# NFS Backgrounder

## The Network Filesystem

Marshall Kirk McKusick, Keith Bostic, Michael Karels, and John Quarterman

Addison-Wesley Publishing Company, Inc. | 1996

TECHNICAL REPORT

Network Appliance, a pioneer and industry leader in data storage technology, helps organizations understand and meet complex technical challenges with advanced storage solutions and global data management strategies.

**Abstract**

This article is Chapter 9 'The Network Filesystem' reprinted from "The Design and Implementation of the 4.4BSD Operating System" by Marshall Kirk McKusick, Keith Bostic, Michael Karels, and John Quarterman. Addison-Wesley Publishing Company, Inc. (0-201-54979-4) Copyright 1996.

This chapter is divided into three main sections. The first gives a brief history of remote filesystems. The second describes the client and server halves of NFS and the mechanics of how they operate. The final section describes the techniques needed to provide reasonable performance for remote filesystems in general, and NFS in particular.

## 9.1 History and Overview

When networking first became widely available in 4.2BSD, users who wanted to share files all had to log in across the net to a central machine on which the shared files were located. These central machines quickly became far more loaded than the user's local machine, so demand quickly grew for a convenient way to share files on several machines at once. The most easily understood sharing model is one that allows a server machine to export its filesystems to one or more client machines. The clients can then import these filesystems and present them to the user as though they were just another local filesystem.

Numerous remote-filesystem protocol designs and protocols were proposed and implemented. The implementations were attempted at all levels of the kernel. Remote access at the top of the kernel resulted in semantics that nearly matched the local filesystem, but had terrible performance. Remote access at the bottom of the kernel resulted in awful semantics, but great performance. Modern systems place the remote access in the middle of the kernel at the vnode layer. This level gives reasonable performance and acceptable semantics.

An early remote filesystem, *UNIX United*, was implemented near the top of the kernel at the system-call dispatch level. It checked for file descriptors representing remote files and sent them off to the server. No caching was done on the client machine. The lack of caching resulted in slow performance, but in semantics nearly identical to a local filesystem. Because the current directory and executing files are referenced internally by vnodes rather than by descriptors, UNIX United did not allow users to change directory into a remote filesystem and could not execute files from a remote filesystem without first copying the files to a local filesystem.

At the opposite extreme was Sun Microsystem's *network disk*, implemented near the bottom of the kernel at the device-driver level. Here, the client's entire filesystem and buffering code was used. Just as in the local filesystem, recently read blocks from the disk were stored in the buffer cache. Only when a file access requested a block that was not already in the cache would the client send a request for the needed physical disk block to the server. The performance was excellent because the buffer cache serviced most of the file-access requests just as it does for the local filesystem. Unfortunately, the semantics suffered because of incoherency between the client and server caches. Changes made on the server would not be seen by the client, and vice versa. As a result, the network disk could be used only by a single client or as a read-only filesystem.

The first remote filesystem shipped with System V was *RFS* [Rifkin et al, 1986]. Although it had excellent UNIX semantics, its performance was poor, so it met with little use. Research at Carnegie-Mellon lead to the *Andrew filesystem* [Howard, 1988]. The Andrew filesystem was commercialized by Transarc and eventually became part of the Distributed Computing Environment promulgated by the Open Software Foundation, and was supported by many vendors. It is designed to handle widely distributed servers and clients and also to work well with mobile computers that operate while detached from the network for long periods.

The most commercially successful and widely available remote-filesystem protocol is the *network filesystem (NFS)* designed and implemented by Sun Microsystems [Walsh et al, 1985; Sandberg et al, 1985]. There are two important components to the success of NFS. First, Sun placed the protocol specification for NFS in the public domain. Second, Sun sells that implementation to all people who want it, for less than the cost of implementing it themselves. Thus, most vendors chose to buy the Sun implementation. They are willing to buy from Sun because they know that they can always legally write their own implementation if the price of the Sun implementation is raised to an unreasonable level. The 4.4BSD implementation was written from the protocol

specification, rather than being incorporated from Sun, because of the developers desire to be able to redistribute it freely in source form.

NFS was designed as a client-server application. Its implementation is divided into a client part that imports filesystems from other machines and a server part that exports local filesystems to other machines. The general model is shown in Fig. 9.1. Many goals went into the NFS design:

- The protocol is designed to be stateless. Because there is no state to maintain or recover, NFS can continue to operate even during periods of client or server failures. Thus, it is much more robust than a system that operates with state.
- NFS is designed to support UNIX filesystem semantics. However, its design also allows it to support the possibly less rich semantics of other filesystem types, such as MS-DOS.
- The protection and access controls follow the UNIX semantics of having the process present a UID and set of groups that are checked against the file's owner, group, and other access modes. The security check is done by filesystem-dependent code that can do more or fewer checks based on the capabilities of the filesystem that it is supporting. For example, the MS-DOS filesystem cannot implement the full UNIX security validation and makes access decisions solely based on the UID.
- The protocol design is transport independent. Although it was originally built using the UDP datagram protocol, it was easily moved to the TCP stream protocol. It has also been ported to run over numerous other non IP-based protocols.

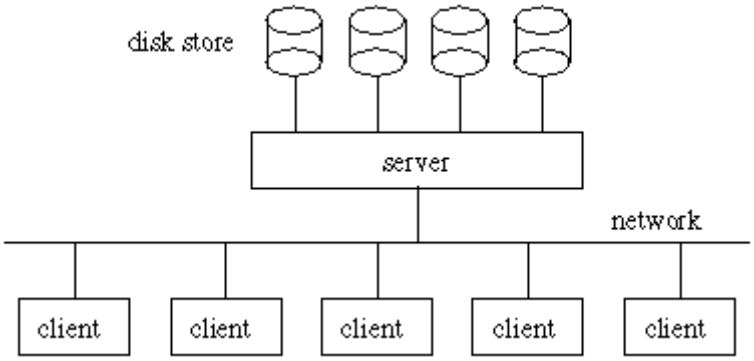


**Figure 9.1** The division of NFS between client and server

Some of the design decisions limit the set of applications for which NFS is appropriate:

- The design envisions clients and servers being connected on a locally fast network. The NFS protocol does not work well over slow links or between clients and servers with intervening gateways. It also works poorly for mobile computing that has extended periods of disconnected operation.
- The caching model assumes that most files will not be shared. Performance suffers when files are heavily shared.
- The stateless protocol requires some loss of traditional UNIX semantics. Filesystem locking (*flock*) has to be implemented by a separate stateful daemon. Deferral of the release of space in an unlinked file until the final process has closed the file is approximated with a heuristic that sometimes fails.

Despite these limitations, NFS proliferated because it makes a reasonable tradeoff between semantics and performance; its low cost of adoption has now made it ubiquitous.

## 9.2 NFS Structure and Operation

NFS operates as a typical client server application. The server receives *remote-procedure-call* (RPC) requests from its various clients. An RPC operates much like a local procedure call: The client makes a procedure call, then waits for the result while the procedure executes. For a remote procedure call, the parameters must be *marshaled* together into a message. Marshaling includes replacing pointers by the data to which they point and converting binary data to the canonical network byte order. The message is then sent to the server, where it is unmarshalled (separated out into its original pieces) and processed as a local filesystem operation. The result must be similarly marshalled and sent back to the client. The client splits up the result and returns that result to the calling process as though the result were being returned from a local procedure call [Birrell & Nelson, 1984]. The NFS protocol uses the Sun's RPC and external data-representation (XDR) protocols [Reid, 1987]. Although the kernel implementation is done by hand to get maximum performance, the user-level daemons described later in this section use Sun's public-domain RPC and XDR libraries.

The NFS protocol can run over any available stream- or datagram-oriented protocol. Common choices are the TCP stream protocol and the UDP datagram protocol. Each NFS RPC message may need to be broken into multiple packets to be sent across the network. A big performance problem for NFS running under UDP on an Ethernet is that the message may be broken into up to six packets; if any of these packets are lost, the entire message is lost and must be resent. When running under TCP on an Ethernet, the message may also be broken into up to six packets; however, individual lost packets, rather than the entire message, can be retransmitted. Section 9.3 discusses performance issues in greater detail.

The set of RPC requests that a client can send to a server is shown in Table 9.1. After the server handles each request, it responds with the appropriate data, or with an error code explaining why the request could not be done. As noted in the table, most operations are *idempotent.* An idempotent operation is one that can be repeated several times without the final result being changed or an error being caused. For example, writing the same data to the same offset in a file is idempotent because it will yield the same result whether it is done once or many times. However, trying to remove the same file more than once is nonidempotent because the file will no longer exist after the first try. Idempotency is an issue when the server is slow, or when an RPC acknowledgment is lost and the client retransmits the RPC request. The retransmitted RPC will cause the server to try to do the same operation twice. For a nonidempotent request, such as a request to remove a file, the retransmitted RPC, if undetected by the server recent-request cache [Juszczak, 1989], will cause a ``no such file'' error to be returned, because the file will have been removed already by the first RPC. The user may be confused by the error, because they will have successfully found and removed the file.

| **RPC request** | Action | Idempotent |
|---|---|---|
| GETATTR | get file attributes | yes |
| SETATTR | set file attributes | yes |
| LOOKUP | look up file name | yes |
| READLINK | read from symbolic link | yes |

| | | |
|---|---|---|
| READ | read from file | yes |
| WRITE | write to file | yes |
| CREATE | create file | yes |
| REMOVE | remove file | no |
| RENAME | rename file | no |
| LINK | create link to file | no |
| SYMLINK | create symbolic link | yes |
| MKDIR | create directory | no |
| RMDIR | remove directory | no |
| READDIR | read from directory | yes |
| STATFS | get filesystem attributes | yes |

**Table 9.1** NFS, Version 2, RPC requests

Each file on the server can be identified by a unique *file handle*. A file handle is the token by which clients refer to files on a server. Handles are globally unique and are passed in operations, such as read and write, that reference a file. A file handle is created by the server when a pathname-translation request (lookup) is sent from a client to the server. The server must find the requested file or directory and ensure that the requesting user has access permission. If permission is granted, the server returns a file handle for the requested file to the client. The file handle identifies the file in future access requests by the client. Servers are free to build file handles from whatever information they find convenient. In the 4.4BSD NFS implementation, the file handle is built from a filesystem identifier, an inode number, and a *generation number*. The server creates a unique filesystem identifier for each of its locally mounted filesystems. A generation number is assigned to an inode each time that the latter is allocated to represent a new file. Each generation number is used only once. Most NFS implementations use a random-number generator to select a new generation number; the 4.4BSD implementation selects a generation number that is approximately equal to the creation time of the file. The purpose of the file handle is to provide the server with enough information to find the file in future requests. The filesystem identifier and inode provide a unique identifier for the inode to be accessed. The generation number verifies that the inode still references the same file that it referenced when the file was first accessed. The generation number detects when a file has been deleted, and a new file is later created using the same inode. Although the new file has the same filesystem identifier and inode number, it is a completely different file from the one that the previous file handle referenced. Since the generation number is included in the file handle, the generation number in a file handle for a previous use of the inode will not match the new generation number in the same inode. When an old-generation file handle is presented to the server by a client, the server refuses to accept it, and instead returns the ``stale file handle'' error message.

The use of the generation number ensures that the file handle is *time stable*. Distributed systems define a time-stable identifier as one that refers uniquely to some entity both while that entity exists and for a long time after it is deleted. A time-stable identifier allows a system to remember an identity across transient failures and allows the system to detect and report errors for attempts to access deleted entities.

## The NFS Protocol

The NFS protocol is *stateless.* Being stateless means that the server does not need to maintain any information about which clients it is serving or about the files that they currently have open. Every RPC request that is received by the server is completely self-contained. The server does not need any additional information beyond that contained in the RPC to fulfill the request. For example, a read request will include the credential of the user doing the request, the file handle on which the read is to be done, the offset in the file to begin the read, and the number of bytes to be read. This information allows the server to open the file, verifying that the user has permission to read it, to seek to the appropriate point, to read the desired contents, and to close the file. In practice, the server caches recently accessed file data. However, if there is enough activity to push the file out of the cache, the file handle provides the server with enough information to reopen the file.

In addition to reducing the work needed to service incoming requests, the server cache also detects retries of previously serviced requests. Occasionally, a UDP client will send a request that is processed by the server, but the acknowledgment returned by the server to the client is lost. Receiving no answer, the client will time-out and resend the request. The server will use its cache to recognize that the retransmitted request has already been serviced. Thus, the server will not repeat the operation, but will just resend the acknowledgment. To detect such retransmissions properly, the server cache needs to be large enough to keep track of at least the most recent few seconds of NFS requests.

The benefit of the stateless protocol is that there is no need to do state recovery after a client or server has crashed and rebooted, or after the network has been partitioned and reconnected. Because each RPC is self-contained, the server can simply begin servicing requests as soon as it begins running; it does not need to know which files its clients have open. Indeed, it does not even need to know which clients are currently using it as a server.

There are drawbacks to the stateless protocol. First, the semantics of the local filesystem imply state. When files are unlinked, they continue to be accessible until the last reference to them is closed. Because NFS knows neither which files are open on clients nor when those files are closed, it cannot properly know when to free file space. As a result, it always frees the space at the time of the unlink of the last name to the file. Clients that want to preserve the freeing-on-last-close semantics convert unlink's of open files to renames to obscure names on the server. The names are of the form *.nfsAxxxx4.4,* where the *xxxx* is replaced with the hexadecimal value of the process identifier, and the A is successively incremented until an unused name is found. When the last close is done on the client, the client sends an unlink of the obscure filename to the server. This heuristic works for file access on only a single client; if one client has the file open and another client removes the file, the file will still disappear from the first client at the time of the remove. Other stateful semantics include the advisory locking described in Section 7.5. The locking semantics cannot be handled by the NFS protocol. On most systems, they are handled by a separate lock manager; the 4.4BSD version of NFS does not implement them at all.

The second drawback of the stateless protocol is related to performance. For version 2 of the NFS protocol, all operations that modify the filesystem must be committed to stable-storage before the RPC can be acknowledged. Most servers do not have battery-backed memory; the stable store requirement means that all written data must be on the disk before they can reply to the RPC. For a growing file, an update may require up to three synchronous disk writes: one for the inode to update its size, one for the indirect block to add a new data pointer, and one for the new data themselves. Each synchronous write takes several milliseconds; this delay severely restricts the write throughput for any given client file.

Version 3 of the NFS protocol eliminates some of the synchronous writes by adding a new asynchronous write RPC request. When such a request is received by the server, it is permitted to acknowledge the RPC without writing the new data to stable storage. Typically, a client will do a series of asynchronous write requests followed by a commit RPC request when it reaches the end of the file or it runs out of buffer space to store the file. The commit RPC request causes the server to write any unwritten parts of the file to stable store before acknowledging the commit RPC. The server benefits by having to write the inode and indirect blocks for the file only once per batch of asynchronous writes, instead of on every write RPC request. The client benefits from having higher throughput for file writes. The client does have the added overhead of having to save copies of all asynchronously written buffers until a commit RPC is done, because the server may crash before having written one or more of the asynchronous buffers to stable store. When the client sends the commit RPC, the acknowledgment to that RPC tells which of the asynchronous blocks were written to stable store. If any of the asynchronous writes done by the client are missing, the client knows that the server has crashed during the asynchronous-writing period, and resends the unacknowledged blocks. Once all the asynchronously written blocks have been acknowledged, they can be dropped from the client cache.

The NFS protocol does not specify the granularity of the buffering that should be used when files are written. Most implementations of NFS buffer files in 8-Kbyte blocks. Thus, if an application writes 10 bytes in the middle of a block, the client reads the entire block from the server, modifies the requested 10 bytes, and then writes the entire block back to the server. The 4.4BSD implementation also uses 8-Kbyte buffers, but it keeps additional information that describes which bytes in the buffer are modified. If an application writes 10 bytes in the middle of a block, the client reads the entire block from the server, modifies the requested 10 bytes, but then writes back only the 10 modified bytes to the server. The block read is necessary to ensure that, if the application later reads back other unmodified parts of the block, it will get valid data. Writing back only the modified data has two benefits:

1. Fewer data are sent over the network, reducing contention for a scarce resource.

2. Non-overlapping modifications to a file are not lost. If two different clients simultaneously modify different parts of the same file block, both modifications will show up in the file, since only the modified parts are sent to the server. When clients send back entire blocks to the server, changes made by the first client will be overwritten by data read before the first modification was made, and then will be written back by the second client.

## The 4.4BSD NFS Implementation

The NFS implementation that appears in 4.4BSD was written by Rick Macklem at the University of Guelph using the specifications of the Version 2 protocol published by Sun Microsystems [Sun Microsystems, 1989; Macklem, 1991]. This NFS Version 2 implementation had several 4.4BSD-only extensions added to it; the extended version became known as the *Not Quite NFS (NQNFS)* protocol [Macklem, 1994a]. This protocol provides

- Sixty-four bit file offsets and sizes
- An access RPC that provides server permission checking on file open, rather than having the client guess whether the server will allow access
- An append option on the write RPC
- Extended file attributes to support 4.4BSD filesystem functionality more fully

- A variant of short-term *leases* with delayed-write client caching that give distributed cache consistency and improved performance [Gray & Cheriton, 1989]

Many of the NQNFS extensions were incorporated into the revised NFS Version 3 specification [Sun Microsystems, 1993; Pawlowski et al, 1994]. Others, such as leases, are still available only with NQNFS. The NFS implementation distributed in 4.4BSD supports clients and servers running the NFS Version 2, NFS Version 3, or NQNFS protocol [Macklem, 1994b]. The NQNFS protocol is described in Section 9.3.

The 4.4BSD client and server implementations of NFS are kernel resident. NFS interfaces to the network with sockets using the kernel interface available through *sosend*() and *soreceive*() (see Chapter 11 for a discussion of the socket interface). There are connection-management routines for support of sockets using connection-oriented protocols; there are timeout and retransmit support for datagram sockets on the client side.

The less time-critical operations, such as mounting and unmounting, as well as determination of which filesystems may be exported and to what set of clients they may be exported are managed by user-level system daemons. For the server side to function, the portmap, mountd, and **nfsd** daemons must be running. The **portmap** daemon acts as a registration service for programs that provide RPC-based services. When an RPC daemon is started, it tells the **portmap** daemon to what port number it is listening and what RPC services it is prepared to serve. When a client wishes to make an RPC call to a given service, it will first contact the **portmap** daemon on the server machine to determine the port number to which RPC messages should be sent.

The interactions between the client and server daemons when a remote filesystem is mounted are shown in Fig. 9.2. The **mountd** daemon handles two important functions:

1. On startup and after a hang-up signal, **mountd** reads the **/etc/exports** file and creates a list of hosts and networks to which each local filesystem may be exported. It passes this list into the kernel using the *mount* system call; the kernel links the list to the associated local filesystem mount structure so that the list is readily available for consultation when an NFS request is received.

2. Client mount requests are directed to the **mountd** daemon. After verifying that the client has permission to mount the requested filesystem, **mountd** returns a file handle for the requested mount point. This file handle is used by the client for later traversal into the filesystem.

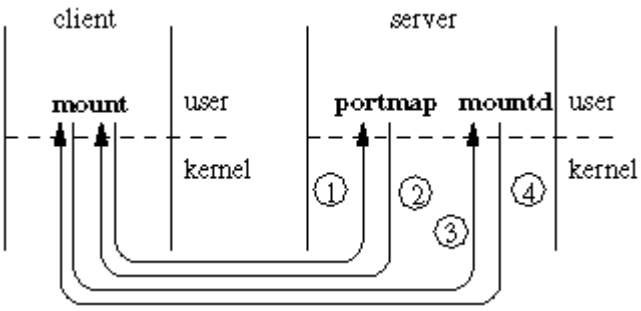


**Figure 9.2** Daemon interaction when a remote filesystem is mounted. Step 1: The client's mount process sends a message to the well-known port of the server's **portmap** daemon, requesting the port address of the server's **mountd** daemon. Step 2: The server's **portmap** daemon returns the port address of its server's **mountd** daemon. Step 3: The client's **mount** process sends a request to the server's **mountd** daemon with the pathname of the filesystem that it wants to mount. Step 4: The server's mountd daemon requests a file handle for the desired mount point from its kernel. If the request is successful, the file handle is returned to the client's **mount** process. Otherwise, the error from the file-handle request is returned. If the request is successful, the client's **mount** process does a *mount* system call, passing in the file handle that it received from the server's **mountd** daemon.

The **nfsd** master daemon forks off children that enter the kernel using the nfssvc system call. The children normally remain kernel resident, providing a process context for the NFS RPC daemons. Typical systems run four to six **nfsd** daemons. If **nfsd** is providing datagram service, it will create a datagram socket when it is started. If **nfsd** is providing stream service, connected stream sockets will be passed in by the master **nfsd** daemon in response to connection-oriented connection requests from clients, When a request arrives on a datagram or stream socket, there is an upcall from the socket layer that invokes the nfsrv_rcv() routine. The *nfsrv_rcv()* call takes the message from the socket receive queue and dispatches that message to an available **nfsd** daemon. The **nfsd** daemon verifies the sender, and then passes the request to the appropriate local filesystem for processing. When the result returns from the filesystem, it is returned to the requesting client. The nfsd daemon is then ready to loop back and to service another request. The maximum degree of concurrency on the server is determined by the number of **nfsd** daemons that are started.

For connection-oriented transport protocols, such as TCP, there is one connection for each client-to-server mount point. For datagram-oriented protocols, such as UDP, the server creates a fixed number of incoming RPC sockets when it starts its **nfsd** daemons; clients create one socket for each imported mount point. The socket for a mount point is created by the **mount** command on the client, which then uses it to communicate with the **mountd** daemon on the server. Once the client-to-server connection is established, the daemon processes on a connection-oriented protocol may do additional verification, such as Kerberos authentication. Once the connection is created and verified, the socket is passed into the kernel. If the connection breaks while the mount point is still active, the client will attempt a reconnect with a new socket.

The client side can operate without any daemons running, but the system administrator can improve performance by running several **nfsiod** daemons (these daemons provide the same service as the Sun **biod** daemons). The purpose of the **nfsiod** daemons is to do asynchronous read-aheads and write-behinds. They are typically started when the kernel begins running multi-user. They enter the kernel using the *nfssvc* system call, and they remain kernel resident, providing a process context for the NFS RPC client side. In their absence, each read or write of an NFS file that cannot be serviced from the local client cache must be done in the context of the requesting process. The process sleeps while the RPC is sent to the server, the RPC is handled by the server, and a reply sent back. No read-aheads are done, and write operations proceed at

the disk-write speed of the server. When present, the **nfsiod** daemons provide a separate context in which to issue RPC requests to a server. When a file is written, the data are copied into the buffer cache on the client. The buffer is then passed to a waiting **nfsiod** that does the RPC to the server and awaits the reply. When the reply arrives, **nfsiod** updates the local buffer to mark that buffer as written. Meanwhile, the process that did the write can continue running. The Sun Microsystems reference port of the NFS protocol flushes all the blocks of a file to the server when that file is closed. If all the dirty blocks have been written to the server when a process closes a file that it has been writing, it will not have to wait for them to be flushed. The NQNFS protocol does not flush all the blocks of a file to the server when that file is closed.

When reading a file, the client first hands a read-ahead request to the **nfsiod** that does the RPC to the server. It then looks up the buffer that it has been requested to read. If the sought-after buffer is already in the cache because of a previous read-ahead request, then it can proceed without waiting. Otherwise, it must do an RPC to the server and wait for the reply. The interactions between the client and server daemons when I/O is done are shown in Fig. 9.3.

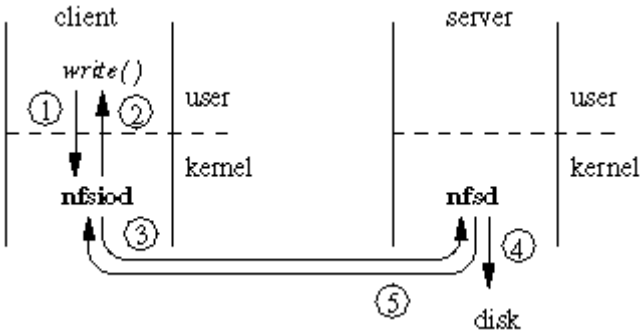


**Figure 9.3** Daemon interaction when I/O is done. Step 1: The client's process does a *write* system call. Step 2: The data to be written are copied into a kernel buffer on the client, and the *write* system call returns. Step 3: An **nfsiod** daemon awakens inside the client's kernel, picks up the dirty buffer, and sends the buffer to the server. Step 4: The in- coming write request is delivered to the next available **nfsd** daemon running inside the kernel on the server. The server's **nfsd** daemon writes the data to the appropriate local disk, and waits for the disk I/O to complete. Step 5: After the I/O has completed, the server's **nfsd** daemon sends back an acknowledgment of the I/O to the waiting **nfsiod** daemon on the client. On receipt of the acknowledgment, the client's **nfsiod** daemon marks the buffer as clean.

## Client Server Interactions

A local filesystem is unaffected by network service disruptions. It is always avail- able to the users on the machine unless there is a catastrophic event, such as a disk or power failure. Since the entire machine hangs or crashes, the kernel does not need to concern itself with how to handle the processes that were accessing the filesystem. By contrast, the client end of a network filesystem must have ways to handle processes that are accessing remote files when the client is still running, but the server becomes unreachable or crashes. Each NFS mount point is provided with three alternatives for dealing with server unavailability:

1. The default is a *hard mount* that will continue to try to contact the server ``forever'' to complete the filesystem access. This type of mount is appropriate when processes on the client that access files in the filesystem do not tolerate I/O system calls that return transient errors. A hard mount is used for processes for which access to the filesystem is critical for normal system operation. It is also useful if the client has a long-running program that simply wants to wait for the server to resume operation (e.g., after the server is taken down to run dumps).

2. The other extreme is a *soft mount* that retries an RPC a specified number of times, and then the corresponding system call returns with a transient error. For a connection-oriented protocol, the actual RPC request is not retransmitted; instead, NFS depends on the protocol retransmission to do the retries. If a response is not returned within the specified time, the corresponding system call returns with a transient error. The problem with this type of mount is that most applications do not expect a transient error return from I/O system calls (since they never occur on a local filesystem). Often, they will mistakenly interpret the transient error as a permanent error, and will exit prematurely. An additional problem is deciding how long to set the timeout period. If it is set too low, error returns will start occurring whenever the NFS server is slow because of heavy load. Alternately, a large retry limit can result in a process hung for a long time because of a crashed server or network partitioning.

3. Most system administrators take a middle ground by using an *interruptible mount* that will wait forever like a hard mount, but checks to see whether a termination signal is pending for any process that is waiting for a server response. If a signal (such as an interrupt) is sent to a process waiting for an NFS server, the corresponding I/O system call returns with a transient error. Normally, the process is terminated by the signal. If the process chooses to catch the signal, then it can decide how to handle the transient failure. This mount option allows interactive programs to be aborted when a server fails, while allowing long-running processes to await the server's return.

The original NFS implementation had only the first two options. Since neither of these two options was ideal for interactive use of the filesystem, the third option was developed as a compromise solution.

## RPC Transport Issues

The NFS Version 2 protocol runs over UDP/IP transport by sending each request-reply message in a single UDP datagram. Since UDP does not guarantee datagram delivery, a timer is started, and if a timeout occurs before the corresponding RPC reply is received, the RPC request is retransmitted. At best, an extraneous RPC request retransmit increases the load on the server and can result in damaged files on the server or spurious errors being returned to the client when nonidempotent RPCs are redone. A recent-request cache normally is used on the server to minimize the negative effect of redoing a duplicate RPC request [Juszczak, 1989].

The amount of time that the client waits before resending an RPC request is called the *round-trip timeout (RTT)*. Figuring out an appropriate value for the RTT is difficult. The RTT value is for the entire RPC operation, including transmitting the RPC message to the server, queuing at the server for an **nfsd**, doing any required I/O operations, and sending the RPC reply message back to the client. It can be highly variable for even a moderately loaded NFS server. As a result, the RTT interval must be a conservative (large) estimate to avoid extraneous RPC request retransmits. Adjusting the RTT interval dynamically and applying a congestion window on outstanding requests has been shown to be of some help with the retransmission problem [Nowicki, 1989].

On an Ethernet with the default 8-Kbyte read write data size, the read write reply-request will be an 8+-Kbyte UDP datagram that normally must be broken into at least six fragments at the IP layer for transmission. For IP fragments to be reassembled successfully into the IP datagram at the receive end, all fragments must be received at the destination. If even one fragment is lost or damaged in transit, the entire RPC message must be retransmitted, and the entire RPC redone. This problem can be exaggerated if the server is multiple hops away from the client through routers or slow links. It can also be nearly fatal if the network inter- face on the client or server cannot handle the reception of back-to-back network packets [Kent & Mogul, 1987].

An alternative to all this madness is to run NFS over TCP transport, instead of over UDP. Since TCP provides reliable delivery with congestion control, it avoids the problems associated with UDP. Because the retransmissions are done at the TCP level, instead of at the RPC level, the only time that a duplicate RPC will be sent to the server is when the server crashes or there is an extended network partition that causes the TCP connection to break after an RPC has been received but not acknowledged to the client. Here, the client will resend the RPC after the server reboots, because it does not know that the RPC has been received.

The use of TCP also permits the use of read and write data sizes greater than the 8-Kbyte limit for UDP transport. Using large data sizes allows TCP to use the full duplex bandwidth of the network effectively, before being forced to stop and wait for RPC response from the server. NFS over TCP usually delivers comparable to significantly better performance than NFS over UDP, unless the client or server processor is slow. For processors running at less than 10 million instructions per second (MIPS), the extra CPU overhead of using TCP transport becomes significant.

The main problem with using TCP transport with Version 2 of NFS is that it is supported between only BSD and a few other vendors clients and servers. However, the clear superiority demonstrated by the Version 2 BSD TCP implementation of NFS convinced the group at Sun Microsystems implementing NFS Version 3 to make TCP the default transport. Thus, a Version 3 Sun client will first try to connect using TCP; only if the server refuses will it fall back to using UDP.

## Security Issues

NFS is not secure because the protocol was not designed with security in mind. Despite several attempts to fix security problems, NFS security is still limited. Encryption is needed to build a secure protocol, but robust encryption cannot be exported from the United States. So, even if building a secure protocol were possible, doing so would be pointless, because all the file data are sent around the net in clear text. Even if someone is unable to get your server to send them a sensitive file, they can just wait until a legitimate user accesses it, and then can pick it up as it goes by on the net.

NFS export control is at the granularity of local filesystems. Associated with each local filesystem mount point is a list of the hosts to which that filesystem may be exported. A local filesystem may be exported to a specific host, to all hosts that match a subnet mask, or to all other hosts (the world). For each host or group of hosts, the filesystem can be exported read-only or read write. In addition, a server may specify a set of subdirectories within the filesystem that may be mounted. However, this list of mount points is enforced by only the **mountd** daemon. If a malicious client wishes to do so, it can access any part of a filesystem that is exported to it.

The final determination of exportability is made by the list maintained in the kernel. So, even if a rogue client manages to snoop the net and to steal a file handle for the mount point of a valid client, the kernel will refuse to accept the file handle unless the client presenting that handle is on the kernel's export list. When NFS is running with TCP, the check is done once when the connection is established. When NFS is running with UDP, the check must be done for every RPC request.

The NFS server also permits limited remapping of user credentials. Typically, the credential for the superuser is not trusted and is remapped to the low-privilege user ``nobody.'' The credentials of all other users can be accepted as given or also mapped to a default user (typically ``nobody''). Use of the client UID and GID list unchanged on the server implies that the UID and GID space are common between the client and server (i.e., UID N on the client must refer to the same user

on the server). The system administrator can support more complex UID and GID mappings by using the **umapfs** filesystem described in Section 6.7.

The system administrator can increase security by using Kerberos credentials, instead of accepting arbitrary user credentials sent without encryption by clients of unknown trustworthiness [Steiner et al, 1988]. When a new user on a client wants to begin accessing files in an NFS filesystem that is exported using Kerberos, the client must provide a Kerberos ticket to authenticate the user on the server. If successful, the system looks up the Kerberos principal in the server's password and group databases to get a set of credentials, and passes in to the server **nfsd** a local translation of the client UID to these credentials. The **nfsd** daemons run entirely within the kernel except when a Kerberos ticket is received. To avoid putting all the Kerberos authentication into the kernel, the **nfsd** returns from the kernel temporarily to verify the ticket using the Kerberos libraries, and then returns to the kernel with the results.

The NFS implementation with Kerberos uses encrypted timestamps to avert replay attempts. Each RPC request includes a timestamp that is encrypted by the client and decrypted by the server using a session key that has been exchanged as part of the initial Kerberos authentication. Each timestamp can be used only once, and must be within a few minutes of the current time recorded by the server. This implementation requires that the client and server clocks be kept within a few minutes of synchronization (this requirement is already imposed to run Kerberos). It also requires that the server keep copies of all timestamps that it has received that are within the time range that it will accept, so that it can verify that a timestamp is not being reused. Alternatively, the server can require that timestamps from each of its clients be monotonically increasing. However, this algorithm will cause RPC requests that arrive out of order to be rejected. The mechanism of using Kerberos for authentication of NFS requests is not well defined, and the 4.4BSD implementation has not been tested for interoperability with other vendors. Thus, Kerberos can be used only between 4.4BSD clients and servers.

## 9.3 Techniques for Improving Performance

Remote filesystems provide a challenging performance problem: Providing both a coherent network wide view of the data and delivering that data quickly are often conflicting goals. The server can maintain coherency easily by keeping a single repository for the data and sending them out to each client when the clients need them; this approach tends to be slow, because every data access requires the client to wait for an RPC round-trip time. The delay is further aggravated by the huge load that it puts on a server that must service every I/O request from its clients. To increase performance and to reduce server load, remote filesystem protocols attempt to cache frequently used data on the clients themselves. If the cache is designed properly, the client will be able to satisfy many of the client's I/O requests directly from the cache. Doing such accesses is faster than communicating with the server, reducing latency on the client and load on the server and network. The hard part of client caching is keeping the caches coherent that is, ensuring that each client quickly replaces any cached data that are modified by writes done on other clients. If a first client writes a file that is later read by a second client, the second client wants to see the data written by the first client, rather than the stale data that were in the file previously. There are two main ways that the stale data may be read accidentally:

1. If the second client has stale data sitting in its cache, the client may use those data because it does not know that newer data are available.

2. The first client may have new data sitting in its cache, but may not yet have written those data back to the server. Here, even if the second client asks the server for up-to-date data, the server may return the stale data because it does not know that one of its clients has a newer version of the file in that client's cache.

The second of these problems is related to the way that client writing is done. Synchronous writing requires that all writes be pushed through to the server during the *write* system call. This approach is the most consistent, because the server always has the most recently written data. It also permits any write errors, such as ``filesystem out of space,'' to be propagated back to the client process via the *write* system-call return. With an NFS filesystem using synchronous writing, error returns most closely parallel those from a local filesystem. Unfortunately, this approach restricts the client to only one write per RPC round-trip time.

An alternative to synchronous writing is delayed writing, where the *write* system call returns as soon as the data are cached on the client; the data are written to the server sometime later. This approach permits client writing to occur at the rate of local storage access up to the size of the local cache. Also, for cases where file truncation or deletion occurs shortly after writing, the write to the server may be avoided entirely, because the data have already been deleted. Avoiding the data push saves the client time and reduces load on the server.

There are some drawbacks to delayed writing. To provide full consistency, the server must notify the client when another client wants to read or write the file, so that the delayed writes can be written back to the server. There are also problems with the propagation of errors back to the client process that issued the *write* system call. For example, a semantic change is introduced by delayed-write caching when the file server is full. Here, delayed-write RPC requests can fail with an ``out of space'' error. If the data are sent back to the server when the file is closed, the error can be detected if the application checks the return value from the *close* system call. For delayed writes, written data may not be sent back to the server until after the process that did the write has exited long after it can be notified of any errors. The only solution is to modify programs writing an important file to do an *fsync* system call and to check for an error return from that call, instead of depending on getting errors from *write* or *close.* Finally, there is a risk of the loss of recently written data if the client crashes before the data are written back to the server.

A compromise between synchronous writing and delayed writing is asynchronous writing. The write to the server is started during the *write* system call, but the *write* system call returns before the write completes. This approach minimizes the risk of data loss because of a client crash, but negates the possibility of reducing server write load by discarding writes when a file is truncated or deleted.

The simplest mechanism for maintaining full cache consistency is the one used by Sprite that disables all client caching of the file whenever concurrent write sharing might occur [Nelson et al, 1988]. Since NFS has no way of knowing when write sharing might occur, it tries to bound the period of inconsistency by writing the data back when a file is closed. Files that are open for long periods are written back at 30-second intervals when the filesystem is synchronized. Thus, the NFS implementation does a mix of asynchronous and delayed writing, but always pushes all writes to the server on close. Pushing the delayed writes on close negates much of the performance advantage of delayed writing, because the delays that were avoided in the *write* system calls are observed in the *close* system call. With this approach, the server is always aware of all changes made by its clients with a maximum delay of 30 seconds and usually sooner, because most files are open only briefly for writing.

The server maintains read consistency by always having a client verify the contents of its cache before using that cache. When a client reads data, it first checks for the data in its cache. Each cache entry is stamped with an attribute that shows the most recent time that the server says that the data were modified. If the data are found in the cache, the client sends a timestamp RPC request to its server to find out when the data were last modified. If the modification time returned by the server matches that associated with the cache, the client uses the data in its cache; otherwise, it arranges to replace the data in its cache with the new data.

The problem with checking with the server on every cache access is that the client still experiences an RPC round-trip delay for each file access, and the server is still inundated with RPC requests, although they are considerably quicker to handle than are full I/O operations. To reduce this client latency and server load, most NFS implementations track how recently the server has been asked about each cache block. The client then uses a tunable parameter that is typically set at a few seconds to delay asking the server about a cache block. If an I/O request finds a cache block and the server has been asked about the validity of that block within the delay period, the client does not ask the server again, but rather just uses the block. Because certain blocks are used many times in succession, the server will be asked about them only once, rather than on every access. For example, the directory block for the **/usr/include** directory will be accessed once for each **#include** in a source file that is being compiled. The drawback to this approach is that changes made by other clients may not be noticed for up to the delay number of seconds.

A more consistent approach used by some network filesystems is to use a *callback* scheme where the server keeps track of all the files that each of its clients has cached. When a cached file is modified, the server notifies the clients holding that file so that they can purge it from their cache. This algorithm dramatically reduces the number of queries from the client to the server, with the effect of decreasing client I/O latency and server load [Howard et al, 1988]. The drawback is that this approach introduces state into the server because the server must remember the clients that it is serving and the set of files that they have cached. If the server crashes, it must rebuild this state before it can begin running again. Rebuilding the server state is a significant problem when everything is running properly; it gets even more complicated and time consuming when it is aggravated by network partitions that prevent the server from communicating with some of its clients [Mogul, 1993].

The 4.4BSD NFS implementation uses asynchronous writes while a file is open, but synchronously waits for all data to be written when the file is closed. This approach gains the speed benefit of writing asynchronously, yet ensures that any delayed errors will be reported no later than the point at which the file is closed. The implementation will query the server about the attributes of a file at most once every 3 seconds. This 3-second period reduces network traffic for files accessed frequently, yet ensures that any changes to a file are detected with no more than a 3-second delay. Although these heuristics provide tolerable semantics, they are noticeably imperfect. More consistent semantics at lower cost are available with the NQNFS lease protocol described in the next section.

## Leases

The NQNFS protocol is designed to maintain full cache consistency between clients in a crash-tolerant manner. It is an adaptation of the NFS protocol such that the server supports both NFS and NQNFS clients while maintaining full consistency between the server and NQNFS clients. The protocol maintains cache consistency by using short-term leases instead of hard-state information about open files [Gray & Cheriton, 1989]. A *lease* is a ticket permitting an activity that is valid until some expiration time. As long as a client holds a valid lease, it knows that the server will give it a callback if the file status changes. Once the lease has expired, the client must contact the server if it wants to use the cached data.

Leases are issued using time intervals rather than absolute times to avoid the requirement of time-of-day clock synchronization. There are three important time constants known to the server. The *maximum_lease_term* sets an upper bound on lease duration typically, 30 seconds to 1 minute. The *clock_skew* is added to all lease terms on the server to correct for differing clock speeds between the client and server. The *write_slack* is the number of seconds that the server is willing to wait for a client with an expired write-caching lease to push dirty writes.

Contacting the server after the lease has expired is similar to the NFS technique for reducing server load by checking the validity of data only every few seconds. The main difference is that the server tracks its clients' cached files, so there are never periods of time when the client is using stale data. Thus, the time used for leases can be considerably longer than the few seconds that clients are willing to tolerate possibly stale data. The effect of this longer lease time is to reduce the number of server calls almost to the level found in a full callback implementation such as the Andrew Filesystem [Howard et al, 1988]. Unlike the callback mechanism, state recovery with leases is trivial. The server needs only to wait for the lease's expiration time to pass, and then to resume operation. Once all the leases have expired, the clients will always communicate with the server before using any of their cached data. The lease expiration time is usually shorter than the time it takes most servers to reboot, so the server can effectively resume operation as soon as it is running. If the machine does manage to reboot more quickly than the lease expiration time, then it must wait until all leases have expired before resuming operation.

An additional benefit of using leases rather than hard state information is that leases use much less server memory. If each piece of state requires 64 bytes, a large server with hundreds of clients and a peak throughput of 2000 RPC requests per second will typically only use a few hundred Kbyte of memory for leases, with a worst case of about 3 Mbyte. Even if a server has exhausted lease storage, it can simply wait a few seconds for a lease to expire and free up a record. By contrast, a server with hard state must store records for all files currently open by all clients. The memory requirements are 3 to 12 Mbyte of memory per 100 clients served.

Whenever a client wishes to cache data for a file, it must hold a valid lease. There are three types of leases: noncaching, read caching, and write caching. A *noncaching lease* requires that all file operations be done synchronously with the server. A *read-caching lease* allows for client data caching, but no file modifications may be done. A *write-caching lease* allows for client caching of writes for the period of the lease. If a client has cached write data that are not yet written to the server when a write-cache lease has almost expired, it will attempt to extend the lease. If the extension fails, the client is required to push the written data.

If all the clients of a file are reading it, they will all be granted a read-caching lease. A read-caching lease allows one or more clients to cache data, but they may not make any modifications to the data. Figure 9.4 shows a typical read-caching scenario. The vertical solid black lines depict the lease records. Note that the time lines are not drawn to scale, since a client server interaction will normally take less than 100 milliseconds, whereas the normal lease duration is 30 seconds. Every lease includes the time that the file was last modified. The client can use this timestamp to ensure that its cached data are still current. Initially, client A gets a read-caching lease for the file. Later, client A renews that lease and uses it to verify that the data in its cache are still valid. Concurrently, client B is able to obtain a read-caching lease for the same file.

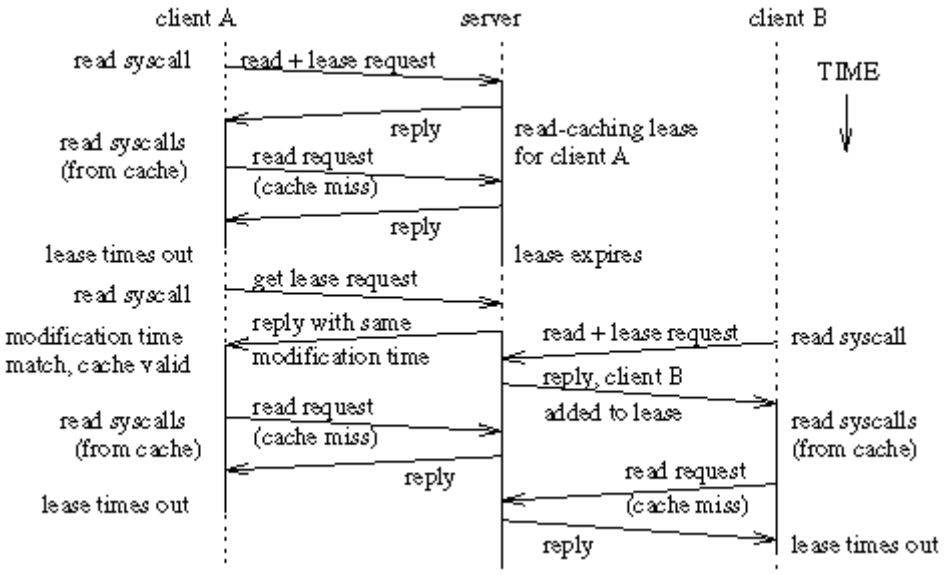


**Figure 9.4** Read-caching leases. Solid vertical lines represent valid leases.

If a single client wants to write a file and there are no readers of that file, the client will be issued a write-caching lease. A write-caching lease permits delayed write caching, but requires that all data be pushed to the server when the lease expires or is terminated by an *eviction notice*. When a write-caching lease has almost expired, the client will attempt to extend the lease if the file is still open, but is required to push the delayed writes to the server if renewal fails (see Fig. 9.5). The writes may not arrive at the server until after the write lease has expired on the client. A consistency problem is avoided because the server keeps its write lease valid for *write_slack* seconds longer than the time given in the lease issued to the client. In addition, writes to the file by the lease-holding client cause the lease expiration time to be extended to at least *write_slack* seconds. This *write_slack* period is conservatively estimated as the extra time that the client will need to write back any written data that it has cached. If the value selected for *write_slack* is too short, a write RPC may arrive after the write lease has expired on the server. Although this write RPC will result in another client seeing an inconsistency, that inconsistency is no more problematic than the semantics that NFS normally provides.

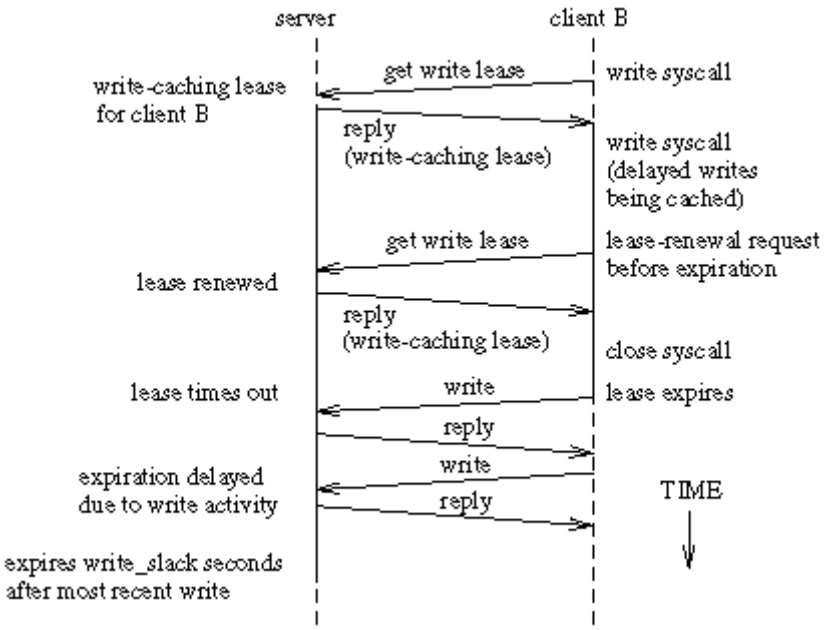


**Figure 9.5** Write-caching lease. Solid vertical lines represent valid leases.

The server is responsible for maintaining consistency among the NQNFS clients by disabling client caching whenever a server file operation would cause inconsistencies. The possibility of inconsistencies occurs whenever a client has a write-caching lease and any other client or a local operation on the server tries to access the file, or when a modify operation is attempted on a file being read cached by clients. If one of these conditions occurs, then all clients will be issued noncaching leases. With a noncaching lease, all reads and writes will be done through the server, so clients will always get the most recent data. Figure 9.6 shows how read and write leases are replaced by a noncaching lease when there is the potential for write sharing. Initially, the file is read by client A. Later, it is written by client B. While client B is still writing, client A issues another read request. Here, the server sends an ``eviction notice'' message to client B, and then waits for lease termination. Client B writes back its dirty data, then sends a ``vacated'' message. Finally, the server issues noncaching leases to both clients. In general, lease termination occurs when a ``vacated'' message has been received from all the clients that have signed the lease or when the lease has expired. The server does not wait for a reply for the message pair ``eviction notice'' and ``vacated,'' as it does for all other RPC messages; they are sent asynchronously to avoid the server waiting indefinitely for a reply from a dead client.

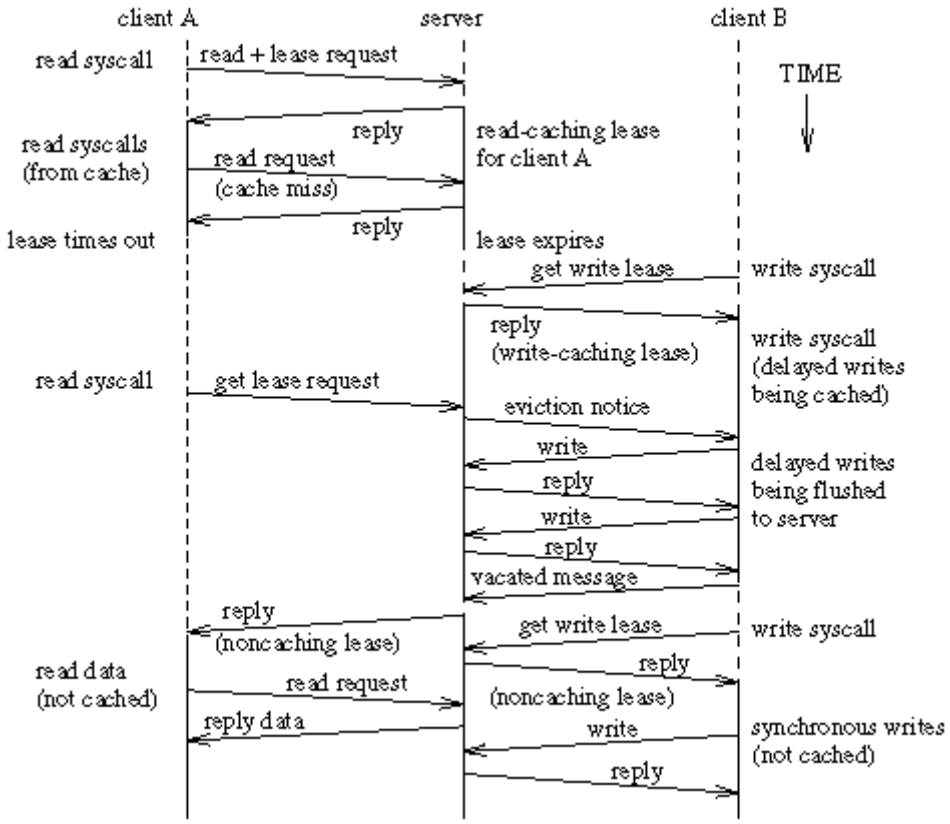


**Figure 9.6** Write-sharing leases. Solid vertical lines represent valid leases.

A client gets leases either by doing a specific lease RPC or by including a lease request with another RPC. Most NQNFS RPC requests allow a lease request to be added to them. Combining lease requests with other RPC requests minimizes the amount of extra network traffic. A typical combination can be done when a file is opened. The client must do an RPC to get the handle for the file to be opened. It can combine the lease request, because it knows at the time of the open whether it will need a read or a write lease. All leases are at the granularity of a file, because all NFS RPC requests operate on individual files, and NFS has no intrinsic notion of a file hierarchy. Directories, symbolic links, and file attributes may be read cached but are not write cached. The exception is the file-size attribute that is updated during cached writing on the client to reflect a growing file. Leases have the advantage that they are typically required only at times when other I/O operations occur. Thus, lease requests can almost always be piggy-backed on other RPC requests, avoiding some of the overhead associated with the explicit open and close RPC required by a long-term callback implementation.

The server handles operations from local processes and from remote clients that are not using the NQNFS protocol by issuing short-term leases for the duration of each file operation or RPC. For example, a request to create a new file will get a short-term write lease on the directory in which the file is being created. Before that write lease is issued, the server will vacate the read leases of all the NQNFS clients that have cached data for that directory. Because the server gets leases for all non-NQNFS activity, consistency is maintained between the server and NQNFS clients, even when local or NFS clients are modifying the filesystem. The NFS clients will continue to be no more or less consistent with the server than they were without leases.

## Crash Recovery

The server must maintain the state of all the current leases held by its clients. The benefit of using short-term leases is that, *maximum_lease_term* seconds after the server stops issuing leases, it knows that there are no current leases left. As such, server crash recovery does not require any state recovery. After rebooting, the server simply refuses to service any RPC requests except for writes (predominantly from clients that previously held write leases) until *write_slack* seconds after the final lease would have expired. For machines that cannot calculate the time that they crashed, the final-lease expiration time can be estimated safely as

*boot_time* + *maximum_lease_term* + *write_slack* + *clock_skew*

Here, *boot_time* is the time that the kernel began running after the kernel was booted. With a *maximum_lease_term* 30 to 60 seconds, and *clock_skew* and *write_slack* at most a few seconds, this delay amounts to about 1 minute, which for most systems is taken up with the server rebooting process. When this time has passed, the server will have no outstanding leases. The clients will have had at least *write_slack* seconds to get written data to the server, so the server should be up to date. After this, the server resumes normal operation.

There is another failure condition that can occur when the server is congested. In the worst-case scenario, the client pushes dirty writes to the server, but a large request queue on the server delays these writes for more than *write_slack* seconds. In an effort to minimize the effect of these *recovery storms*, the server replies ``try again later'' to the RPC requests that it is not yet ready to service [Baker & Ousterhout, 1991]. The server takes two steps to ensure that all clients have been able to write back their written data. First, a write-caching lease is terminated on the server only when there are have been no writes to the file during the previous write_slack seconds. Second, the server will not accept any requests other than writes until it has not been overloaded during the previous *write_slack* seconds. A server is considered overloaded when there are pending RPC requests and all its **nfsd** processes are busy.

Another problem that is solved by short-term leases is how to handle a crashed or partitioned client that holds a lease that the server wishes to vacate. The server detects this problem when it needs to vacate a lease so that it can issue a lease to a second client, and the first client holding the lease fails to respond to the vacate request. Here, the server can simply wait for the first client's lease to expire before issuing the new one to the second client. When the first client reboots or gets reconnected to the server, it simply reacquires any leases it now needs. If a client-to-server network connection is severed just before a write- caching lease expires, the client cannot push the dirty writes to the server. Other clients that can contact the server will continue to be able to access the file and will see the old data. Since the write-caching lease has expired on the client, the client will synchronize with the server as soon as the network connection has been re-established. This delay can be avoided with a write-through policy.

A detailed comparison of the effects of leases on performance is given in [Macklem, 1994a]. Briefly, leases are most helpful when a server or network is loaded heavily. Here, leases allow up to 30 to 50 percent more clients to use a network and server before beginning to experience a level of congestion equal to what they would on a network and server that were not using leases. In addition, leases provide better consistency and lower latency for clients, independent of the load. Although leases are new enough that they are not widely used in commercial implementations of NFS today, leases or a similar mechanism will need to be added to commercial versions of NFS if NFS is to be able to compete effectively against other remote filesystems, such as Andrew.

## References

Baker & Ousterhout, 1991.

M. Baker & J. Ousterhout, "A vailability in the Sprite Distributed File System," *ACM Operating System Review*, vol. 25, no. 2, p. 95 98, April 1991.

Birrell & Nelson, 1984.

A. D. Birrell & B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, p. 39 59, Association for Computing Machinery, February 1984.

Gray & Cheriton, 1989.

C. Gray & D. Cheriton, "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," *Proceedings of the Twelfth Symposium on Operating Systems Principles*, p. 202 210, December 1989.

Howard, 1988.

J. Howard, "An Overview of the Andrew File System," *USENIX Association Conference Proceedings*, p. 23 26, January 1988.

Howard et al, 1988.

J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, & M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, p. 51 81, Association for Computing Machinery, February 1988.

Juszczak, 1989.

C. Juszczak, "Improving the Performance and Correctness of an NFS Ser- ver," USENIX Association Conference Proceedings, p. 53 63, January 1989.

Kent & Mogul, 1987.

C. Kent & J. Mogul, "Fragmentation Considered Harmful," Research Report 87/3, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, December 1987.

Macklem, 1991.

R. Macklem, "Lessons Learned Tuning the 4.3BSD-Reno Implementation of the NFS Protocol," USENIX Association Conference Proceedings,p. 53 64, January 1991.

Macklem, 1994a.

R. Macklem, "Not Quite NFS, Soft Cache Consistency for NFS," USENIX Association Conference Proceedings, p. 261 278, January 1994.

Macklem, 1994b.

R. Macklem, "The 4.4BSD NFS Implementation," in 4.4BSD System Manager's Manual, p. 6:1 14, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

Mogul, 1993.

J. Mogul, "Recovery in Spritely NFS," Research Report 93/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, June 1993.

Nelson et al, 1988.

M. Nelson, B. Welch, & J. Ousterhout, "Caching in the Sprite Network File System," ACM Transactions on Computer Systems, vol. 6, no. 1, p. 134 154, Association for Computing Machinery, February 1988.

Nowicki, 1989.

B. Nowicki, "Transport Issues in the Network File System," Computer Communications Review, vol. 19, no. 2, p. 16 20, April 1989.

Pawlowski et al, 1994.

B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, & D. Hitz, "NFS Version 3: Design and Implementation," USENIX Association Conference Proceedings, p. 137 151, June 1994.

Reid, 1987.

Irving Reid, "RPCC: A Stub Compiler for Sun RPC," USENIX Association Conference Proceedings, p. 357 366, June 1987.

Rifkin et al, 1986.

A. Rifkin, M. Forbes, R. Hamilton, M. Sabrio, S. Shah, & K. Yueh, "RFS Architectural Overview," USENIX Association Conference Proceedings,p. 248 259, June 1986.

Sandberg et al, 1985.

R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, & B. Lyon, "Design and Implementation of the Sun Network Filesystem," USENIX Association Conference Proceedings, p. 119 130, June 1985.

Steiner et al, 1988.

J. Steiner, C. Neuman, & J. Schiller, "Kerberos: An Authentication Service for Open Network Systems," USENIX Association Conference Proceedings, p. 191 202, February 1988.

Sun Microsystems, 1989.

Sun Microsystems, "NFS: Network File System Protocol Specification," RFC 1094, available by anonymous FTP from ds.internic.net, March 1989.

Sun Microsystems, 1993.

Sun Microsystems, NFS: Network File System Version 3 Protocol Specification, Sun Microsystems, Mountain View, CA, June 1993.

Walsh et al, 1985.

D. Walsh, B. Lyon, G. Sager, J. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, & P. Weiss, "Overview of the Sun Network File System," USENIX Association Conference Proceedings, p. 117 124, January 1985.