The operation is not supported.

`NFS4ERR_NOT_SAME`

The attributes requested in the VERIFY operation do not match those on the server.

`NFS4ERR_NXIO`

The device or address does not exist or had an I/O error.

`NFS4ERR_OLD_STATEID`

An older state identifier was used.

`NFS4ERR_PERM`

The operation is not permitted because it was not performed by the superuser or the file’s owner.

`NFS4ERR_READDIR_NOSPC`

The space provided by the client for filling in a READDIR request is not sufficient.

NFS4ERR_RESOURCE

While processing a COMPOUND procedure, the host ran out of resources (such as memory).

NFS4ERR_ROFS

A file system–modifying operation was attempted on a read-only file system.

NFS4ERR_SAME

The attributes requested in the NVERIFY operation match those on the server.

NFS4ERR_SERVERFAULT

An unknown error occurred on the server and it cannot be mapped to any predefined NFS error code.

NFS4ERR_SHARE_DENIED

Cannot open a file with a share reservation because a share is already reserved by another client.

NFS4ERR_STALE

The filehandle is invalid, the file referenced by it does not exist, or access to that file was revoked.

NFS4ERR_STALE_CLIENTID

The server does not recognize the client ID sent to it by a locking request or a confirmation request for setting a client ID.

NFS4ERR_STALE_STATEID

The state ID used was generated by an earlier or older server.

NFS4ERR_SYMLINK

This error is returned if the client tries to open a symlink file or look up a file in a directory that is specified by a symlink. The client is supposed to recursively traverse symlinks by getting their values explicitly (using READLINK) and then passing the non-symlink value to a LOOKUP or OPEN operation.

NFS4ERR_TOOSMALL

The space provided in a buffer or request is too small.

NFS4ERR_WRONGSEC

The client used a security mechanism that is not supported by the server's security policy.

NFS4ERR_XDEV

The client tried to create a hard-link across a different device.

Linux Implementations

The Center for Information Technology Integration (CITI) at the University of Michigan has been working on a prototype implementation of an NFSv4 server and client for Linux and NetBSD. CITI's first goal for the Linux port was to support the most mandatory NFSv4 features; this goal was achieved in early 2000. The group then went on to complete the Linux port, based on the 2.2.*x* kernel and the existing NFSv3 client/server code. The status of these ports is available from CITI's NFSv4 Web page at `www.citi.umich.edu/projects/nfsv4/`.

This section describes the procedures for retrieving, configuring, installing, and running the prototype implementation of NFSv4 for Linux. Note that the procedures described here are likely to change often, and the NFSv4 support in Linux is highly experimental—alpha quality at best. If these procedures do not work for you, refer to CITI's Web site and the NFS mailing list at `nfs@nfs.sourceforge.net`. For general guidelines of how to build and install a Linux kernel and user-level software, refer to Chapter 7, "Building and Installing the Linux Kernel and NFS Software."

1. Start by downloading the following three packages from CITI's download page: `www.citi.umich.edu/projects/nfsv4/download/`. Store them in `/usr/src`.

   `linux_nfsv4.tar.gz`

   Linux 2.2.14 kernel source and the NFSv4 additions.

   `redhat_mount.tar.gz`

   Linux `mount-2.9o` source that contains the NFSv4 additions.

   `rpcsec_gss.tar.gz`

   Linux RPCSEC_GSS user-level library and the GSS daemon.

2. Then download the server side utilities from `www.citi.umich.edu/projects/nfsv4/sept_2000_rel/server_util/server_utils.tar.gz`.

3. Unpack the sources:

   ```
   [root]# cd /usr/src
   [root]# tar xzf linux_nfsv4.tar.gz
   [root]# tar xzf redhat_mount.tar.gz
   [root]# tar xzf rpcsec_gss.tar.gz
   ```

4. Configure and build the kernel in `/usr/src/linux_nfsv4`. When configuring the kernel, make sure you select the SUNRPC, NFS_FS, and NFSD modules, but not NFSv3 support.

   ```
   [root]# cd /usr/src
   ```

```
[root]# mv linux linux.old
[root]# ln -s linux_nfsv4 linux
[root]# cd linux
[root]# make menuconfig
[root]# make dep
[root]# make clean
[root]# make bzImage
[root]# make modules modules_install
```

5. Manually enable NFSv4 in your kernel configuration file:

```
[root]# echo "CONFIG_NFS_V4=y" >> .config
[root]# make bzImage modules
[root]# cp arch/i386/boot/bzImage /boot/vmlinuz-nfsv4
[root]# cp System.map /boot/System.map-nfsv4
```

6. Add a new kernel boot entry to the end of your `/etc/lilo.conf` file, which looks as follows:

```
image=/boot/vmlinuz-nfsv4
        label=linux-nfsv4
        read-only
        root=/dev/hda1
```

   Of course, `/dev/hda1` is just an example. Make sure that your `root` entry matches your host's root disk.

7. Run `/sbin/lilo`.

8. Configure and build the linux mount program that supports NFSv4.

```
[root]# cd /usr/src/redhat_mount
[root]# ln -s ../setproctitle.o lib/setproctitle.o
[root]# make
[root]# install -s -c -m 755 mount /usr/sbin/mount-nfsv4
```

9. Install MIT's Kerberos version 5. If you are using Red Hat 7, insert the installation CD-ROM and install the following RPMs:

```
[root]# cd /mnt/cdrom/RedHat/RPMS
[root]# rpm -Uvh krb5-server-1.2.1-8.i386.rpm
[root]# rpm -Uvh krb5-devel-1.2.1-8.i386.rpm
[root]# rpm -Uvh krb5-libs-1.2.1-8.i386.rpm
[root]# rpm -Uvh pam_krb5-1-19.i386.rpm
[root]# rpm -Uvh krb5-workstation-1.2.1-8
```

10. Build and install the GSS server:

```
[root]# cd /usr/src/rpcsec_gss
[root]# ./configure
[root]# make
[root]# make install
[root]# install -s -c -m 755 gssd/gssd /usr/sbin/gssd
```

Server-Side Support

Unpack and install server-side utilities (only needed for NFSv4 servers):

```
[root]# mkdir /usr/src/server_utils
[root]# cd /usr/src/server_utils
[root]# tar xzf ../server_utils.tar.gz
[root]# install -c -m 755 exportfs /usr/sbin/exportfs
[root]# install -c -m 755 rpc.nfsd /usr/sbin/rpc.nfsd
[root]# install -c -m 755 nfsv4 /etc/rc.d/init.d/nfsv4
```

Turn on NFSv4 services and start the servers:

```
[root]# chkconfig nfsv4 on
[root]# /etc/rc.d/init.d/nfsv4 restart
```

Client-Side Support

Follow the general instructions to build and install the NFSv4 software and utilities, and then start the GSS server and load up the proper modules.

```
[root]# /usr/sbin/gssd &
[root]# insmod sunrpc
[root]# insmod nfs
```

Now you can use the special `/usr/sbin/mount-nfsv4` program to mount an NFSv4 server:

```
[root]# /usr/sbin/mount-nfsv4 server:/home /mnt
```

📖 NFSv4 was designed to work across many platforms, including Windows. The Windows world uses the *Server Message Block* (SMB) protocol to share files and printers with other Windows systems. Also, through the use of SAMBA—a Unix-based SMB server—Windows and Unix systems can share files and printers. While NFSv4 is expected to integrate file access in a better way between all Unix and non-Unix systems, NFSv4 was not designed to share printers.

In Sum

NFSv4 is a major redesign of the NFS protocol. The goals of this protocol are to provide good performance over the Internet, work well with many clients, work across firewalls and network appliances, be highly reliable and available, include strong security, interoperate with any platform, support international character sets, and extend easily.

This complex new protocol subsumes all previous protocols. It includes lots of support for clients and servers to maintain caches and share state, as well as checking those for consistency and coherency.

Great care was taken to allow for many possible optimizations. Since the Internet is a wide area network and subject to high latency and greater loss of packets, the protocol ensures that only the bare essential information is exchanged over the network. The protocol allows servers to delegate access to clients for a duration of time, so the servers need not be involved with the client at all for that time.

The protocol also defines a new concept of a compound procedure: a wrapper routine that encapsulates many other operations. Such a composition is sent as a single RPC message to the server and processed all at once.

To avoid having to redesign this protocol again in just a couple years, extensibility methods were built into the protocol. Clients and servers can, for example, create new named attributes for file system objects. Through minor revisions of the protocol, new operations and attributes can be created, and much more functionality can be changed, as long as it complies with the rules set forth for minor revisioning.

The NFSv4 protocol is the first NFS protocol on its way to becoming an Internet standard. For the first time in its history, an NFS protocol design was opened to the whole community and is guided by the IETF’s strict rules for creating new protocols. The full detailed description of NFSv4 is available in RFC-3010 and spans over 200 pages. Happy reading.