All seven principles we discussed in the first part of this book apply to distributed file systems, and we discuss each of them for the example systems in this chapter. We conclude with a comparison of the various file systems.

10.1 SUN NETWORK FILE SYSTEM

We start our discussion on distributed file systems by taking a look at Sun Microsystem's **Network File System**, universally known as **NFS**. NFS was originally developed by Sun for use on its UNIX-based workstations, but has been implemented for many other systems as well. The basic idea behind NFS is that each file server provides a standardized view of its local file system. In other words, it should not matter how that local file system is implemented, each NFS server supports the same model. This model comes with a communication protocol that allows clients to access the files stored on a server. This approach allows a heterogeneous collection of processes, possibly running on different operating systems and machines, to share a common file system.

NFS has a relatively long history. The first version was never released and was kept internal to Sun. The second version of NFS was incorporated into Sun's operating system SunOS 2.0, and is described in (Sandberg et al., 1985). It took several years before version 3 was released (Pawloski et al., 1994). These two versions are described in detail by Callaghan (2000).

NFS version 3 is now undergoing major revisions that will lead to the next version (see Shepler et al., 2000). These revisions will primarily allow better performance across the Internet, turning NFS into a true wide-area file system. Other improvements are related to security and interoperability. In the following, we examine in detail the various aspects of NFS, concentrating on versions 3 and 4.

10.1.1 Overview of NFS

As we mentioned, NFS is not so much a true file system, as a collection of protocols that together provide clients with a model of a distributed file system. In this sense, NFS is comparable to CORBA, which essentially also exists only in the form of a specification. Like CORBA, there are various implementations of NFS, running on different operating systems and machines. The NFS protocols have been designed in such a way that different implementations should easily interoperate. In this way, NFS can run on a heterogeneous collection of computers.

NFS Architecture

The model underlying NFS is that of a **remote file service**. In this model, clients are offered transparent access to a file system that is managed by a remote server. However, clients are normally unaware of the actual location of files.

Instead, they are offered an interface to a file system that is similar to the interface offered by a conventional local file system. In particular, the client is offered only an interface containing various file operations, but the server is responsible for implementing those operations. This model is therefore also referred to as the **remote access model**. It is shown in Fig. 10-1(a).

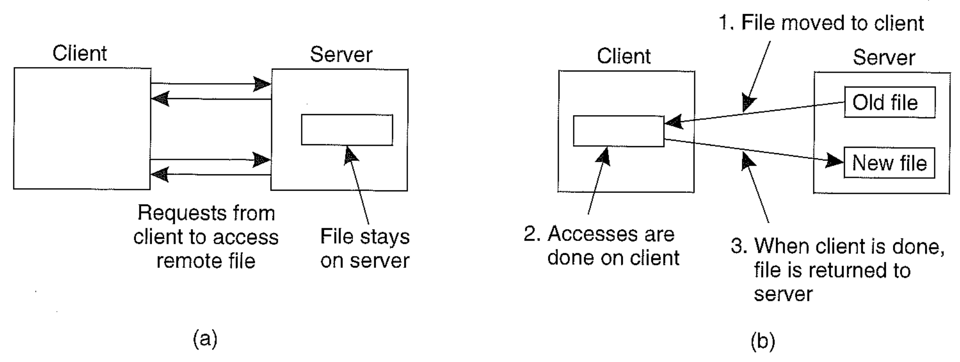

Client
Server
Requests from client to access remote file
File stays on server
(a)
1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server
(b)

Figure 10-1. (a) The remote access model. (b) The upload/download model.

In contrast, in the **upload/download model** a client accesses a file locally after having downloaded it from the server, as shown in Fig. 10-1(b). When the client is finished with the file, it is uploaded back to the server again so that it can be used by another client. The Internet's FTP service can be used this way when a client downloads a complete file, modifies it, and then puts it back.

NFS has been implemented for many different operating systems, although the UNIX-based versions are predominant. For virtually all modern UNIX systems, NFS is generally implemented following the layered architecture shown in Fig. 10-2.

A client accesses the file system using the system calls provided by its local operating system. However, the local UNIX file system interface is replaced by an interface to the **Virtual File System (VFS)**, which by now is a defacto standard for interfacing to different (distributed) file systems (Kleiman, 1986). Operations on the VFS interface are either passed to a local file system, or passed to a separate component known as the **NFS client**, which takes care of handling access to files stored at a remote server. In NFS, all client-server communication is done through RPCs. The NFS client implements the NFS file system operations as RPCs to the server. Note that the operations offered by the VFS interface can be different from those offered by the NFS client. The whole idea of the VFS is to hide the differences between various file systems.

On the server side, we see a similar organization. The **NFS server** is responsible for handling incoming client requests. The RPC stub unmarshals requests and the NFS server converts them to regular VFS file operations that are subsequently

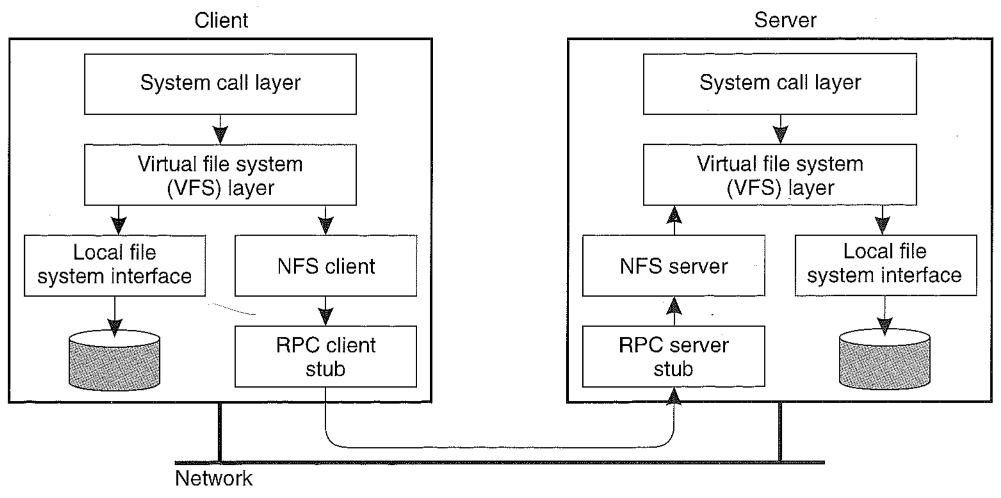

Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

Figure 10-2. The basic NFS architecture for UNIX systems.

passed to the VFS layer. Again, the VFS is responsible for implementing a local file system in which the actual files are stored.

An important advantage of this scheme is that NFS is largely independent of local file systems. In principle, it really does not matter whether the operating system at the client or server implements a UNIX file system, a Windows 2000 file system, or even an old MS-DOS file system. The only important issue is that these file systems are compliant with the file system model offered by NFS. For example, MS-DOS with its short file names cannot be used to implement an NFS server in a fully transparent way.

File System Model

The file system model offered by NFS is almost the same as the one offered by UNIX-based systems. Files are treated as uninterpreted sequences of bytes. They are hierarchically organized into a naming graph in which nodes represent directories and files. NFS also supports hard links as well as symbolic links, like any UNIX file system. Files are named, but are otherwise accessed by means of a UNIX-like **file handle**, which we discuss in detail below. In other words, to access a file, a client must first look up its name in a naming service and obtain the associated file handle. Furthermore, each file has a number of attributes whose values can be looked up and changed. We return to naming in detail below.

Fig. 10-3 shows the general file operations supported by NFS versions 3 and 4, respectively. The create operation is used to create a file, but has different meanings in version 3 and 4. In version 3, the operation is used for creating regular files. Nonregular files are created using separate operations. The link operation

is used to create hard links. symlink is used to create symbolic links. Mkdir is for creating subdirectories. Special files, such as device files, sockets, and named pipes are created by means of the mknod operation.

This situation is changed completely in version 4. In version 4, create is used for creating *nonregular* files, which include symbolic links, directories, and special files. Hard links are still created using a separate link operation, but regular files are created by means of the open operation, which is new to NFS and is a major deviation from the approach to file handling in older versions. Up until version 4, NFS was designed to allow its file servers to be stateless. For reasons we discuss later in this chapter, this design criterion has been abandoned in version 4, in which it is assumed that servers will generally maintain state between operations on the same file.

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Get the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

Figure 10-3. An incomplete list of file system operations supported by NFS.

The operation rename is used to change the name of an existing file.

Files are deleted by means of the remove operation. In version 4, this operation is used to remove any kind of file. In previous versions, a separate rmdir operation was needed to remove a subdirectory. A file is removed by its name and has the effect that the number of hard links to it is decreased by one. If the number of links drops to zero, the file may be destroyed.

Version 4 allows clients to open and close (regular) files. Opening a nonexisting file has the side effect that a new file is created. To open a file, a client provides a name, along with various values for attributes. For example, a client may specify that a file should be opened for write access. After a file has been successfully opened, a client can access that file by means of its file handle. That handle is also used to close the file, by which the client tells the server that it will no longer need to have access to the file. The server, in turn, can release any state it maintained to provide that client access to the file.

The lookup operation is used to look up a file handle for a given path name. In NFS version 3, the lookup operation will not resolve a name beyond a mount point. (Recall from Chap. 4 that a mount point is a directory that essentially represents a link to a subdirectory in a *foreign* name space.) For example, assume that the name */remote/vu* refers to a mount point in a naming graph. When resolving the name */remote/vu/mbox*, the lookup operation in NFS version 3 will return the file handle for the mount point */remote/vu* along with the remainder of the path name (i.e., *mbox*). The client is then required to explicitly mount the file system that is needed to complete the name lookup. A file system in this context is the collection of files, attributes, directories, and data blocks that are jointly implemented as a logical block device (Crowley, 1997; Tanenbaum and Woodhull, 1997).

In version 4, matters have been simplified. In this case, lookup will attempt to resolve the entire name, also if this means crossing mount points. Note that this approach is possible only if a file system has already been mounted at mount points. The client is able to detect that a mount point has been crossed by inspecting the file system identifier that is later returned when the lookup completes.

There is a separate operation readdir to read the entries in a directory. This operation returns a list of *(name, file handle)*-pairs, along with attribute values that the client requested. The client can also specify how many entries should be returned. The operation returns an offset that can be used in a subsequent call to readdir in order to read the next series of entries.

Operation readlink is used to read the data associated with a symbolic link. Normally, this data corresponds to a path name that can be subsequently looked up. Note that the lookup operation cannot handle symbolic links. Instead, when a symbolic link is reached, name resolution stops and the client is required to first call readlink to find out where name resolution should continue.

Files have various attributes associated with them. Again, there are important differences between NFS version 3 and 4, which we discuss in detail later. Typical attributes include the type of the file (telling whether we are dealing with a directory, a symbolic link, a special file, etc.), the file length, the identifier of the file system that contains the file, and the last time the file was modified. File attributes can be read and set using the operations getattr and setattr, respectively.

Finally, there are operations for reading data from a file, and writing data to a file. Reading data by means of the operation read is straightforward. The client specifies the offset and the number of bytes to be read. The client is returned the actual number of bytes that have been read, along with additional status information (e.g., whether the end-of-file has been reached).

Writing data to a file is done using the write operation. The client again specifies the position in the file where writing should start, the number of bytes to be written, and the data. In addition, it can instruct the server to ensure that all data are to be written to stable storage (we discussed stable storage in Chap. 7). NFS servers are required to support storage devices that can survive power supply failures, operating system failures, and hardware failures.

10.1.2 Communication

An important issue in the design of NFS is the independence of operating systems, network architectures, and transport protocols. This independence ensures that, for example, clients running on a Windows system can communicate with a UNIX file server. This independence can be largely attributed to the fact that the NFS protocol itself is placed on top of an RPC layer that hides the differences between various operating systems and networks.

In NFS, all communication between a client and server proceeds along the **Open Network Computing RPC** (**ONC RPC**) protocol, which is formally defined in (Srinivasan, 1995a), together with a standard for representing marshaled data (Srinivasan, 1995b). ONC RPC is similar to other RPC systems as discussed in Chap. 2. Its programming interfaces and practical use are described in (Bloomer, 1992).

Every NFS operation can be implemented as a single remote procedure call to a file server. In fact, up until NFS version 4, the client was made responsible for making the server's life as easy as possible by keeping requests relatively simple. For example, in order to read data from a file for the first time, a client normally first has to look up the file handle using the lookup operation, after which it can issue a read request, as shown in Fig. 10-4(a).

This approach requires two successive RPCs. The drawback becomes apparent when considering the use of NFS in a wide-area system. In that case, the extra latency of a second RPC may lead to performance degradation. To circumvent such problems, NFS version 4 supports **compound procedures** by which several RPCs can be grouped into a single request, as shown in Fig. 10-4(b).

In our example, the client combines the lookup and read request into a single RPC. In the case of version 4, it is also necessary to open the file before reading can take place. After the file handle has been looked up, it is passed to the open operation, after which the server continues with the read operation. The overall effect in this example, is that only two messages need to be exchanged between the client and server.

There are no transactional semantics associated with compound procedures. The operations grouped together in a compound procedure are simply handled in

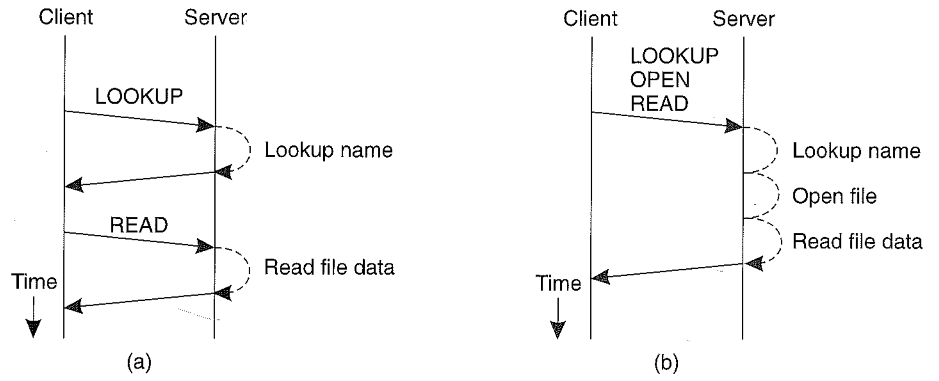


Figure 10-4. (a) Reading data from a file in NFS version 3. (b) Reading data using a compound procedure in version 4.

the order as requested. If there are concurrent operations from other clients, then no measures are taken to avoid conflicts. If an operation fails for whatever reason, then no further operations in the compound procedure are executed, and the results found so far are returned to the client. For example, if lookup fails, a succeeding open is not even attempted.

10.1.3 Processes

NFS is a traditional client-server system in which clients request a file server to perform operations on files. One of its long-lasting distinguishing features compared to other distributed file systems, was the fact that servers could be stateless. In other words, the NFS protocol did not require that servers maintained any client state. This approach was followed in versions 2 and 3, but has been abandoned for version 4.

The main advantage of the stateless approach is simplicity. For example, when a stateless server crashes, there is essentially no need to enter a recovery phase to bring the server to a previous state. However, as we explained in Sec. 7.3, we still need to take into account that the client cannot be given any guarantees whether or not a request has actually been carried out. We return to NFS fault tolerance later.

The stateless approach in the NFS protocol could not always be followed in practical implementations to its full extent. For example, locking a file cannot easily be done by a stateless server. In the case of NFS, a separate lock manager is used to handle this situation. Likewise, certain authentication protocols require that the server maintains state on its clients. Nevertheless, NFS servers could generally be designed in such a way that only very little information on clients needed to be maintained. For the most part, the scheme worked adequately.

Starting with version 4, the stateless approach was abandoned, although the protocol is designed in such a way that a server does not need to maintain much information on its clients. Besides those just mentioned, there are other reasons to choose for a stateful approach. An important reason is that NFS version 4 is expected to also work across wide-area networks. This requires that clients can make effective use of caches, in turn requiring an efficient cache consistency protocol. Such protocols often work best in collaboration with a server that maintains some information on files as used by its clients. For example, a server may associate a lease with each file it hands out to a client, promising to give the client exclusive read and write access until the lease expires or is refreshed. We return to such issues later in this chapter.

The most apparent difference with the previous versions, is the support for the open operation, which is inherently stateful. In addition, NFS supports callback procedures by which a server can do an RPC to a client. Clearly, callbacks also require a server to keep track of its clients.

10.1.4 Naming

As in any other distributed file system, naming plays an important role in NFS. The fundamental idea underlying the NFS naming model is to provide clients complete transparent access to a remote file system as maintained by a server. This transparency is achieved by letting a client be able to mount a remote file system into its own local file system, as shown in Fig. 10-5.

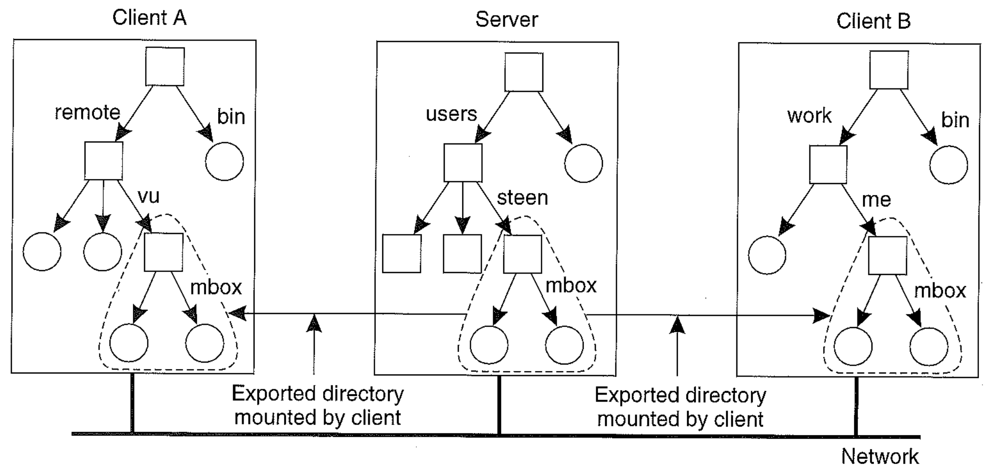

Client A
Server
Client B
remote
bin
vu
mbox
users
steen
mbox
work
bin
me
mbox
Exported directory mounted by client
Exported directory mounted by client
Network

Figure 10-5. Mounting (part of) a remote file system in NFS.

Instead of mounting an entire file system, NFS allows clients to mount only part of a file system, as also shown in Fig. 10-5. A server is said to **export** a directory when it makes that directory and its entries available to clients. An exported directory can be mounted into a client's local name space.

This design approach has a serious implication: in principle, users do not share name spaces. As shown in Fig. 10-5, the file named */remote/vu/mbox* at client *A* is named */work/me/mbox* at client *B*. A file's name therefore depends on how clients organize their own local name space, and where exported directories are mounted. The drawback of this approach in a distributed file system, is that sharing files becomes much harder. For example, Alice cannot tell Bob about a file using the name she assigned to that file, for that name may have a completely different meaning in Bob's name space of files.

There are several ways to solve this problem, but the most common one is to provide each client with a name space that is partly standardized. For example, each client may be using the local directory */usr/bin* to mount a file system containing a standard collection of programs that are available to everyone. Likewise, the directory */local* may be used as a standard to mount a local file system that is located on the client's host.

An NFS server can itself mount directories that are exported by other servers. However, it is not allowed to export those directories to its own clients. Instead, a client will have to explicitly mount such a directory from the server that maintains it, as shown in Fig. 10-6. This restriction comes partly from simplicity. If a server could export a directory that it mounted from another server, it would have to return special file handles that include an identifier for a server. NFS does not support such file handles.

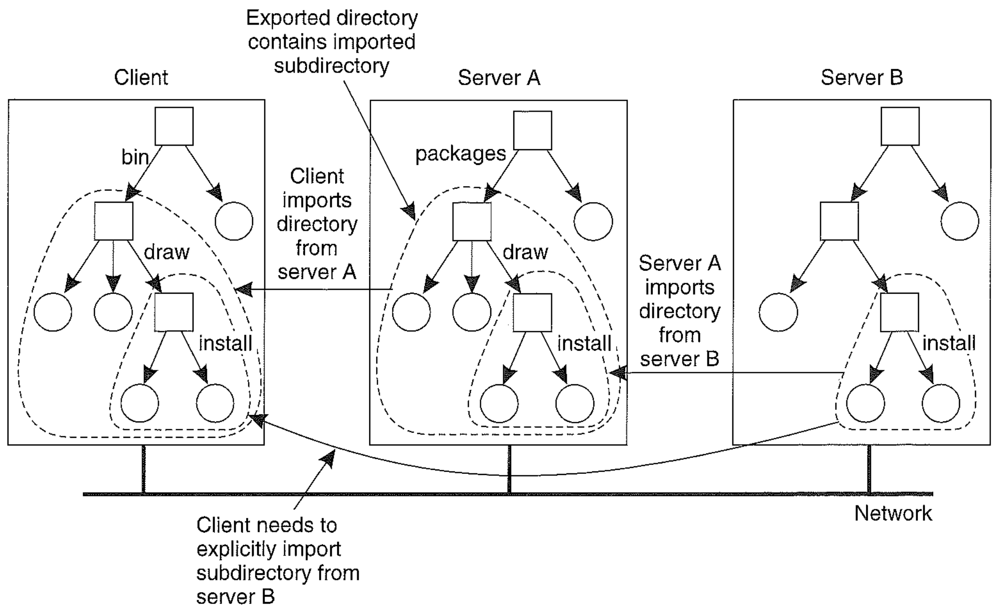

Exported directory contains imported subdirectory
Client
Server A
Server B
bin
draw
install
packages
draw
install
install
Client imports directory from server A
Server A imports directory from server B
Network
Client needs to explicitly import subdirectory from server B

Figure 10-6. Mounting nested directories from multiple servers in NFS.

To explain this point, assume that server *A* hosts a file system FS_A from which it exports the directory */packages*. This directory contains a subdirectory */draw* that acts as a mount point for a file system FS_B that is exported by server *B* and mounted by *A*. Let *A* also export */packages/draw* to its own clients, and assume that a client has mounted */packages* into its local directory */bin* as shown in Fig. 10-6.

If name resolution is iterative (as is the case in NFS version 3), then to resolve the name */bin/draw/install*, the client contacts server *A* when it has locally resolved */bin* and requests *A* to return a file handle for directory */draw*. In that case, server *A* should return a file handle that includes an identifier for server *B*, for only *B* can resolve the rest of the path name, in this case */install*. As we have said, this kind of name resolution is not supported by NFS.

Name resolution in NFS version 3 (and earlier versions) is strictly iterative in the sense that only a single file name at a time can be looked up. In other words, resolving a name such as */bin/draw/install* requires three separate calls to the NFS server. Moreover, the client is fully responsible for implementing the resolution of a path name. NFS version 4 also supports recursive name lookups. In this case, a client can pass a complete path name to a server and request that server to resolve it.

There is another peculiarity with NFS name lookups that has been solved with version 4. Consider a file server hosting several file systems. With the strict iterative name resolution in version 3, whenever a lookup was done for a directory on which another file system was mounted, the lookup would return the file handle of the directory. Subsequently reading that directory would return its *original* content, not that of the root directory of the mounted file system.

To explain, assume that in our previous example that both file systems FS_A and FS_B are hosted by a single server. If the client has mounted */packages* into its local directory */bin*, then looking up the file name *draw* at the server would return the file handle for *draw*. A subsequent call to the server for listing the directory entries of *draw* by means of readdir would then return the list of directory entries that were *originally* stored in FS_A in subdirectory */packages/draw*. Only if the client had also mounted file system FS_B, would it be possible to properly resolve the path name *draw/install* relative to */bin*.

NFS version 4 solves this problem by allowing lookups to cross mount points at a server. In particular, lookup returns the file handle of the *mounted* directory instead of that of the original directory. The client can detect that the lookup has crossed a mount point by inspecting the file system identifier of the looked up file. If required, the client can locally mount that file system as well.

File Handles

A file handle is a reference to a file within a file system. It is independent of the name of the file it refers to. A file handle is created by the server that is hosting the file system and is unique with respect to all file systems exported by the server. It is created when the file is created. The client is kept ignorant of the actual content of a file handle; it is completely opaque. File handles used to be 32 bytes in NFS version 2, but can have a variable length up to 64 bytes in version 3 and 128 bytes in version 4. Of course, the length of a file handle is not opaque.

Ideally, a file handle is implemented as a true identifier for a file relative to a file system. For one thing, this means that as long as the file exists, it should have one and the same file handle. This persistence requirement allows a client to store a file handle locally once the associated file has been looked up by means of its name. One benefit is performance: as most file operations require a file handle instead of a name, the client can avoid having to look up a name repeatedly before every file operation. Another benefit of this approach is that the client can now access the file independent of its (current) names.

Because a file handle can be locally stored by a client, it is also important that a server does not reuse a file handle after deleting a file. Otherwise, a client may mistakenly access the wrong file when it uses its locally stored file handle.

Note that the combination of iterative name lookups and not letting a lookup operation allow crossing a mount point introduces a problem with getting an initial file handle. In order to access files in a remote file system, a client will need to provide the server with a file handle of the directory where the lookup should take place, along with the name of the file or directory that is to be resolved. NFS version 3 solves this problem through a separate mount protocol, by which a client actually mounts a remote file system. After mounting, the client is passed back the **root file handle** of the mounted file system, which it can subsequently use as a starting point for looking up names.

In NFS version 4, this problem is solved by providing a separate operation putrootfh that tells the server to solve all file names relative to the root file handle of the file system it manages. The root file handle can be used to look up any other file handle in the server's file system. This approach has the additional benefit that there is no need for a separate mount protocol. Instead, mounting can be integrated into the regular protocol for looking up files. A client can simply mount a remote file system by requesting the server to resolve names relative to the file system's root file handle using putrootfh.

Automounting

As we mentioned, the NFS naming model essentially provides users with their own name space. Sharing in this model may become difficult if users name the same file differently. One solution to this problem is to provide each user with a local name space that is partly standardized, and subsequently mounting remote file systems the same for each user.

Another problem with the NFS naming model has to do with deciding *when* a remote file system should be mounted. Consider a large system with thousands of users. Assume that each user has a local directory */home* that is used to mount the home directories of other users. For example, Alice's home directory may be locally available to her as */home/alice*, although the actual files are stored on a remote server. This directory can be automatically mounted when Alice logs into her workstation. In addition, she may have access to Bob's public files by accessing Bob's directory through */home/bob*.

The question, however, is whether Bob's home directory should also be mounted automatically when Alice logs in. The benefit of this approach would be that the whole business of mounting file systems would be transparent to Alice. However, if this policy were followed for every user, logging in could incur a lot of communication and administrative overhead. In addition, it would require that all users are known in advance. A much better approach is to transparently mount another user's home directory on demand, that is, when it is first needed.

On demand mounting of a remote file system (or actually an exported directory), is handled in NFS by an **automounter**, which runs as a separate process on the client's machine. The principle underlying an automounter is relatively simple. Consider a simple automounter implemented as a user-level NFS server on a UNIX operating system (for alternative implementations, see Callaghan, 2000).

Assume that for each user, the home directories of all users are available through the local directory */home*, as described above. When a client machine boots, the automounter starts with mounting this directory. The effect of this local mount is that whenever a program attempts to access */home*, the UNIX kernel will forward a lookup operation to the NFS client, which in this case, will forward the request to the automounter in its role as NFS server, as shown in Fig. 10-7.

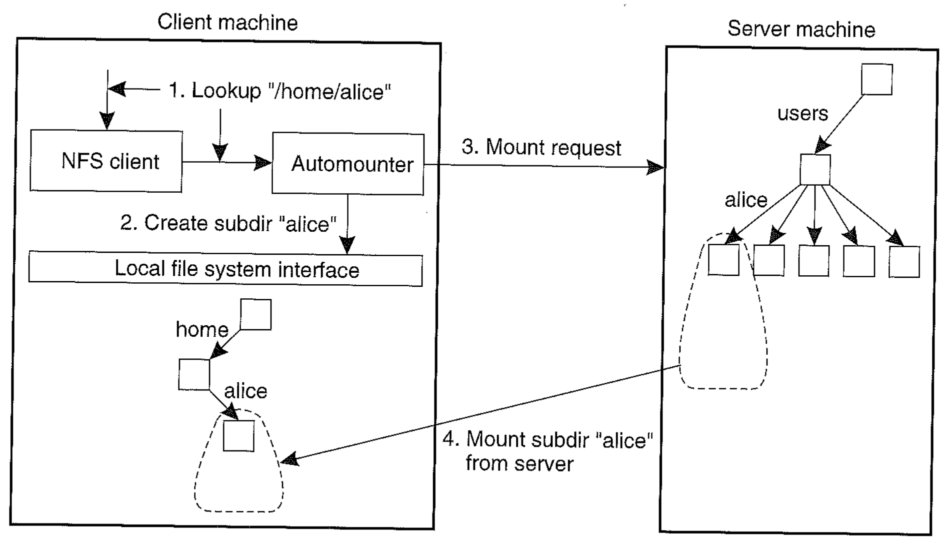


Figure 10-7. A simple automounter for NFS.

For example, suppose that Alice logs in. The login program will attempt to read the directory */home/alice* to find information such as login scripts. The automounter will thus receive the request to look up subdirectory */home/alice*, for which reason it first creates a subdirectory */alice* in */home*. It then looks up the NFS server that exports Alice's home directory to subsequently mount that directory in */home/alice*. At that point, the login program can proceed.

The problem with this approach is that the automounter will have to be involved in all file operations to guarantee transparency. If a referenced file is not locally available because the corresponding file system has not yet been mounted, the automounter will have to know. In particular, it will need to handle all read and write requests, even for file systems that have already been mounted. This approach may incur a serious performance problem. It would be much better to have the automounter only mount and unmount directories, but otherwise stay out of the loop.

A simple solution is to let the automounter mount directories in a special subdirectory, and install a symbolic link to each mounted directory. This approach is shown in Fig. 10-8.

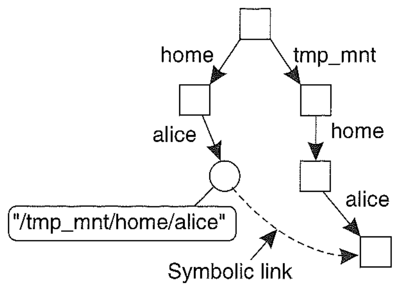


Figure 10-8. Using symbolic links with automounting.

In our example, the user home directories are mounted as subdirectories of */tmp_mnt*. When Alice logs in, the automounter mounts her home directory in */tmp_mnt/home/alice* and creates a symbolic link */home/alice* that refers to that subdirectory. In this case, whenever Alice executes a command such as

```
ls –l /home/alice
```

the NFS server that exports Alice's home directory is contacted directly without further involvement of the automounter.

File Attributes

An NFS file has a number of associated attributes. In version 3, the set of attributes was fixed and every NFS implementation was expected to support those attributes. However, because NFS has traditionally been strongly modeled along the lines of UNIX file systems, fully implementing NFS on other file systems (e.g., Windows) was sometimes difficult or even impossible.

With NFS version 4, the set of file attributes has been split into a set of mandatory attributes that every implementation must support, a set of recommended attributes that should be preferably supported, and an additional set of named attributes.

Named attributes are actually not part of the NFS protocol, but are encoded as an array of *(attribute, value)*-pairs in which an attribute is represented as a string, and its value as an uninterpreted sequence of bytes. They are stored along with the file (or directory) and NFS provides operations to read and write attribute values. However, interpretation of attributes and their values is entirely left to an application.

Fig. 10-9(a) shows a number of the mandatory file attributes (there are a total of 12). These attributes were present in previous versions of NFS. Noteworthy is the *CHANGE* attribute, which may be a server-defined attribute by which a client can see if or when a file has last been modified.

Attribute	Description
TYPE	The type of the file (regular, directory, symbolic link)
SIZE	The length of the file in bytes
CHANGE	Indicator for a client to see if and/or when the file has changed
FSID	Server-unique identifier of the file's file system

(a)

Attribute	Description
ACL	An access control list associated with the file
FILEHANDLE	The server-provided file handle of this file
FILEID	A file-system unique identifier for this file
FS_LOCATIONS	Locations in the network where this file system may be found
OWNER	The character-string name of the file's owner
TIME_ACCESS	Time when the file data were last accessed
TIME_MODIFY	Time when the file data were last modified
TIME_CREATE	Time when the file was created

(b)

Figure 10-9. (a) Some general mandatory file attributes in NFS. (b) Some general recommended file attributes.

Fig. 10-9(b) lists a number of recommended attributes. The current NFS version 4 lists a total of 43 recommended file attributes. An important attribute is an access control list, which replaces the UNIX-style permission bits. There is also an attribute listing locations where possible replicas of the file system can be found. This list can be used by a client to contact another server, for example, if the current one is unavailable. The complete list of recommended attributes is described in (Shepler et al., 2000).

10.1.5 Synchronization

Files in a distributed file system are generally shared by multiple clients. If sharing never occurred, it would not really make sense to have a distributed file system in the first place. Unfortunately, sharing comes at a price: to ensure that shared files remain consistent synchronization is needed. Synchronization is relatively simple if files are kept at a central server. Doing so, however, introduces a potential performance problem. Therefore, clients are often allowed to keep a local copy of a file while they are reading and writing its contents. Note that this implementation corresponds to the upload/download model of Fig. 10-1(b). Before we go into the details of synchronization, let us take a look at what it means to share a file.

Semantics of File Sharing

When two or more users share the same file, it is necessary to define the semantics of reading and writing precisely to avoid problems. To explain the semantics of file sharing in NFS, let us first consider some general issues related to file sharing.

In single-processor systems that permit processes to share files, such as UNIX, the semantics normally state that when a read operation follows a write operation, the read returns the value just written, as shown in Fig. 10-10(a). Similarly, when two writes happen in quick succession, followed by a read, the value read is the value stored by the last write. In effect, the system enforces an absolute time ordering on all operations and always returns the most recent value. We will refer to this model as **UNIX semantics**. This model is easy to understand and straightforward to implement.

In a distributed system, UNIX semantics can be achieved easily as long as there is only one file server and clients do not cache files. All reads and writes go directly to the file server, which processes them strictly sequentially. This approach gives UNIX semantics (except for the minor problem that network delays may cause a read that occurred a microsecond after a write to arrive at the server first and thus gets the old value).

In practice, however, the performance of a distributed system in which all file requests must go to a single server is frequently poor. This problem is often solved by allowing clients to maintain local copies of heavily used files in their private (local) caches. Although we will discuss the details of file caching below, for the moment it is sufficient to point out that if a client locally modifies a cached file and shortly thereafter another client reads the file from the server, the second client will get an obsolete file, as illustrated in Fig. 10-10(b).

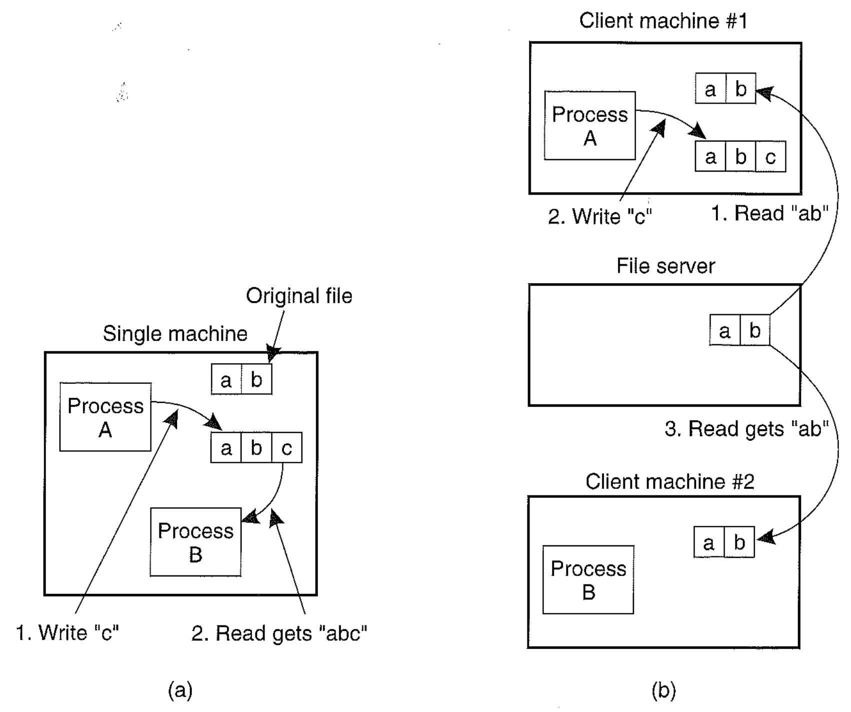


Figure 10-10. (a) On a single processor, when a read follows a write, the value returned by the read is the value just written. (b) In a distributed system with caching, obsolete values may be returned.

One way out of this difficulty is to propagate all changes to cached files back to the server immediately. Although conceptually simple, this approach is inefficient. An alternative solution is to relax the semantics of file sharing. Instead of requiring a read to see the effects of all previous writes, one can have a new rule that says: "Changes to an open file are initially visible only to the process (or possibly machine) that modified the file. Only when the file is closed are the changes made visible to other processes (or machines)." The adoption of such a rule does not change what happens in Fig. 10-10(b), but it does redefine the actual behavior (*B* getting the original value of the file) as being the correct one. When *A* closes the file, it sends a copy to the server, so that subsequent reads get the new value, as required. This rule is widely implemented and is known as **session semantics**. Like most distributed file systems, NFS implements session semantics. This means that although NFS in theory follows the remote access model of Fig. 10-1(a), most implementations make use of local caches, effectively implementing the upload/download model of Fig. 10-1(b).

Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file. One solution is to say that as each file is closed in turn, its value is sent back to the server, so the final result depends on whose close request is most recently processed by the server. A less pleasant, but easier to implement alternative is to say that the final result is one of the candidates, but leave the choice of which one unspecified.

A completely different approach to the semantics of file sharing in a distributed system is to make all files immutable. There is thus no way to open a file for writing. In effect, the only operations on files are create and read.

What is possible is to create an entirely new file and enter it into the directory system under the name of a previous existing file, which now becomes inaccessible (at least under that name). Thus although it becomes impossible to modify the file x, it remains possible to replace x by a new file atomically. In other words, although *files* cannot be updated, *directories* can be. Once we have decided that files cannot be changed at all, the problem of how to deal with two processes, one of which is writing on a file and the other of which is reading it, just disappears.

What does remain is the problem of what happens when two processes try to replace the same file at the same time. As with session semantics, the best solution here seems to be to allow one of the new files to replace the old one, either the last one or nondeterministically.

A somewhat stickier problem is what to do if a file is replaced while another process is busy reading it. One solution is to somehow arrange for the reader to continue using the old file, even if it is no longer in any directory, analogous to the way UNIX allows a process that has a file open to continue using it, even after it has been deleted from all directories. Another solution is to detect that the file has changed and make subsequent attempts to read from it fail.

A fourth way to deal with shared files in a distributed system is to use transactions, as we discussed in detail in Chap. 5. To summarize briefly, to access a file or a group of files, a process first executes some type of BEGIN_TRANSACTION primitive to signal that what follows must be executed indivisibly. Then come system calls to read and write one or more files. When the work has been completed, an END_TRANSACTION primitive is executed. The key property of this method is that the system guarantees that all the calls contained within the transaction will be carried out in order, without any interference from other, concurrent transactions. If two or more transactions start up at the same time, the system ensures that the final result is the same as if they were all run in some (undefined) sequential order.

In Fig. 10-11 we summarize the four approaches we have discussed for dealing with shared files in a distributed system.

File Locking in NFS

For a distributed file system in which servers can be stateless, locking files can be problematic. In NFS, file locking has traditionally been handled by means of a separate protocol, implemented by a (stateful) lock manager. However, for

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transactions	All changes occur atomically

Figure 10-11. Four ways of dealing with the shared files in a distributed system.

various reasons, file locking using the NFS locking protocol has never been very popular due to the intricacies of the protocol, and the resulting poorly performing, or even faulty implementations (Shepler, 1999). With version 4, file locking is integrated into the NFS file access protocol. It is expected that this approach will make it simpler for clients to apply file locking instead of resorting to ad-hoc solutions that were used instead of NFS version 3 locking.

File locking in a distributed file system is complicated by the fact that clients and servers may fail while locks are still held. Proper recovery then becomes important to ensure consistency of shared files. We return to these issues below when discussing fault tolerance in NFS.

Conceptually, file locking in NFS version 4 is simple. There are essentially only four operations related to locking, as shown in Fig. 10-12. NFS distinguishes read locks from write locks. Multiple clients can simultaneously access the same part of a file provided they only read data. A write lock is needed to obtain exclusive access to modify part of a file.

Operation	Description
Lock	Create a lock for a range of bytes
Lockt	Test whether a conflicting lock has been granted
Locku	Remove a lock from a range of bytes
Renew	Renew the lease on a specified lock

Figure 10-12. NFS version 4 operations related to file locking.

Operation lock is used to request a read or write lock on a consecutive range of bytes in a file. It is a nonblocking operation; if the lock cannot be granted due to another conflicting lock, the client gets back an error message and has to poll the server at a later time. Alternatively, the client can request to be put on a FIFO-ordered list maintained by the server. As soon as the conflicting lock has been removed, the server will grant the next lock to the client at the top of the list, provided it polls the server before a certain time expires. This approach prevents the server from having to notify clients, while still being fair to clients whose lock request could not be granted because grants are made in FIFO order.

The lockt operation is used to test whether a conflicting lock exists. For example, a client can test whether there are any read locks granted on a specific range of bytes in a file, before requesting a write lock for those bytes. In the case of a conflict, the requesting client is informed exactly who is causing the conflict and on which range of bytes. It can be implemented more efficiently than lock, because there is no need to attempt to open a file.

Removing a lock from a file is done by means of the locku operation.

Locks are granted for a specific time (determined by the server). In other words, they have an associated lease. Unless a client renews the lease on its granted lock, the server will automatically remove it. As we shall see, this approach is followed for other server-provided resources as well and helps in recovery after failures. Using the renew operation, a client requests the server to renew the lease on its lock (and, in fact, other resources as well).

In addition to these operations, there is also an implicit way to lock a file, referred to as **share reservation**. Share reservation is completely independent from locking, and can be used to implement NFS for Windows-based systems. When a client opens a file, it specifies the type of access it requires (namely *READ*, *WRITE*, or *BOTH*), and which type of access the server should deny other clients (*NONE*, *READ*, *WRITE*, or *BOTH*). If the server cannot meet the client's requirements, the open operation will fail for that client. In Fig. 10-13 we show exactly what happens when a new client opens a file that has already been successfully opened by another client. For an already opened file, we distinguish two different state variables. The access state specifies how the file is currently being accessed by the current client. The denial state specifies what accesses by new clients are not permitted.

In Fig. 10-13(a), we show what happens when a client tries to open a file requesting a specific type of access, given the current denial state of that file. Likewise, Fig. 10-13(b) shows the result of opening a file that is currently being accessed by another client, but now requesting certain access types to be disallowed.

10.1.6 Caching and Replication

Like many distributed file systems, NFS makes extensive use of client caching to enhance performance. In addition, it also provides minor support for file replication.

Client Caching

Caching in NFS version 3 has been mainly left outside of the protocol. This approach has led to the implementation of different caching policies, of which most never guaranteed consistency. At best, cached data could be stale for a few seconds compared to the data stored at a server. However, implementations also exist that allowed cached data to be stale for 30 seconds without the client knowing. This state of affairs is less than desirable.

		Current file denial state			
		NONE	READ	WRITE	BOTH
Request access	READ	Succeed	Fail	Succeed	Fail
	WRITE	Succeed	Succeed	Fail	Fail
	BOTH	Succeed	Fail	Fail	Fail

(a)

		Requested file denial state			
		NONE	READ	WRITE	BOTH
Current access state	READ	Succeed	Fail	Succeed	Fail
	WRITE	Succeed	Succeed	Fail	Fail
	BOTH	Succeed	Fail	Fail	Fail

(b)

Figure 10-13. The result of an open operation with share reservations in NFS. (a) When the client requests shared access given the current denial state. (b) When the client requests a denial state given the current file access state.

NFS version 4 solves some of these consistency problems, but essentially still leaves cache consistency to be handled in an implementation-dependent way. The general caching model that is assumed by NFS is shown in Fig. 10-14. Each client can have a memory cache that contains data previously read from the server. In addition, there may also be a disk cache that is added as an extension to the memory cache, using the same consistency parameters.

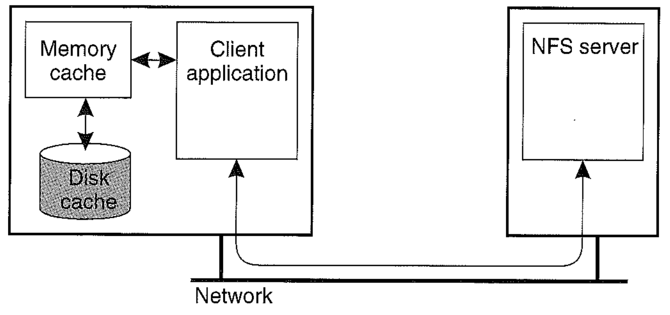


Figure 10-14. Client-side caching in NFS.

Typically, clients cache file data, attributes, file handles, and directories. Different strategies exist to handle consistency of the cached data, cached attributes, and so on. Let us first take a look at caching file data.

NFS version 4 supports two approaches for caching file data. The simplest approach is when a client opens a file and caches the data it obtains from the server as the result of various read operations. In addition, write operations can be carried out in the cache as well. When the client closes the file, NFS requires that if modifications have taken place, the cached data must be flushed back to the server. This approach corresponds to implementing session semantics as discussed above.

Once (part of) a file has been cached, a client can keep its data in the cache even after closing the file. Also, several clients on the same machine can share a single cache. NFS requires that whenever a client opens a previously closed file that has been (partly) cached, the client must immediately revalidate the cached data. Revalidation takes place by checking when the file was last modified and invalidating the cache in case it contains stale data.

New to the NFS specifications is that a server may delegate some of its rights to a client when a file is opened. **Open delegation** takes place when the client machine is allowed to locally handle open and close operations from other clients on the same machine. Normally, the server is in charge of checking whether opening a file should succeed or not, for example, because share reservations need to be taken into account. With open delegation, the client machine is sometimes allowed to make such decisions, avoiding the need to contact the server.

For example, if a server has delegated the opening of a file to a client that requested write permissions, file locking requests from other clients on the same machine can also be handled locally. The server will still handle locking requests from clients on other machines, by simply denying those clients access to the file. Note that this scheme does not work in the case of delegating a file to a client that requested only read permissions. In that case, whenever another local client wants to have write permissions, it will have to contact the server; it is not possible to handle the request locally.

An important consequence of delegating a file to a client, is that the server needs to be able to recall the delegation, for example, when another client on a different machine needs to obtain access rights to the file. Recalling a delegation requires that the server can do a callback to the client, as illustrated in Fig. 10-15.

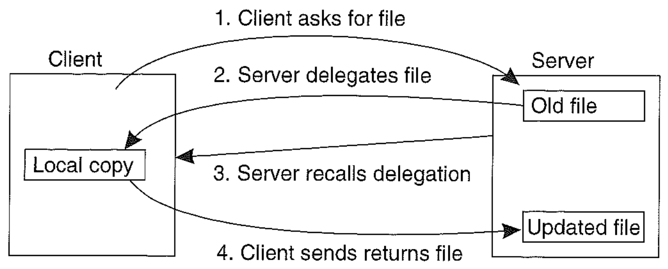


Figure 10-15. Using the NFS version 4 callback mechanism to recall file delegation.

A callback is implemented in NFS using its underlying RPC mechanisms. Note, however, that callbacks require that the server keeps track of clients to which it has delegated a file. Here, we see another example where an NFS server can no longer be implemented in a stateless manner. As we discuss below, the combination of delegation and stateful servers may lead to various problems in the presence of client and server failures.

Clients can also cache attribute values, but are largely left on their own when it comes to keeping cached values consistent. In particular, attribute values of the same file cached by two different clients may be different unless the clients keep these attributes mutually consistent. Modifications to an attribute value should be immediately forwarded to the server, thus following a write-through cache coherence policy.

A similar approach is followed for caching file handles (or rather, the name-to-file handle mapping) and directories. To mitigate the effects of inconsistencies, NFS uses leases on cached attributes, file handles, and directories. After some time has elapsed cache entries are thus automatically invalidated and revalidation is needed before they are used again.

Replica Servers

NFS version 4 provides minimal support for file replication. Only whole file systems can be replicated (i.e., the logical block device consisting of files, attributes, directories, and data blocks). Support is provided in the form of the *FS_LOCATIONS* attribute that is recommended for each file. This attribute provides a list of locations where the file system in which the associated file is contained, may possibly occur. Each location is given as a DNS name or in the form of an IP address. Note that it is up to a specific NFS implementation to actually provide replicated servers. NFS version 4 does not dictate how replication should take place.

10.1.7 Fault Tolerance

Until the most recent version of NFS, fault tolerance in NFS has hardly been an issue. The reason for this is that the NFS protocol did not require servers to be stateful. As a consequence, recovering from a server crash was rather simple, as no state was ever lost. Of course, state was maintained by separate lock managers, and in such cases, special measures were needed.

By abandoning the stateless design, fault tolerance and recovery need to be dealt with in NFS version 4. In particular, statefulness occurs in two occasions: file locking and delegation. In addition, special measures need to be taken to handle the unreliability of the RPC mechanism underlying the NFS protocol. Let us take a look at these issues separately, starting with RPC failures.

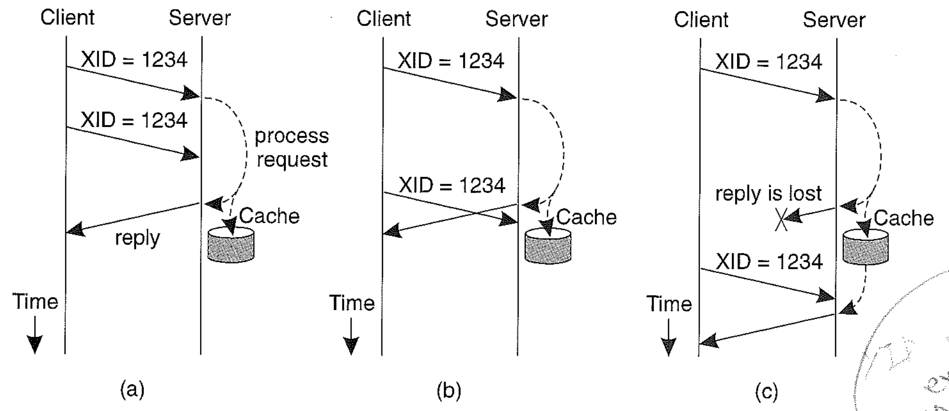
RPC Failures

The problem with the RPC mechanism as used by NFS is that it provides no guarantees with respect to reliability. In practice, the client and server RPC stubs can be generated to either rely on a reliable connection-oriented transport protocol such as TCP, or an unreliable connectionless transport protocol such as UDP.

The main problem that NFS has with the underlying RPC semantics is its lack of detecting duplicate requests. As a consequence, when an RPC *reply* is lost and the client successfully retransmits the original request, the server will eventually have carried out the request more than once. When dealing with nonidempotent operations, this situation should not occur.

These problems are alleviated by means of a **duplicate-request cache** that is implemented by the server (Juszczak, 1990). Each RPC request from a client carries a unique **transaction identifier** (**XID**) in its header, which is cached by the server when the request comes in. As long as the server has not sent a reply, it will indicate that the RPC request is in progress. When the request has been handled, its associated reply is also cached, after which the reply is returned to the client.

There are now three situations that need to be dealt with. In the first case, shown in Fig. 10-16(a), the client sends a request and starts a timer. If the timer goes off before the reply comes back, the client retransmits the original request with the same XID as the original one. The server has not yet completed the original request, and will therefore just ignore the retransmitted request.



XID 1235 erases XID 1234

Figure 10-16. Three situations for handling retransmissions. (a) The request is still in progress. (b) The reply has just been returned. (c) The reply has been sent some time ago, but was lost.

Second, the server may receive a retransmitted request just after it had sent the reply to the client. If the time of arrival for the retransmitted request is close to the time the server sent the reply, the server concludes that the retransmission and

the reply have crossed each other, so, again, it ignores the retransmitted request. This situation is shown in Fig. 10-16(b).

In the third case, the reply actually did get lost and the retransmitted request should be responded to by sending the cached results of the operation to the client, as shown in Fig. 10-16(c). Note that the results of the file operation should remain cached, for there is no guarantee that the retransmission will succeed this time. How such and related problems can be solved is explored in the exercises.

File Locking in the Presence of Failures

File locking in NFS version 3 was handled through a separate server. In this way, the stateless nature of the NFS protocol could be maintained and problems related to fault tolerance were to be handled by the lock server. However, with the introduction of file locks in NFS version 4, it became inevitable to provide a basic mechanism to handle client and server crashes as part of the NFS protocol. The problem to be solved is relatively simple. To lock a file, a client issues a lock request at the server. Assuming the lock is granted, we may now run into trouble when the client or server crashes.

To solve client crashes, the server issues a lease on every lock it grants. When the lease expires, the server will remove the lock, thereby releasing the associated resources (i.e., files). To prevent the server from removing a lock, the client should renew its lease before it expires. For this purpose, NFS provides the renew operation explained before.

There are situations in which locks are removed even if the client does not crash. Notably, false removal may happen if the client cannot reach the server to renew a lease, for example, because the network is (temporarily) partitioned. No special measures are taken to handle such situations.

When the server crashes and subsequently recovers, it will presumably have lost information on locks it granted to clients. The approach followed in NFS version 4, is to enter a **grace period** in which a client can reclaim locks that were previously granted to it. In this way, the server builds up its previous state with respect to locks. Note that this lock recovery is independent of the recovery of other lost data. During the grace period, only requests for reclaiming locks are generally accepted; normal lock requests may be refused until the grace period is over.

At this point, note that using leases introduces numerous problems that are only partly solved in NFS. For example, leasing requires that the client and the server have their clocks synchronized. If the server states that a lease expires at time *T*, the client will need to have the same notion of what the actual time is as the server. On the other hand, if the server states that a lease expires after some duration *D*, the time it takes to send the lease to the client will have to be known. In wide-area systems, these timing issues may not be easily solved.

Another problem concerns the reliability of sending a lease renewal message to the server. If message delivery fails or is delayed, the lease may unintentionally expire. In either case, the server will have to explicitly issue a new lease to the client before the latter can continue to use a file.

Open Delegation in the Presence of Failures

Open delegation introduces additional problems when a client or server crashes. First, consider a client to which the opening of a file has been delegated. If that client crashes, it will presumably have been working on its local copy without immediately propagating updates to the server. In other words, the client follows a write-back cache coherence policy. In that case, unless the client's updates are locally saved to stable storage, full recovery of the file will be impossible. In any case, the client is made partially responsible for file recovery.

When the server crashes and subsequently recovers, it follows a procedure similar to lock recovery. A client to which the opening of a file has been delegated will reclaim that delegation when the server comes up again. However, in contrast to locking, the server forces the client to flush all modifications back to the server, effectively recalling the delegation.

This approach has two effects. First, the server is now up-to-date with respect to the most recent modifications of each file it had delegated to a client. Second, the server is again in full charge of the file, and it may decide to delegate the file to another client.

10.1.8 Security

As we mentioned before, the basic idea behind NFS is that a remote file system should be presented to clients as if it were a local file system. In this light, it should come as no surprise that security in NFS mainly focuses on the communication between a client and a server. Secure communication means that a secure channel between the two should be set up as we discussed in Chap. 8.

In addition to secure RPCs, it is necessary to control accesses to files, which are handled by means of access control file attributes in NFS. A file server is in charge of verifying the access rights of its clients, as we will explain below. Combined with secure RPCs, the NFS security architecture then looks as is shown in Fig. 10-17.

Secure RPCs

Because NFS is layered on top of an RPC system, setting up a secure channel in NFS boils down to establishing secure RPCs. Up until NFS version 4, a secure RPC meant that only authentication was taken care of. There were three ways for doing authentication. We will now examine each one in turn.

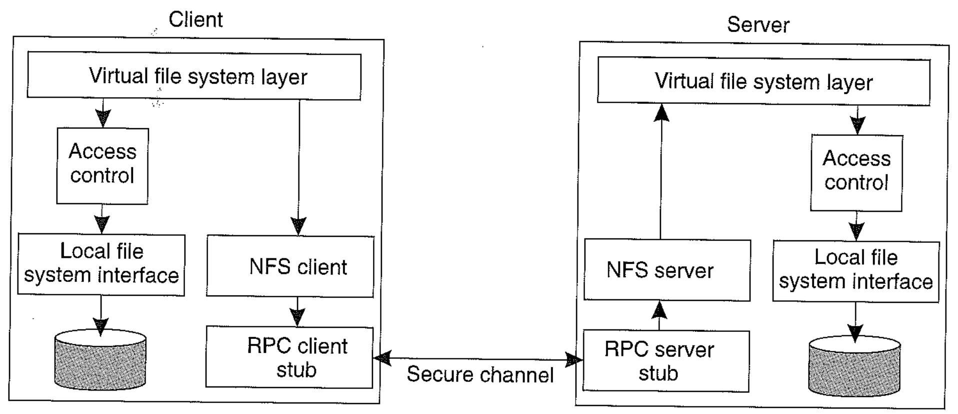

Client
Virtual file system layer
Access control
Local file system interface
NFS client
RPC client stub
Secure channel
Server
Virtual file system layer
Access control
NFS server
Local file system interface
RPC server stub

Figure 10-17. The NFS security architecture.

The most widely used method, one that actually hardly does any authentication, is known as system authentication. In this UNIX-based method, a client simply passes its effective user ID and group ID to the server, along with a list of groups it claims to be a member of. This information is sent to the server as unsigned plaintext. In other words, the server has no way of verifying whether the claimed user and group identifiers are actually associated with the sender. In essence, the server assumes that the client has passed a proper login procedure, and that it can trust the client's machine.

The second authentication method in older NFS versions uses Diffie-Hellman key exchange to establish a session key, leading to what is called **secure NFS**. We explained how Diffie-Hellman key exchange works in Chap. 8. This approach is much better than system authentication, but is more complex, for which reason it is less often implemented. Diffie-Hellman can be viewed as a public-key cryptosystem. Initially, there was no way to securely distribute a server's public key, but this was later corrected with the introduction of a secure name service. A point of criticism has always been the use of relatively small public keys, which are only 192 bits in NFS. It has been shown that breaking a Diffie-Hellman system with such short keys is nearly trivial (Lamacchia and Odlyzko, 1991).

The third authentication protocol is Kerberos (version 4), which we also described in Chap. 8.

With the introduction of NFS version 4, security is enhanced by the support for RPCSEC_GSS. **RPCSEC_GSS** is a general security framework that can support a myriad of security mechanism for setting up secure channels (Eisler et al., 1997). In particular, it not only provides the hooks for different authentication systems, but also supports message integrity and confidentiality, two features that were not supported in older versions of NFS.

RPCSEC_GSS is based on a standard interface for security services, namely **GSS-API**, which we briefly mentioned in Chap. 8 and which is fully described in

(Linn, 1997). The RPCSEC_GSS is layered on top of this interface, leading to the organization shown in Fig. 10-18.

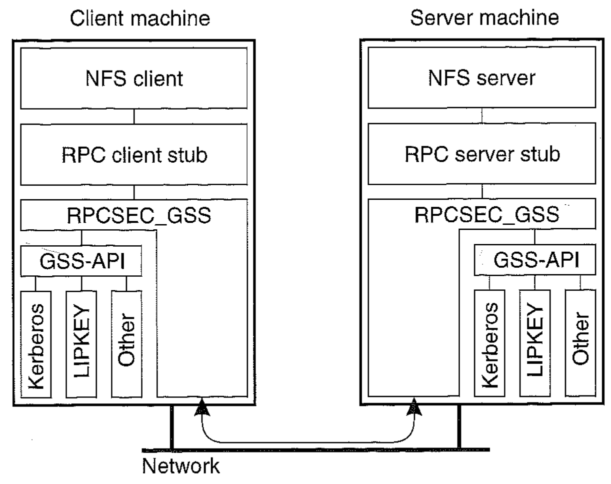


Figure 10-18. Secure RPC in NFS version 4.

For NFS version 4, RPCSEC_GSS should be configured with support for Kerberos version 5. In addition, the system must also support a method known as **LIPKEY**, described in (Eisler, 2000). LIPKEY is a public-key system that allows clients to be authenticated using a password while servers can be authenticated using a public key.

The important aspect of secure RPC in NFS, is that the designers have chosen not to provide their own security mechanisms, but only to provide a standard way for handling security. As a consequence, proven security mechanisms, such Kerberos, can be incorporated into an NFS implementation without affecting other parts of the system. Also, if an existing security mechanisms turns out to be flawed (such as in the case of Diffie-Hellman when using small keys), it can easily be replaced.

It should be noted that because RPCSEC_GSS is implemented as part of the RPC layer that underlies the NFS protocols, it can also be used for older versions of NFS. However, this adaptation to the RPC layer became available only with the introduction of NFS version 4.

Access Control

Authorization in NFS is analogous to secure RPC: it provides the mechanisms but does not specify any particular policy. Access control is supported by means of the *ACL* file attribute. This attribute is a list of access control entries, where each entry specifies the access rights for a specific user or group. The operations

that NFS distinguishes with respect to access control are relatively straightforward and are listed in Fig. 10-19.

Operation	Description
Read_data	Permission to read the data contained in a file
Write_data	Permission to modify a file's data
Append_data	Permission to append data to a file
Execute	Permission to execute a file
List_directory	Permission to list the contents of a directory
Add_file	Permission to add a new file to a directory
Add_subdirectory	Permission to create a subdirectory to a directory
Delete	Permission to delete a file
Delete_child	Permission to delete a file or directory within a directory
Read_acl	Permission to read the ACL
Write_acl	Permission to write the ACL
Read_attributes	The ability to read the other basic attributes of a file
Write_attributes	Permission to change the other basic attributes of a file
Read_named_attrs	Permission to read the named attributes of a file
Write_named_attrs	Permission to write the named attributes of a file
Write_owner	Permission to change the owner
Synchronize	Permission to access a file locally at the server with synchronous reads and writes

Figure 10-19. The classification of operations recognized by NFS with respect to access control.

Compared to the simple access control mechanisms in, for example, UNIX systems, NFS distinguishes many different kinds of operations. Noteworthy is also the synchronize operation that essentially tells whether a process that is colocated with a server can directly access a file, bypassing the NFS protocol for improved performance. The NFS model for access control has much richer semantics than most UNIX models. This difference comes from the requirements that NFS should be able to interoperate with Windows 2000 systems. The underlying thought is that it is much easier to fit the UNIX model of access control to that of Windows 2000, then the other way around.

Another aspect that makes access control different from file systems such as in UNIX, is that access can be specified for different users and different groups. Traditionally, access to a file is specified for a single user (the owner of the file), a single group of users (e.g., members of a project team), and for everyone else. NFS has many different kinds of users and processes, as shown in Fig. 10-20.

Type of user	Description
Owner	The owner of a file
Group	The group of users associated with a file
Everyone	Any user or process
Interactive	Any process accessing the file from an interactive terminal
Network	Any process accessing the file via the network
Dialup	Any process accessing the file through a dialup connection to the server
Batch	Any process accessing the file as part of batch job
Anonymous	Anyone accessing the file without authentication
Authenticated	Any authenticated user or process
Service	Any system-defined service process

Figure 10-20. The various kinds of users and processes distinguished by NFS with respect to access control.

10.2 THE CODA FILE SYSTEM

Our next example of a distributed file system is Coda. **Coda** has been developed at Carnegie Mellon University (CMU) in the 1990s, and is now integrated with a number of popular UNIX-based operating systems such as Linux. Coda is in many ways different from NFS, notably with respect to its goal for high availability. This goal has led to advanced caching schemes that allow a client to continue operation despite being disconnected from a server. Overviews of Coda are described in (Satyanarayanan et al., 1990; Kistler and Satyanarayanan, 1992). A detailed description of the system can be found in (Kistler, 1996).

10.2.1 Overview of Coda

Coda was designed to be a scalable, secure, and highly available distributed file system. An important goal was to achieve a high degree of naming and location transparency so that the system would appear to its users very similar to a pure local file system. By also taking high availability into account, the designers of Coda have also tried to reach a high degree of failure transparency.

Coda is a descendant of version 2 of the **Andrew File System** (**AFS**), which was also developed at CMU (Howard et al., 1988; Satyanarayanan, 1990), and inherits many of its architectural features from AFS. AFS was designed to support the entire CMU community, which implied that approximately 10,000 workstations would need to have access to the system. To meet this requirement, AFS nodes are partitioned into two groups. One group consists of a relatively small number of dedicated **Vice** file servers, which are centrally administered. The other

group consists of a very much larger collection of **Virtue** workstations that give users and processes access to the file system, as shown in Fig. 10-21.

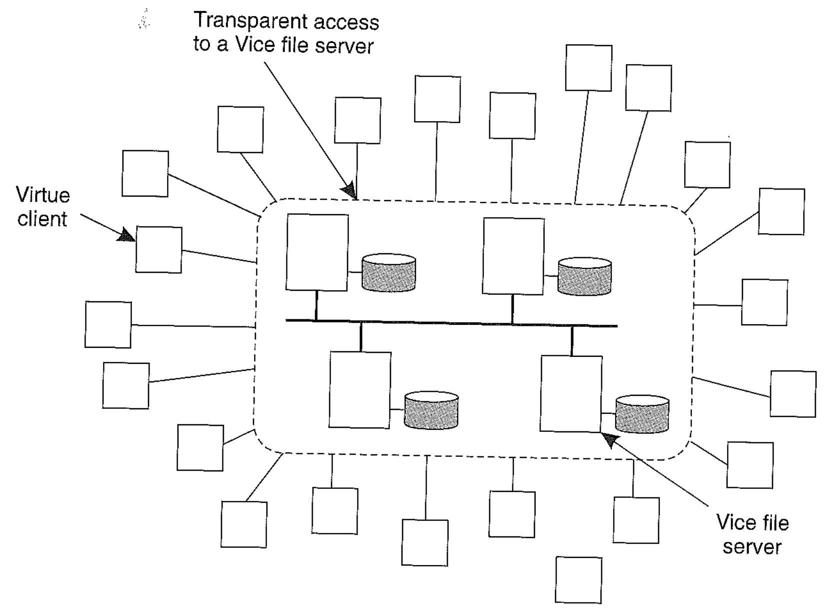


Figure 10-21. The overall organization of AFS.

Coda follows the same organization as AFS. Every Virtue workstation hosts a user-level process called **Venus**, whose role is similar to that of an NFS client. A Venus process is responsible for providing access to the files that are maintained by the Vice file servers. In Coda, Venus is also responsible for allowing the client to continue operation even if access to the file servers is (temporarily) impossible. This additional role is a major difference with the approach followed in NFS.

The internal architecture of a Virtue workstation is shown in Fig. 10-22. The important issue is that Venus runs as a user-level process. Again, there is a separate Virtual File System (VFS) layer that intercepts all calls from client applications, and forwards these calls either to the local file system or to Venus, as shown in Fig. 10-22. This organization with VFS is the same as in NFS. Venus, in turn, communicates with Vice file servers using a user-level RPC system. The RPC system is constructed on top of UDP datagrams and provides at-most-once semantics.

There are three different server-side processes. The great majority of the work is done by the actual Vice file servers, which are responsible for maintaining a local collection of files. Like Venus, a file server runs as a user-level process. In addition, trusted Vice machines are allowed to run an authentication server, which we discuss in detail later. Finally, update processes are used to keep meta-information on the file system consistent at each Vice server.