Network File System

This article describes the Network File System on SunOS.

The Network File System (NFS) allows a user to have transparent access to files on other systems on a network. The user accesses files on remote systems as though the files were on the local system. In other words, the network is transparent to the user.

Some of the NFS characteristics are as follows:

- Sharing Files

  Files on any machine can be accessed and shared by many other users on other machines in the network.

- Consistency

  Since the user community can now share access to the original copy of a file, the problem of maintaining consistency between multiple copies of that file is eliminated.

- Administration

  Only the central system must be updated to maintain common files. Otherwise, each system needs to be updated.

- Storage Savings

  There is a great savings in disk storage. Common files and programs can be placed on one system for the entire network—for example, man pages, compilers, and databases.

Problems

The network has to be very robust before NFS can be effectively used. If the network has a tendency to break down frequently or the performance is not acceptable, then NFS is not a good solution. When the network is down, users cannot access files that are on remote systems.

One of the biggest problems in networked systems is inconsistent files on multiple systems. You do not want to have multiple versions of databases, compilers, and other tools in a networked environment. NFS can solve this problem, but unless the networks are robust there must be multiple copies of files on multiple systems to avoid a single point of failure.

Solution

The solution is to have robust, high-performance networks. In addition, have important files mounted on multiple systems. This avoids the single system failure problem, although you may encounter consistency problems again.

NFS Utilities Overview

NFS comes with the following network utilities:

- Network Information Service (NIS)

  NIS is a lookup facility that allows you to maintain a set of consistent databases on all the hosts of a network. NIS was previously referred to as Yellow Pages.

- Portmapper (`portmap`)

  The portmapper is a network service that provides a client with a standard way of looking up the port number for any remote program available on a server. The port number is an integer that is mapped to a queue pointer.

- `mount`

  The `mount` function is used to convert local file path names into a unique filehandle that is used by the client in NFS file operations requests to the server. NFS filehandles are file system transparent.

- `lockd`

  The `lockd` function is used to implement file locking on NFS.

- RPC - Remote Procedure Call.

  This is the protocol used to provide a procedure-oriented interface to remote service procedures.

- XDR - External Data Representation.

  XDR translates machine-dependent data formats that can be used by all the network hosts using RPC/XDR into a data description language that provides machine-independent data formats. This enables two different hosts to exchange information.

Figure 1 shows the RPC-based protocols:

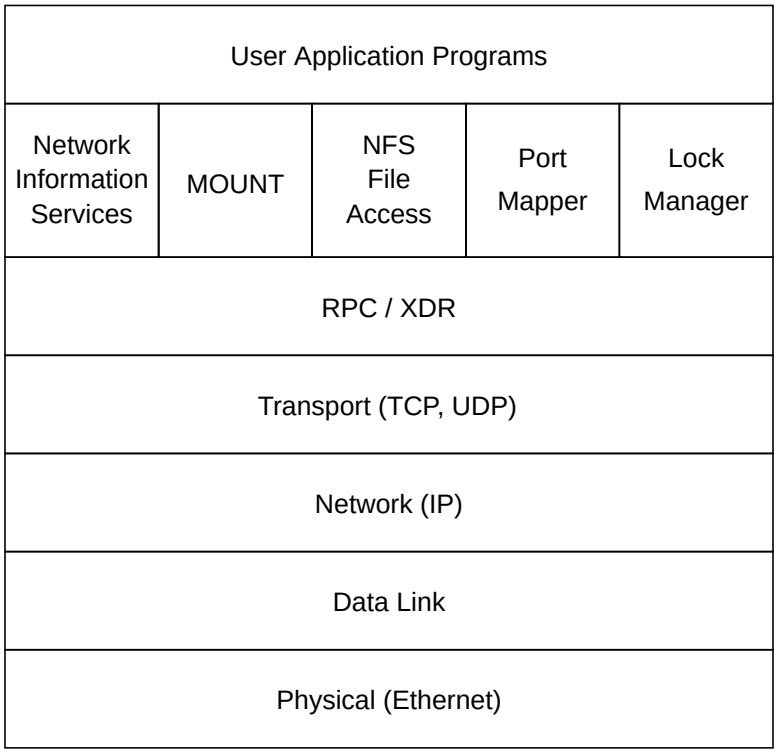


Figure 1 RPC-based Protocols

NFS Overview, Sun Implementation

In the Sun NFS implementation,[1] there are three entities to be considered: the operating system interface, the Virtual File System (VFS) interface, and the NFS interface.

1. *Networking Services*, Sun Microsystems, May 1988, p. 12.

Operating System Interface

The Unix operating system interface has been preserved in the Sun implementation of the NFS, thereby ensuring compatibility for existing applications.

Note – Remote devices are not supported over NFS.

Virtual File System Interface

A Virtual File System (VFS) means that multiple types of file systems on a single computer are accessed in the same way. VFS allows file systems to share the same user interface.

VFS is best seen as a layer that Sun has wrapped around the traditional Unix file system. This traditional file system is composed of directories and files, each of which has a corresponding *inode* (index node), containing administrative information about the file, such as location, size, ownership, permissions, and access times.

Inodes are assigned unique numbers within a file system, but a file on one file system can have the same number as a file on another file system. This is a problem in a network environment, because remote file systems need to be mounted dynamically, and numbering conflicts would cause wide-spread problems.

To solve this problem, Sun designed VFS, which is based on a data structure called *vnode*. In VFS, files are guaranteed to have unique numerical designators, even within a network. Vnodes cleanly separate file system operations from the semantics of their implementation.

Above the VFS interface, the operating system deals in vnodes; below this interface, the file system may or may not implement inodes. The VFS interface can connect the operating system to a variety of file systems (for example, 4.2 BSD, MS-DOS®, DEC-VMS®, IBM®, and so on). A local VFS connects to file system data on a local device.

Vnode Interface

A vnode is an abstraction of an inode that can support multiple types of file systems. The vnode support is transparent to the user. The operating system uses the mounting information to determine the type of each file system. Then, depending on the type of file system, a different set of procedures is executed to carry out file system operations.

Figure 2 shows the *application-to-vnode* interface:

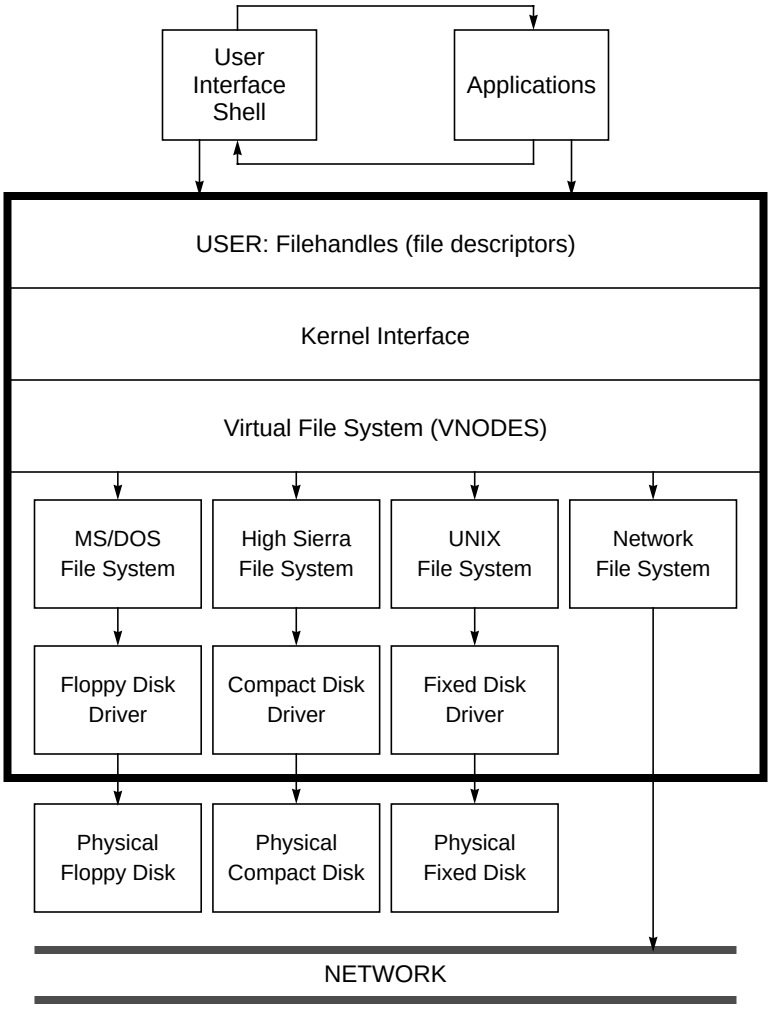


Figure 2 Vnode Interface

Vnode Layer Interconnection

Figure 3 shows the vnode layer interconnection. Only some of the data structure fields are shown.[2]

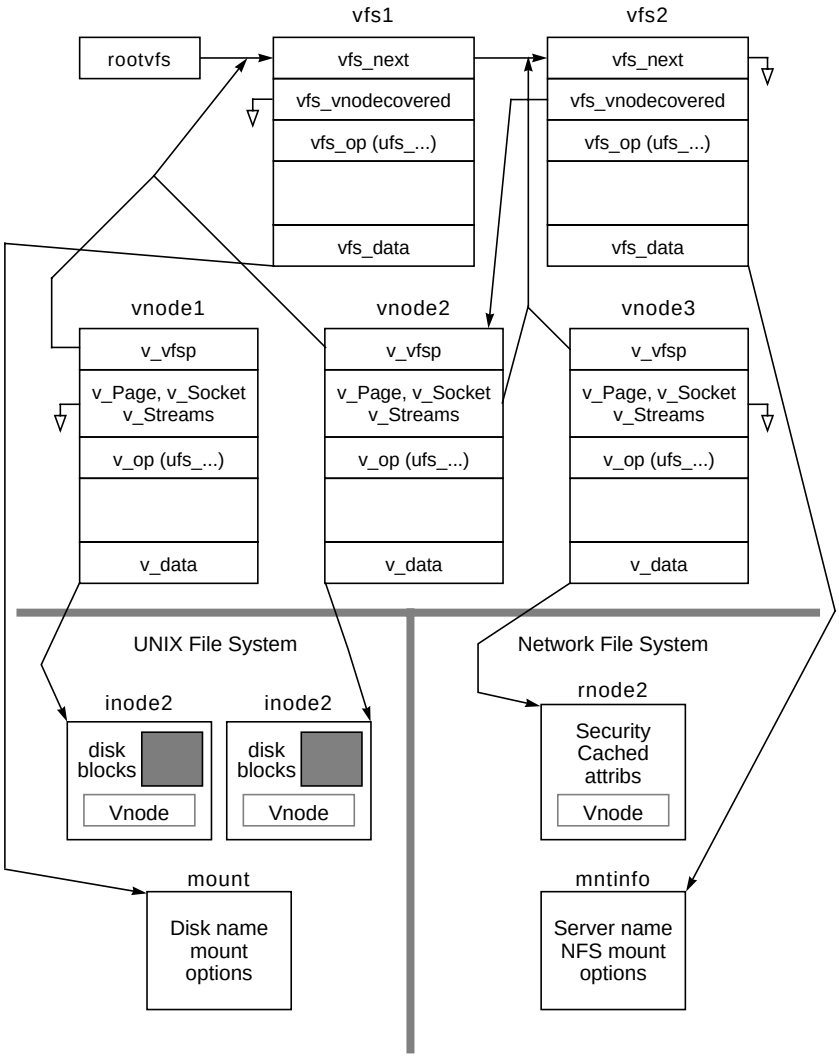


Figure 3 Vnode Layer Interconnection

2. Kleinman, S. R.: "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", Sun Microsystems.

Vnode Data Structures

Figure 4 shows the vnode data structures. Only some of the data fields in each of the data structures are shown. These data structures are based on the SunOS 4.1 release.

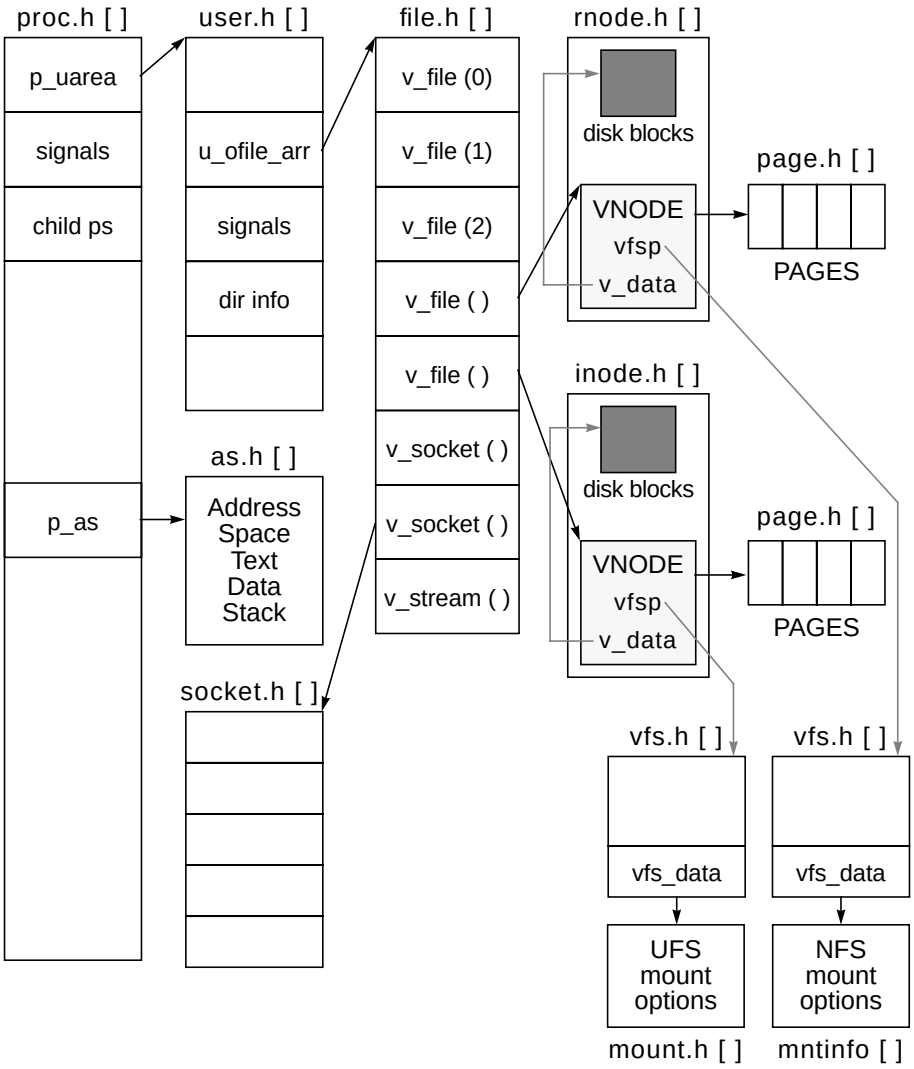


Figure 4 Vnode Data Structures

Sun NFS Interface

Figure 5 shows details of a NFS client-server protocol model. The client sends NFS requests to the server, which executes these requests on the local file and returns the results. For performance reasons, NFS and RPC/XDR are implemented in the SunOS kernel. This reduces the overhead of copying data out of kernel space into user space and then back into kernel space.

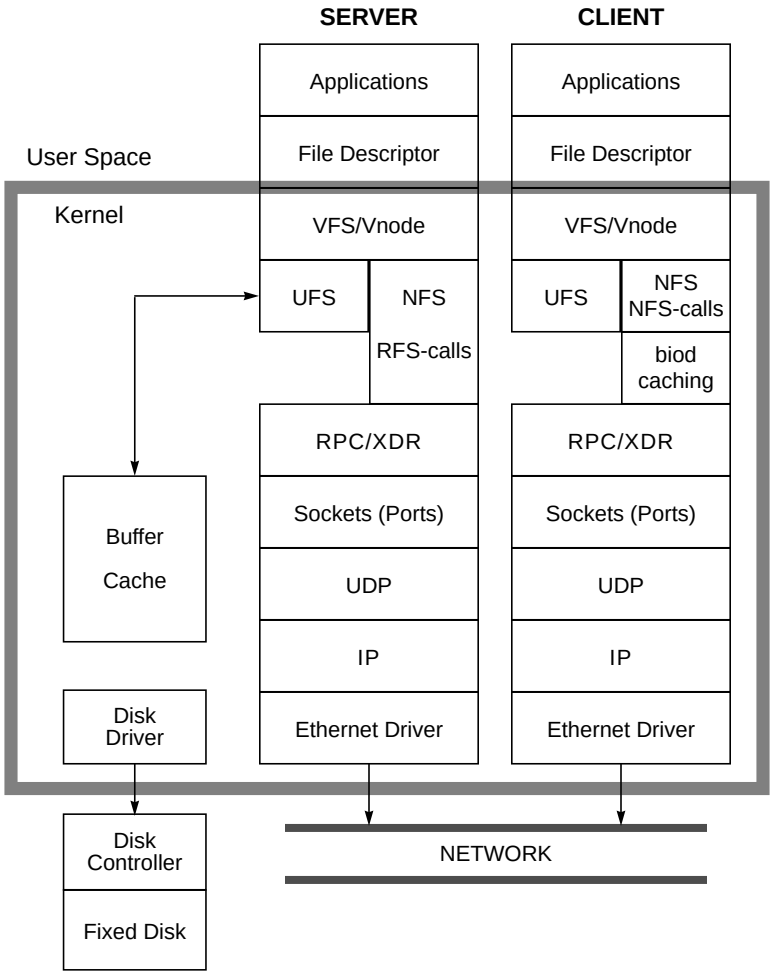


Figure 5 NFS Protocol Internals

Client-side NFS Protocol

Figure 6 shows example NFS client-side code:

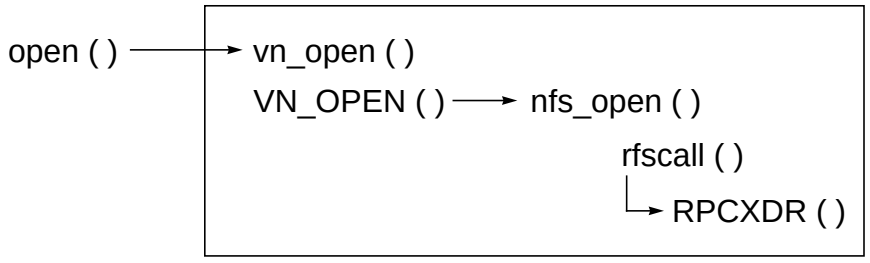


Figure 6 NFS Client-side Code

Each file system call is mapped to a virtual call, such as `vn_open` in the above example. The virtual call does some parameter checking, then calls a virtual macro. Depending on the type of file system, the virtual macro calls different file-system-specific routines to carry out the original system call. In Figure 6 above, the `VN_OPEN` macro calls `nfs_open`, which sends a request out to the server to open a file. Each virtual call is mapped into different file-system-specific calls.

The client-side NFS protocols are illustrated in Figure 7:

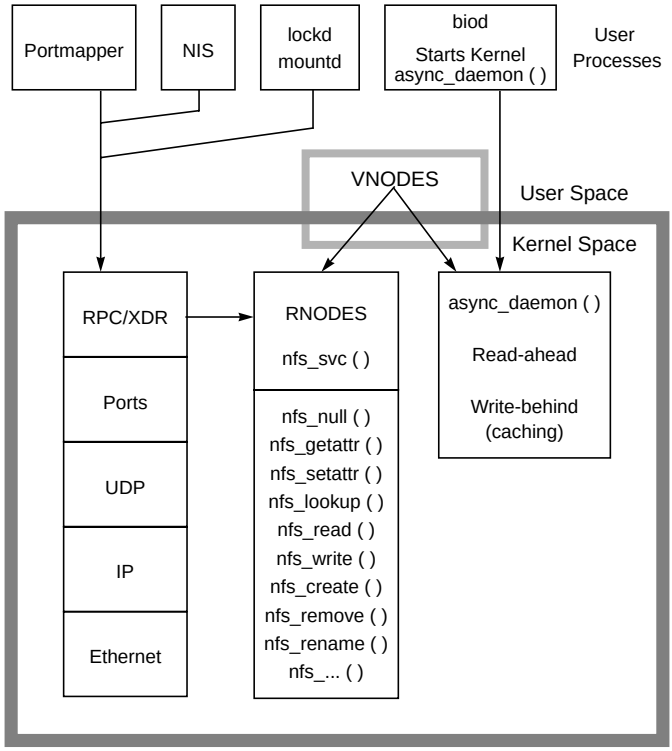


Figure 7 Client-side NFS Protocols

The caching-with-consistency checks (typically on the order of every 30 seconds) introduce a window where a client's modifications to a file are not noticed by other clients during the cache consistency check. (The cache consistency check consists of a `NFS_GETATTR` call to inspect the modified time of the file to see if the cached file data is still valid.) Unfortunately these consistency checks may differ from implementation to implementation. There is a trade-off here between performance and consistency. Actually, cache check intervals in our current NFS implementation differ for directories and plain files.

Directory timeouts are 30-60 seconds, while file timeouts are 3-60 seconds. The following values are devined In the NFSSRC 4.0 reference port:

`acregmin=`*n*

Hold cached attributes for at least *n* seconds after file modification.

`acregmax=`*n*

Hold cached attributes for no more than *n* seconds after file modification.

`acdirmin=`*n*

Hold cached attributes for at least *n* seconds after directory update.

`acdirmax=`*n*

Hold cached attributes for no more than *n* seconds after directory update.

`actimeo=`*n*

Set min and max times for regular files and directories to *n* seconds.

These values are bounded in the kernel between the above values. The reasoning in being able to set the file timeout lower than 30 seconds is to allow stronger (more frequent) consistency checks. In the SunOS 4.1 version of NFS, there is also an option to turn off caching on a mount point. Caching is important to performance.

NFS introduces a window of inconsistency defined by the timeout period. Other distributed file systems use callbacks to ensure data consistency. Another option to ensure consistency of single-writer / multiple-readers (or multiple-writers / multiple-readers) is to use locking, particularly in custom applications. This forces exact consistency and proper serialization of readers and writers.

Cache consistency checking on a more frequent basis loads both the network and the server with unnecessary checks in most circumstances; that is, having synchronized writers and readers is an infrequent case in most applications.

Many files accessed through NFS are read-only (shared executables). Locking is available in the case requiring synchronization and tight consistency.

For example, suppose Client 1 and Client 2 are accessing the same file from the server. Client 1 is writing and Client 2 is reading.

Client 1 writes new information into its local copy of the file. The new information is written into pages that are also written across the network to the server. This allows the server to update its copy of the same file. Only the updated pages are written to the server. After the server gets the NFS write request, it writes the data to the local disk. At this time, Client 2 has an older version of the same file. Client 2 runs `NFS_GETATTR` every so often to check if the cached files it has have been modified on the server. The new file attributes from the server are compared with the cached attributes; if they differ, then all the file pages in the client are invalidated.

This means that the next time Client 2 accesses a particular block of the file, the system generates a page fault. This causes a new page to be read from the server. Client 2 now has the most recent copy of the data.

But in all this, there is a window of inconsistency between the same file on Client 1 and Client 2.

Server-side NFS Protocol

On the server side, the NFS daemons are started and the server begins to listen for incoming NFS (RPC/XDR) requests. Every NFS file-system-type request has a filehandle in the request. The filehandle maps into the local file system. From this the virtual file system calls the appropriate local file system call to fulfill the client's NFS request and send a reply.

On the server side, the `nfsd` daemon starts a kernel-based NFS daemon. This daemon processes incoming NFS RPC requests. More than one daemon can be started.

Figure 8 displays NFS server-side code:

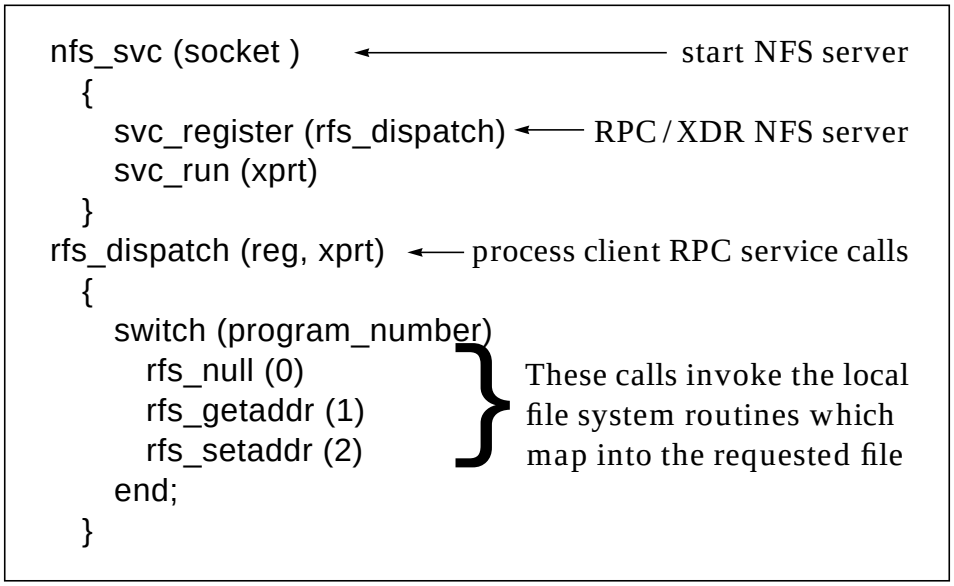


Figure 8 NFS Server-side Code

Figure 9 illustrates server-side NFS protocols:

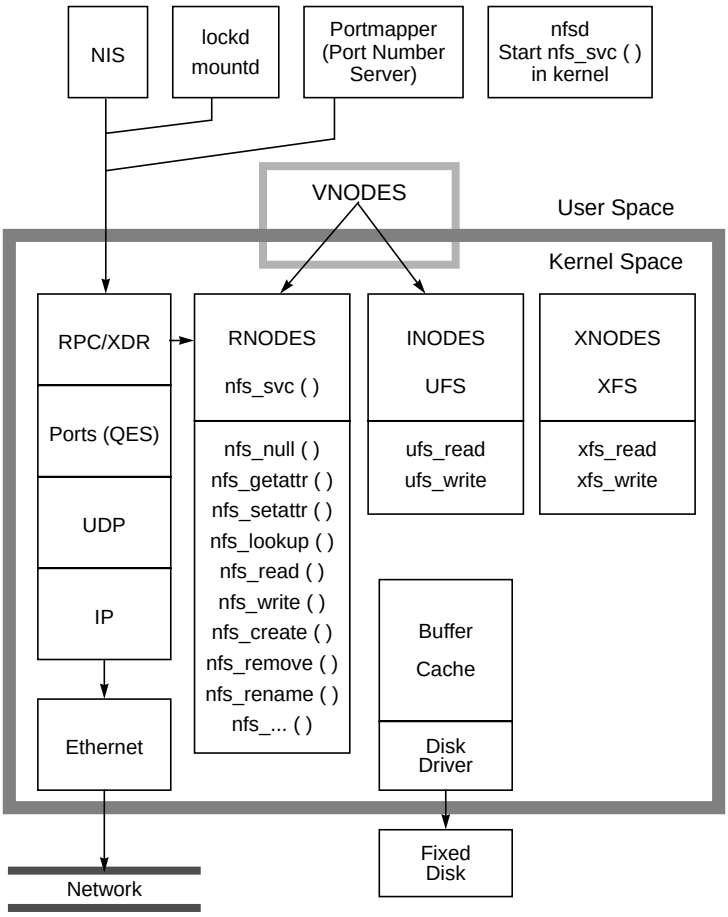


Figure 9 Server-side NFS Protocols

NFS Procedure Calls

In NFS Version 2, there are 15 procedure calls (see Table 1). There are eight NFS procedure calls that change data and should be only executed once by the server. The rest of the calls are inquiry-type calls that do not change data; these calls can be executed multiple times.

Table 1 NFS Procedure Calls

Name	Description	Type
`NULL`	do nothing	Inquiry
`GETATTR`	get file attributes	Inquiry
`SETATTR`	set file attributes	Data Modified
`LOOKUP`	lookup file name	Inquiry
`READLINK`	read symbolic link	Inquiry
`READ`	read from a file	Inquiry
`WRITE`	write to a file	Data Modified
`CREATE`	create a file	Data Modified
`REMOVE`	remove a file	Data Modified
`LINK`	create a link to a file	Data Modified
`SYMLINK`	create a symbolic link	Data Modified
`MKDIR`	make a directory	Data Modified
`RMDIR`	remove a directory	Data Modified
`READADDR`	read from a directory	Inquiry
`STATS`	get file system attributes	Inquiry

The server must complete each data modifying operation fully (synchronously) before responding to the client.

NFS Clients and Servers

NFS client systems range in size from small workstations to large multi-processor timesharing machines with hundreds of users.

A typical NFS client system repeatedly retransmits a request if it has not received a response from the server for that request within an interval of time (the default is .7 seconds). This interval is implementation-dependent and can also be specified as a parameter when the file system is mounted. The interval is increased using a back-off algorithm (interval = current interval x 2) up to a ceiling value (60 seconds) after each successive timeout. The level of impatience that a client has for a server is determined solely by the client and is not dynamically adjusted based on past server performance.

The number of retries within a timeout cycle is implementation-dependent and is also a mount-time parameter. The reference port sets the default number of retries in a timeout cycle to 3. With soft mount (a mount option), if a response is not received for a request within the first timeout cycle, then the client operation (system call) fails. With hard mounts, a single NFS request may span more than one timeout cycle before it receives a reply. The back-off algorithm described above continues to increase the timeout interval across timeout cycles.

The client Remote Procedure Call (RPC) layer assigns a transaction ID (`xid`) to each outgoing request. Duplicate requests within the same timeout cycle have the same `xid`. Duplicate requests have different `xid`s when the number of retransmissions exceeds the number of retries in a timeout cycle. Version 2 of the NFS protocol does not define an NFS transaction ID that is unique to each NFS request from a given client. This makes it impossible for a server to reliably determine whether two requests are actually duplicates. (It is not enough to know that two requests appear to be the same—for example, a write to the same location in the same file—because they could have been generated by two separate client processes in sequence.)

The client system can have multiple outstanding read and/or write requests. A client process blocks whenever a read or write request cannot be satisfied locally and must be processed by the server. When it blocks, another process can run; that process may also generate read or write requests. A single process can have multiple outstanding read and/or write requests if the client system is running NFS block I/O (`biod`) daemons. These daemons perform client read-ahead and write-behind functions asynchronously, allowing the client process to continue execution. Each outstanding request times out

individually; each can result in a retransmission. Clients discard duplicate responses (the second response received for a single request) as unsolicited input, but they are counted as a bad `xid`, and are available via `nfsstat` .

A typical *NFS server system* waits for work to appear on an incoming request queue. This queue is the socket buffer allocated for the NFS socket. Incoming requests are converted into a form understandable by the local file system routines that actually perform the work of getting the data to and from a disk. The incoming request queue is of a fixed size. If the queue fills (requests coming in faster than they can be processed), then some incoming requests might be dropped.

The amount of work that a server can perform is called *server bandwidth*. It is usually not limited by CPU speed, but by network interface and/or the disk subsystem performance. Server bandwidth is sometimes measured in a general manner (for example, NFS operations/second) and sometimes specifically (for example, read or write speed in Kbytes/second). A typical server does not prioritize incoming requests based on type of request or originating client.

Ignoring access rights and security, an NFS server has limited control over how it is used. An administrator decides:

- Whether to serve or not: a system serves by running `nfsd` and mount daemons.
- What to serve: a server allows operations only on exported file systems.
- Who to serve: the server's export access list controls which remote hosts can mount file systems.
- How many `nfsd` daemons to run: this controls the number of NFS requests that the server can work on concurrently. It also controls the amount of resources (for example, disk bandwidth) available for remote use, versus use by local processes.

The server depends on its clients to attenuate their request loads as it becomes heavily loaded (that is, the aggregate load is coming in faster than it can be processed). However, nothing makes a particular client implementation act kindly towards a server. There is no way to enforce the client back-off scheme described above. Most implementations allow client administrators (or workstation users) to run as many block I/O daemons as they wish, and to mount with retransmission timeout values that are very small. When faced

with one of the latest generation workstations armed with a suitable workload, the performance of even the most powerful server configurations can degrade drastically.

NFS Writes

Because the NFS server is bound by a stateless protocol, it must commit any modified data to stable storage before telling the client that the request is complete. If a server is not following this rule, then it is not living up to its part of the agreement implicit in the NFS crash recovery design. A synchronous operation carries with it the promise to fully complete that operation at some later time. The protocol contains no provisions for recalling past promises (which is precisely why crash recovery is so easy). Therefore, a server must complete each data modifying operation fully (synchronously) before responding.

For each remote write request, at least one and possibly two or three synchronous disk operations must be performed by the server before a response can be sent to the client indicating that the request has been completed. At the very least, the data block in question must be written. If the write increased the size of the file, or on-disk structures have changed (for example, adding a direct block to fill a hole in the file), then the block containing the inode must be written. Finally, if an indirect block was modified, then it too must be written before responding.

The reference port makes a special case for the file modify time in the inode. If the modify time is the only item changed in the inode as a result of a write operation (that is, a write to a previously allocated block), then the inode update to disk is performed asynchronously. This is one promise that the server may not keep; the risk is taken for the benefits of better performance.

Duplicate Requests

Duplicate requests (sometimes called delayed retransmissions) are part of the NFS crash recovery design. A server can receive a duplicate request while performing the original request. Multiple copies of a request can be received and placed on the input queue before any are processed. If a client is preparing to retransmit when the response it wanted is received, the server is still sent a duplicate request.[3]

3. Juszczak, Chet, “Improving the Performance and Correctness of an NFS Server”, USENIX, Winter 1989.

NFS Caching

Cache strategies are central to performance, reliability, and correctness of distributed file services. Caching improves performance by avoiding unnecessary disk traffic and server use. However, caching implies the potential existence of multiple copies of the same data, and keeping these multiple copies consistent is a challenge. This is especially true when the caches are kept by the clients of a distributed file service, which might be attempting concurrent access to the same file.

NFS adheres to a stateless-server model; the server maintains no state information between RPC requests. This simplifies the server implementation, avoids hard limits on the number of simultaneous clients, and makes server-crash recovery trivial.

However, because the server has no record of which clients are currently using a file, it cannot make guarantees about cache consistency. An NFS client periodically checks with the server to see if a file has been modified; if so, the client invalidates its cache for that file. Because the cache is invalid the next time the file is accessed for reading, the client sends a request to the sever for the latest copy of that file block. The interval between checks is a compromise between performance (frequent checking loads the server and delays the client) and consistency (insufficiently frequent checking may mean that a client uses stale data from its cache).

Because one NFS client has no way of identifying other clients that may be concurrently accessing a file, all of its consistency checks must be made with the file server. The file server, because it is stateless, also does not know which clients are caching a file. Therefore, whenever a client modifies a file, it must immediately communicate the change back to the server. In this way, it limits the potential inconsistency between the server's copy and its own cache to a short period. This "write-through" policy also limits the amount of damage caused by a crash; because an NFS server is required to write data onto disk (or other stable storage) before returning from the remote procedure call, the amount of cached information that is vulnerable to loss during a crash is quite limited.

The write-through policy has two distinct disadvantages. First, write-through limits the performance benefits of client-side caching, since a server disk access is done for every write. A surprising number of Unix files have short lifetimes and are never shared by multiple clients, and thus need not be kept anywhere but in the client-side cache of the host where they are created. Unfortunately,

NFS cannot distinguish between shared and unshared files, and so must treat every file as if it were potentially shared. Both client and server waste effort performing unnecessary write-through operations.

The second disadvantage of a strict write-through policy is that it forces applications to run synchronously with the disk. While an application is waiting for the data to make its way over the network, through the server queues, and onto the disk, it is blocked. The application therefore takes longer to complete than if disk writes were performed asynchronously, as they are in the local Unix file system. On single-user workstations especially, this time is wasted.

Actual NFS client implementations do not always write data synchronously. Instead, a block may be handed to a daemon process, which immediately writes it to the server; the original requesting process does not wait for the write to complete. This modification appears to be necessary to obtain reasonable performance. It loosens the consistency guarantee, however, so an NFS client synchronously finishes all pending write-throughs when a file is closed.[4]

NFS Notes

The following notes pertain to Sun Microsystems SunOS Version 4.0.3 or later.

`nfsd`

A command used to start a number of Unix daemons on the NFS server to handle client NFS requests. In most Unix systems, the NFS daemon is implemented in the kernel for performance reasons.

`biod`

A command that is used to start a number of asynchronous block I/O daemons on an NFS client. The `biod` daemons allow a client to make read-ahead and write-behind requests on the client to improve the overall performance of NFS.

4. Srinivasan, V., J. Mogul, "Spritely NFS: Implementation and Performance of Cache-Consistency Protocols", May 1989.

NFS provides for caching of disk blocks and asynchronous read-ahead by server and client to improve performance. `biod` starts a kernel-based daemon, `async_daemon()`.

All of the procedures in the NFS protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any modified data associated with the request is now on stable storage. For example, a client write request can cause the server to update data blocks, file system information blocks, and file attribute information (size and modify times). When the write operation completes, the client can assume that the write data is safe and discard it. This is a very important part of the statelessness of the server. If the server does not flush dirty data before returning to the client, the client has no way of knowing when it is safe to discard modified data.

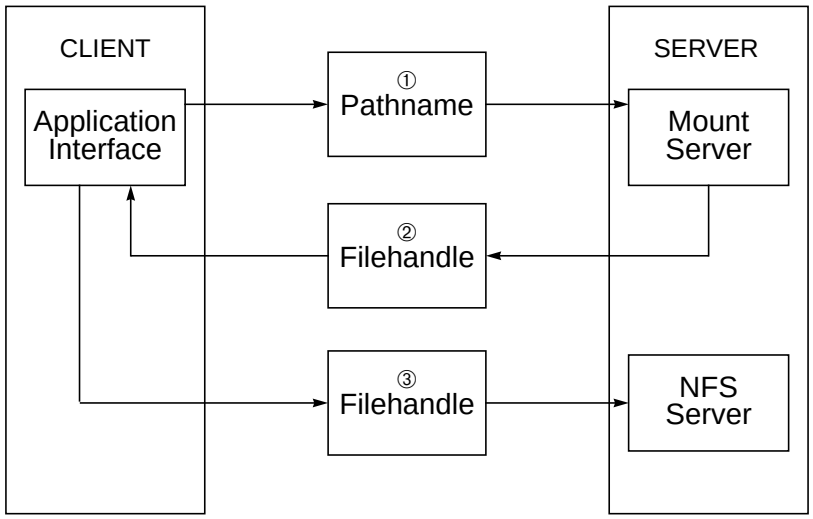
NFS Mount Daemon (`mountd`*)*

The NFS interface defines traditional file system operations for reading directories, creating and destroying files, reading and writing files, and reading and setting file attributes. The interface is designed so that file operations address files with an uninterrupted identifier called a *filehandle*, a starting byte address, and a length in bytes. NFS never deals with pathnames, only with filehandles. It gets these filehandles from the *mount daemon* (`mountd`).

Given a filehandle for a directory, a client program can use NFS procedures to get other filehandles, and thereby navigate throughout the directories and files of a file system. A client must, however, get its first filehandle for a file system by using an RPC call to the mount server. Then `mount` returns a filehandle that grants access to the file system.

When a Unix client boots, the `mount` program reads its `/etc/fstab` file. The `mount` program in turn requests the appropriate remote server or servers to provide access to the directories the client has specified in `/etc/fstab`. The mount daemon on the server receives the request and determines if the requested file system is available to the client. If it is, the mount daemon returns a filehandle to the client system's kernel. The filehandle is a piece of data that the client uses to identify the file system to the server whenever any further access is required. The filehandle is opaque to the client—that is, the client uses the filehandle without interpreting its contents. The filehandle is essentially an address that is understood only by the server.

Figure 10 shows the mounting and NFS filehandle sequence:



Legend: ① Client sends pathname to mount server
② Mount server returns corresponding filehandle
③ Client uses filehandle in NFS requests to the server

Figure 10 Mounting and NFS Filehandle Sequence

Table 2 shows mount RPC calls.

Table 2 Mount RPC Calls

Operation Name	**Description**
`NULL`	do nothing
`MOUNT`	return filehandle for pathname
`READMOUNT`	return mount list
`UNMOUNT`	remove mount list entry
`UMOUNTALL`	clear mount list
`READEXPORT`	return export list

Ports and the Portmapper

Communication transports such as TCP/IP provide a method of interprocess communication whereby messages are delivered across a network to a particular address. In the case of TCP/IP and many other protocols, this address identifies a unique host and a unique location on that host called a *port*. A port is a logical communications channel between one machine and another machine. By waiting for messages on a particular port, an application need not be concerned with how a particular message got to the machine. By knowing the individual port that an application is waiting for messages on, another can communicate with it.

Ports are useful in a heterogeneous sense as well. By simply knowing the port number to send a message to, the remote program need not know how the remote computer system's operating system identifies that particular application. Client programs need a way to find server programs; that is, they need a way to look up and find the port numbers of server programs. Network transport services do not provide such a service; they merely provide process-to-process message transfer across a network. A message typically contains a transport address that contains a network number, host number, and port number.

The *portmapper* (`portmap`) is a network service that provides a client with a standard way of looking up the port number for any service available on a remote server. The portmapper on any one particular machine maintains a database of port-to-program number mappings. Services on that host update the port map, and clients query the port map via the portmapper. To find the port for a particular network service, the client sends an RPC message to that server's portmapper. Since the portmapper is always listening at a well-known port number, the client need not solve the problem of how to find the portmapper. The portmapper returns the correct port for the program number in question.

Like all RPC services, the portmapper is implemented by a set of procedures that can be called over the network. The procedures are shown in Table 3:

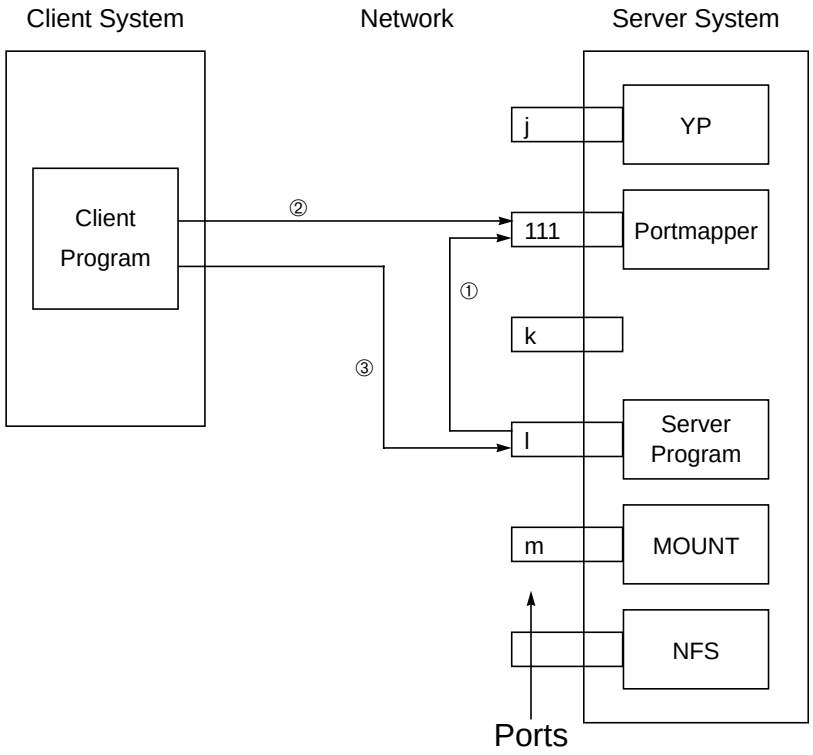
Table 3 Portmapper Procedures

#	Procedure
0	`null`
1	`set` (add entry)
2	`unset` (remove entry)
3	`getport`
4	`dump` (get all entries)
5	`callit` (call the remote procedure)

It is possible to do an indirect RPC through the portmapper using `callit`. To improve efficiency, the reply to the indirect call is made directly to the client and not through the portmapper. Within the standard RPC library are calls that provide access to all of the portmapper routines.

Portmapper functionality is implemented statically in the Internet protocols. In the Internet protocol, TCP/IP, there is an `/etc/services` file that contains the port-to-service mapping.

Figure 11 illustrates portmapper sequence of calls:



Legend: ① Server program registers with portmapper and gets a port number
② Client gets server's port number from the server portmapper
③ Client calls server program

Figure 11 Portmapper Sequence of Calls

Every instance of a remote program can be mapped to a different port on every server. However, the portmapper procedure can be used to broadcast a remote procedure call indirectly, since all portmappers are associated with port number 111.

Current RPC Services

Table 4 below shows a sample list of RPC program numbers, version numbers, and port numbers. The port numbers are dynamically assigned, which means they can be different from system to system.

Note – Server programs based on Sun's RPC library typically get port numbers assigned at run-time by calling an RPC library procedure.

Table 4 RPC Port Assignments

Program	Ver	Protocol	Port	ServiceName
100000	2	tcp	111	portmap
100007	1	udp	1027	ypbind
100029	1	udp	657	keyserv
100003	2	udp	2049	ufs
100005	2	tcp	704	mountd
100024	1	tcp	711	status
100021	1	tcp	712	nlockmgr
100020	1	tcp	1036	llockmgr
100020	1	tcp	721	llockmgr
100021	2	tcp	724	nlockmgr
100021	2	tcp	1037	nlockmgr
100011	1	udp	1039	rquotad
100001	2	udp	1040	rstatd
100001	3	udp	1040	rstatd
100001	4	udp	1040	rstatd
100002	1	udp	1041	rusersd
100002	2	udp	1041	rusersd
100012	1	udp	1042	sprayd
100008	1	udp	1043	walld
100015	6	udp	1045	selection_svc
100019	1	udp	1072	sched