Appendix 4–B. Case study of the Network File System (NFS)

The network file system (NFS), designed by Sun Microsystems, Inc. in the 1980s, is a client/service application that provides shared file storage for clients across a network. An NFS client grafts a remote file system onto the client's local file system name space and makes it behave like a local UNIX file system (see appendix 2–A). Multiple clients can mount the same remote file system so that users can share files.

The need for NFS arose because of technology improvements. Before the 1980s computers were so expensive that each one had to be shared among multiple users and each computer had a single file system. But a benefit of the economic pressure was it allowed for easy collaboration, because users could share files easily. In the early 1980s, it became economically feasible to build workstations, which allowed each engineer to have a private computer. But, users desired to still have a shared file system for ease of collaboration. NFS provides exactly that: it allows a user at any workstation to use files stored on a shared server, a powerful workstation with local disks but often without a graphical display.

NFS also simplifies the management of a collection of workstations. Without NFS, a system administrator must manage each workstation and, for example, arrange for backups of each workstation's local disk. NFS allows for centralized management; for example, a system administrator needs to back up only the disks of the server to archive the file system. In the 1980s, the setup had also a cost benefit: NFS allowed organizations to buy workstations without disks, saving the cost of a disk interface on every workstation and, at the time, the cost of unused disk space on each workstation.

The design of NFS had four major goals. It should work with existing applications, which means NFS ideally should provide the same semantics as a local UNIX file system. NFS should be deployable easily, which means its implementation should be able to retrofit into existing UNIX systems. The client should be implementable in other operating systems such as Microsoft's DOS, so that a user on a personal computer can have access to the files on an NFS server; this goal implies that the client design cannot be too UNIX-specific. Finally, NFS should be efficient enough to be tolerable to users, but it doesn't have to provide as high performance as local file systems. NFS only partially achieves these goals.

This appendix describes version 2 of NFS. Version 1 was never deployed outside of Sun Microsystems, while version 2 has been in use since 1987. The appendix concludes with a brief summary of the changes in version 3 (1990s) and 4 (early 2000s), which address weaknesses in version 2.

1. Naming remote files and directories

To programs, NFS appears as a UNIX file system providing the file interface presented in appendix 2–A. User programs can name remote files in the same way as local files. When a user program invokes, say, OPEN ("/users/smith/.profile", READONLY), it cannot tell from the path name whether "users" or "smith" are local or remote directories.

To make naming remote file transparent to users and their programs, the client must mount the root directory of a remote file system on the local name space. NFS performs this operation by using a separate program, called the *mounter.* This program serves a similar function as the MOUNT call (see page 2–62); it grafts the remote file system— named by *host*:*path*, where *host* is a DNS name and *path* a path name—onto the local file name space. The mounter sends a remote procedure call to the file server *host* and asks for a *file handle,* a 32-byte name*,* for the inode of *path*. On receiving the reply, the client marks the mount point in the local file system as a remote file system. It also remembers the file handle for *path* and the network address for the server.

NFS doesn't use path names to name files and directories internally, but instead uses file handles. To the client a file handle is a 32-byte opaque name that identifies an inode on a remote server. A client obtains file handles from the server when the client mounts a remote file system or it looks up a file in a directory on the server. In all subsequent remote procedures calls to the server for that file, the client includes the file handle.

One might wonder why the NFS designers chose to use file handles to name files and directories instead of path names. To see why consider the following scenario with two user programs running on different clients:

Program 1 on client 1	**Program 2 on client 2**	Time
CHDIR ("dir1");		
fd = OPEN ("f", READONLY);		
	RENAME ("dir1", "dir2");	
	RENAME ("dir3", "dir1");	
READ (*fd*, *buf*, *n*);		

When program 1 invokes READ, does the program read data from "dir1/f", or "dir2/f"? If the two programs where running on the same computer and sharing a local UNIX file system, program 1 would read "dir2/f", according to the UNIX specification. The goal is that NFS should provide the same behavior. Unfortunately, if the NFS client were to use path names, then the READ call would result in a remote procedure for the file "dir1/f". To avoid this problem, NFS clients name files and directories using file handles.

To the server a file handle is a structured name—containing a *file system identifier,* an *inode number*, and a *generation number*—which it uses to locate the file. The file system identifier allows the server to identify the file system responsible for the file. The inode number (see page 2–55) allows the identified file system to locate the file on the disk.

The file handle includes a generation number to detect that a program on a client 1 deletes a file and creates a new one while another program on a client 2 already has opened the original file. If the server should happen to reuse the inode of the old file for the new file, remote procedure calls of client 2 will get the new file, the one created by client 1, instead of the old file. If the two programs where running on the same computer and sharing a local UNIX file system, program 2 would read the old file. The generation number allows NFS to detect this case. When the server reuses an inode, it increases the generation number by one. In the example, client 1 and client 2 would have different file handles, and client 2 will use the old handle. NFS does not provide identical semantics to a local UNIX file system, though, because that would require that the server knew which files are in use; instead, when client 2 uses the file handle it will receive a new error message: “stale file handle”.

File handles are usable names across server failures, so that even if the server computer fails between a client program opening a file and then reading from the file, the server can identify the file using the information in the file handle. Making file handles usable across server failures requires small changes to the server’s on-disk file system: the NFS designers modified the super block to record the file system identifier and inodes to record the generation number for the inode.

2. The NFS remote procedure calls

Table 4-1 shows the remote procedure calls used by NFS. The remote procedure calls are best explained by using an example. Suppose we have the following fragment of a user program:

```
fd = OPEN (“f”, READONLY);
READ (fd, buf, n);
```

Figure 4–10 shows the corresponding timing diagram where “f” is a remote file. The NFS client implements each file system operation using one or remote procedure calls.

In response to the program’s call to OPEN, the NFS client sends the following remote procedure call to the server:

```
LOOKUP (dirfh, “f”)
```

From before the program runs, the client has a file handle for the current working directory's (*dirfh*). It obtained this handle as a result of a previous lookup or as a result of mounting the remote file system.

On receiving the LOOKUP request, the NFS server extracts the file system identifier from *dirfh*, and asks the identified file system to look up the inode for *dirfh*. The identified file system uses the inode number in *dirfh* to locate the directory’s inode. Now the NFS server searches the directory for the inode for “f”. If present, the server creates a handle for the “f”. The handle contains the file system identifier of the local file system, the inode number for “f”, and the generation number stored in “f”’s inode. The NFS server sends this file handle to the client.

On receiving the response, the client allocates the first unused entry in the program’s file descriptor table, stores a reference to f’s file handle in that entry, and returns the index for the entry (*fd)* to the user program.

Table 4-1: NFS remote procedure calls

Remote procedure call	Returns
NULL ()	Do nothing.
LOOKUP (*dirfh*, *name*)	fh and file attributes
CREATE (*dirfh*, *name*, *attr*)	fh and file attributes
REMOVE (*dirfh*, *name*)	status
GETATTR (*fh*)	file attributes
SETATTR (*fh*, *attr*)	file attributes
READ (*fh*, *offset*, *count*)	file attributes and data
WRITE (*fh*, *offset*, *count*, *data*)	file attributes
RENAME (*dirfh*, *name*, *tofh*, *toname*)	status
LINK (*dirfh*, *name*, *tofh*, *toname*)	status
SYMLINK (*dirfh*, *name*, *string*)	status
READLINK (*fh*)	string
MKDIR (*dirfh*, *name*, *attr*)	fh and file attributes
RMDIR (*dirfh*, *name*)	status
READDIR (*dirfh*, *offset*, *count*)	directory entries
STATFS (*fh*)	file system information

Next, the program calls READ (*fd*, *buf*, *n*). The client sends the following remote procedure call to the NFS server:

READ (*fh*, 0, *n*)

Like with the directory file handle, the NFS server looks up the inode for *fh*. Then, the server reads the data and sends the data in a reply message to the client.

The NFS remote procedure calls are designed so that the server can be *stateless*, that is the server doesn't need to maintain any other state than the on-disk files. NFS achieves this property by making each remote procedure call contain all the information necessary to carry out that request. The server does not maintain any state about past remote procedure calls to process a new request. For example, the client must keep track of the file cursor (see page 2–47) and include it as an argument in the READ remote procedure call; the server doesn't. As another example, the file handle contains all information to find the inode on the server, as explained above.

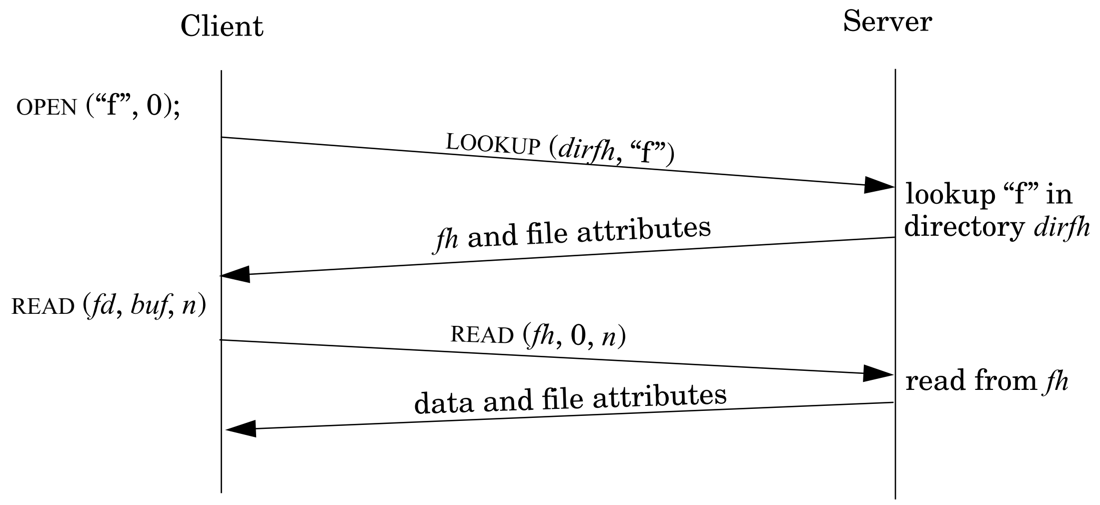

Figure 4–10: Example interaction between a NFS client and service

This stateless property simplifies recovery from server failures: a client can just repeat a request until it receives a reply. In fact, the client cannot tell the difference between a server that failed and recovered, and a server that is slow. Because a client repeats a request until it receives a response, it can happen that the server executes a request twice. That is, NFS implements at-least-once semantics for remote procedure calls. Since many requests are idempotent (e.g., LOOKUP, READ, WRITE) that is not a problem, but for some it is. For example, REMOVE is not idempotent: if the first request succeeds, then a retry will return an error saying that the file doesn't exist.

The first implementation of NFS followed this design exactly and had surprising behavior for users. Consider a user program that calls UNLINK on an existing file that is stored on a remote file system. The NFS client would send a REMOVE remote procedure call and the server would execute it, but it could happen that the network lost the reply. In that case, the client would resend the REMOVE request, the server would execute the request again, and the user program would receive an error saying that the file didn't exist!

Later implementations of NFS address minimize this surprising behavior by providing at-most-once semantics for remote procedure calls. In these implementations, each remote procedure call is tagged with a transaction number and the server maintains some "soft" state, namely a reply cache. The reply cache is indexed by transaction identifier and records the response for the transaction identifier. When the server receives a request, it looks up its transaction identifier in the reply cache. If it is in the cache, it returns the reply from the cache, without re-executing the request. If it is not in the cache, the server processes the request.

The replay cache is soft state because making it hard state is expensive. It would require that the reply cache be stored on a disk and would require a disk write for each remote procedure call to record the response. Because the reply cache is soft state, remote procedure calls provide at-most-once semantics. If the server doesn't fail, a retry of a REMOVE request will receive the same response as the first attempt. If, however, the server fails between the first

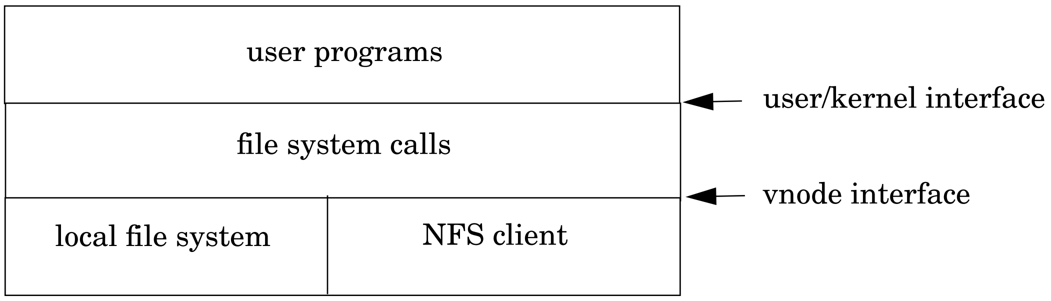

Figure 4–11: Layers in the UNIX implementation

attempt and a retry, a request is executed twice. For the idempotent operations that is not a problem, but for operations like REMOVE the second attempt may generate a different result from the first attempt, if the server failed between the first and second attempt.

Although the stateless property of NFS simplifies recovery, it makes it impossible to implement the UNIX file interface completely correctly, because the UNIX specification requires maintaining state. Consider again the case where one program deletes a file that another program has open. The UNIX specification is that the file exists until the second program closes the file.

If the programs run on different clients, NFS cannot adhere to this specification, because it would require that the server keep state: it would have to maintain a reference count per file, which would be incremented by one on an OPEN and decremented by one on a CLOSE, and persist across server failures. Instead, NFS just does the wrong thing: remote procedure calls of a client can return an error "stale file handle", if a program on another client deletes a file that the first client has open.

NFS does not implement the UNIX specification faithfully, because that simplifies the design of NFS. NFS preserves most of the UNIX semantics and only in rarely encountered situations may users see different behavior. In practice, these rare situations are not a serious problem, and in return NFS gets by with simple recovery.

3. Extending the UNIX file system to support NFS

To implement NFS as an extension of the UNIX file system while minimizing the number of changes required to the UNIX file system, the NFS designers split the file system program by introducing an interface that provides *vnodes*, virtual nodes (see figure 4–11). A vnode is a structure in volatile memory that abstracts whether a file or directory is implemented by a local file system or a remote file system. This design allows many functions in the file system call layer to be implemented in terms of vnodes, without having to worry about whether a file or directory is local or remote.

When a file system call must perform an operation on a file (e.g., reading data), it invokes the corresponding procedure through the vnode interface. The vnode interface has

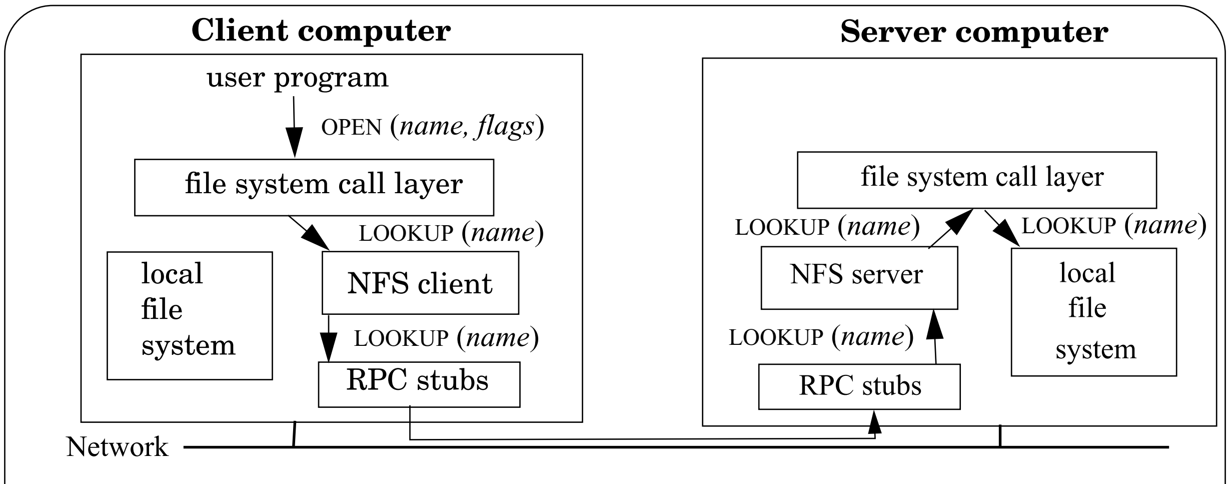

Figure 4–12: NFS implementation in UNIX

procedures for looking up a file name in the contents of a directory vnode, reading from a vnode, writing to a vnode, closing a vnode, etc. The local file system and NFS support their own implementation of these procedures.

By using the vnode interface, most of the code for file descriptor tables, current directory, name lookup, etc. can be moved from the local file system module into the file system call layer with minimal effort. For example, with a few changes the procedure PATHNAME_TO_INODE from appendix 2–A can be modified to be PATHNAME_TO_VNODE and be provided by the file system call layer.

To illustrate the vnode design, consider a user program that invokes OPEN for a file (see figure 4–12). To open the file, the file system call layer invokes PATHNAME_TO_VNODE, passing the vnode for the current working directory and the path name for the file as arguments. PATHNAME_TO_VNODE will parse the path name, invoking LOOKUP in the vnode interface for each component in the path name. If the directory is a local directory, LOOKUP invokes the LOOKUP procedure implemented by the local file system to obtain a vnode for the path name component. If the directory is a remote directory, LOOKUP invokes the LOOKUP procedure implemented by the NFS client.

To implement LOOKUP, the NFS client invokes the LOOKUP remote procedure call on the NFS server, passing as arguments the file handle of the directory and the path name's component.

On receiving the lookup request, the NFS server breaks apart the file handle for the directory and then invokes LOOKUP in the vnode interface, passing the path name's component as argument. If the directory is implemented by the server's local file system, the vnode layer invokes the procedure LOOKUP implemented by the server's local file system, passing the path name's component as argument. The local file system looks up the name, and if present, creates a vnode and returns the vnode to the NFS server. The NFS server sends a reply containing the vnode's file handle and some metadata for the vnode to the NFS client.

On receiving the reply, the NFS client creates a vnode, which contains the file handle, on the client computer and returns it to the file system call layer on the client machine. When the file system call layer has resolved the complete path name, it returns a file descriptor for the file to the user program.

To achieve usable performance, a typical NFS client maintains various caches. A client stores the vnode for every open file so that the client knows the file handles for open files. A client also caches recently-used vnodes, their attributes, recently-used blocks of those cached vnodes, and the mapping from name to vnode. Caching reduces the latency of file system operations on remote files, because for cached files a client can avoid the cost of remote procedure calls. In addition, because clients make fewer remote procedure calls, a single server can support more clients. If multiple clients cache the same file, however, the NFS protocol must ensure read/write coherence in some way.

4. Coherence

When programs share a local file in UNIX, the program calling READ observes the data from the most recent WRITE, even if this WRITE was performed by another program. This property is called read/write coherence (see page 2–11). If the programs are running on different clients, caching complicates implementing these semantics correctly.

To illustrate the problem consider a user program on one computer that writes a block of a file. The file system call layer on that computer performs the update to the block in the cache, delaying the write to the server, just like the local UNIX file system delays a write to disk. If soon later a program on another computer reads the file from the server, it may not observe the change made on the first computer, because that change may not have been propagated to the server yet.

Providing read/write coherence can be done at different levels of granularity. One option is to provide read/write coherence for files. That is, if an application OPENs a file, WRITEs, and CLOSEs the file on one client, and if later an application on a second client opens the same file, then the second application will observe the results of the writes by the first application. This option is called *close-to-open coherence.* Another option is to provide read/write coherence on the granularity of bytes of a file. That is, if two applications on different clients have the same file open concurrently, then a READ of one observes the results of WRITEs of the other.

Many NFS implementations choose to provide close-to-open coherence, because it is simpler to implement and good enough for many applications. Figure 4–13 illustrates close-to-open semantics in more detail. If, as in case 1, a program on one client calls WRITE and then CLOSE, and then, another client calls OPEN and READ, the NFS implementation will ensure that the READ will include the results of the WRITEs by the first client. But, as in case 2, if two clients have the same file open, one client writes a block of the file, and then the other client invokes READ, READ may return the data either from before or after the last WRITE; the NFS implementation make no guarantees in that case.

NFS implementations provide close-to-open semantics as follows. When a user program opens a file, the clients check with the server if the client has the most recent version of the file in its cache. If so, the client uses the version in its cache. If not, it removes its version from its cache. The client implements READ by returning the data from the cache, after fetching the block from the server if it is not in the cache. The client implements WRITE by modifying its

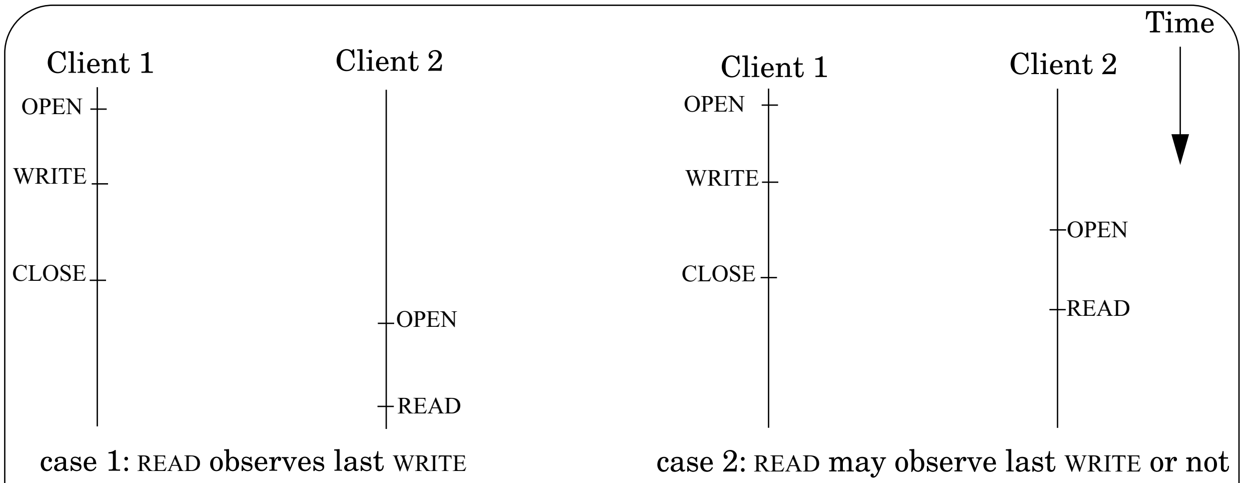

Figure 4–13: Two cases illustrating close-to-open coherence

local cached version, without incurring the overhead of remote procedure calls. When the user program invokes CLOSE on the file, CLOSE will send any modifications to that file to the server and wait until the server acknowledges that the modifications have been received.

This implementation is simple and provides decent performance. The client can perform WRITES at local memory speeds. By delaying sending the modified blocks until CLOSE, the client absorbs modifications that are overwritten (e.g., the application writes the same block multiple times) and aggregates writes to the same block (e.g., WRITES that modify different parts of the block).

By providing close-to-open semantics, most user programs written for a local UNIX file system will work correctly when their files are stored on NFS. For example, if a user edits a program on his personal workstation but prefers to compile on a faster compute machine, then NFS with close-to-open coherence works well, requiring no modifications to the editor and the compiler. After the editor has written out the modified file, and the users starts the compiler on the compute machine, the compiler will observe the edits.

On the other hand, certain programs will not work correctly using NFS implementations that provide close-to-open coherence. For example, a database program cannot store its database in a file over NFS, because, as the second case in figure 4–13 illustrates, close-to-open semantics doesn't specify the semantics of when client execute operations concurrently. If two clients run the database program, apply a transaction concurrently, and then exit the database program, only the results of one of the transactions may be visible. To make this even more clear, consider another scenario. Client 2 opens the database file before client 1 closes it and client 3 opens the database file after client 1 closes it. If client 2 and 3 then read data from the file, client 2 may not see the data written by client 1 while client 3 will see the data written by client 1.

To provide the correct semantics in this case requires more sophisticated protocols, which NFS implementations don't provide, because databases often have their own special-purpose solutions anyway, as we discuss chapters 9 and 10. If the database program doesn't provide a special-purpose solution, then tough luck, one cannot run it over NFS.

5. NFS version 3 and beyond

NFS version 2 is still widely used, but is slowly being replaced by NFS version 3. Version 3 addresses a number of limitations in version 2, but the extensions do not significantly change the preceding description; for example, version 3 supports 64-bit numbers for recording file sizes and adds an asynchronous write (i.e., the server may acknowledge an asynchronous WRITE request as soon as it receives the request, before it has written the data to disk).

NFS version 4 is a bigger change than version 3; in version 4 the server maintains some state. Version 4 also protects against intruders who can snoop and modify network traffic using techniques discussed in chapter 11. Furthermore, it provides a more efficient scheme for providing close-to-open coherence, and works well across the Internet, where the client and server may be connected using low-speed links.

The following references provide more details on NFS:

1. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and implementation of the Sun Network File System", *Proceedings of the 1985 Summer Usenix Technical Conference*, June 1985, El Cerrito, CA, pages 119–130.

2. Chet Juszezak, "Improving the performance and correctness of an NFS server", *Proceedings of the 1989 Winter Usenix Technical Conference*, January 1989, Berkeley, CA, pages 53–63.

3. Brian Pawlowski, Chet Juszezak, Peter Staubach, Carl Smith, Diana, Lebel, and David Hitz, "NFS Version 3 design and implementation", *Proceedings of the 1990 Summer Usenix Technical Conference*, June 1994, Boston, MA.

4. Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Turlow, "The NFS Version 4 protocol", *Proceedings of 2nd International SANE Conference*, May 2000, Maastricht, The Netherlands.