The Sun Network Filesystem: Design, Implementation and Experience

Russel Sandberg

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA. 94043
(415) 960–7293

Introduction

The Sun Network Filesystem (NFS™) provides transparent, remote access to filesystems. Unlike many other remote filesystem implementations under UNIX®, NFS is designed to be easily portable to other operating systems and machine architectures. It uses an External Data Representation (XDR) specification to describe protocols in a machine and system independent way. NFS is implemented on top of a Remote Procedure Call package (RPC) to help simplify protocol definition, implementation, and maintenance.

In order to build NFS into the UNIX kernel in a way that is transparent to applications, we decided to add a new interface to the kernel which separates generic filesystem operations from specific filesystem implementations. The "filesystem interface" consists of two parts: the Virtual File System (VFS) interface defines the operations that can be done on a filesystem, while the virtual node (vnode) interface defines the operations that can be done on a file within that filesystem. This new interface allows us to implement and install new filesystems in much the same way as new device drivers are added to the kernel.

In this paper we discuss the design and implementation of the filesystem interface in the UNIX kernel and the NFS virtual filesystem. We compare NFS to other remote filesystem implementations, and describe some interesting NFS ports that have been done, including the IBM PC implementation under MS/DOS and the VMS server implementation. We also describe the user-level NFS server implementation which allows simple server ports without modification to the underlying operating system. We conclude with some ideas for future enhancements.

In this paper we use the term *server* to refer to a machine that provides resources to the network; a *client* is a machine that accesses resources over the network; a *user* is a person "logged in" at a client; an *application* is a program that executes on a client; and a *workstation* is a client machine that typically supports one user at a time.

Design Goals

NFS was designed to simplify the sharing of filesystem resources in a network of non-homogeneous machines. Our goal was to provide a way of making remote files available to local programs without having to modify, or even relink, those programs. In addition, we wanted remote file access to be comparable in speed to local file access.

The overall design goals of NFS were:

Machine and Operating System Independence
: The protocols used should be independent of UNIX so that an NFS server can supply files to many different types of clients. The protocols should also be simple enough that they can be implemented on low-end machines like the PC.

Crash Recovery
: When clients can mount remote filesystems from many different servers it is very important that clients and servers be able to recover easily from machine crashes and network problems.

Transparent Access
: We want to provide a system which allows programs to access remote files in exactly the same way as local files, without special pathname parsing, libraries, or recompiling. Programs should not need or be able to tell whether a file is remote or local.

UNIX Semantics Maintained on UNIX Client

In order for transparent access to work on UNIX machines, UNIX filesystem semantics have to be maintained for remote files.

Reasonable Performance

People will not use a remote filesystem if it is no faster than the existing networking utilities, such as *rcp*, even if it is easier to use. Our design goal was to make NFS as fast as a small local disk on a SCSI interface.

Basic Design

The NFS design consists of three major pieces: the protocol, the server side and the client side.

NFS Protocol

The NFS protocol uses the Sun Remote Procedure Call (RPC) mechanism [1]. For the same reasons that procedure calls simplify programs, RPC helps simplify the definition, organization, and implementation of remote services. The NFS protocol is defined in terms of a set of procedures, their arguments and results, and their effects. Remote procedure calls are synchronous, that is, the client application blocks until the server has completed the call and returned the results. This makes RPC very easy to use and understand because it behaves like a local procedure call.

NFS uses a stateless protocol. The parameters to each procedure call contain all of the information necessary to complete the call, and the server does not keep track of any past requests. This makes crash recovery very easy; when a server crashes, the client resends NFS requests until a response is received, and the server does no crash recovery at all. When a client crashes, no recovery is necessary for either the client or the server.

If state is maintained on the server, on the other hand, recovery is much harder. Both client and server need to reliably detect crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can rebuild the server’s state.

A stateless protocol avoids complex crash recovery. If a client just resends requests until a response is received, data will never be lost due to a server crash. In fact, the client cannot tell the difference between a server that has crashed and recovered, and a server that is slow.

Sun’s RPC package is designed to be transport independent. New transport protocols, such as ISO and XNS, can be “plugged in” to the RPC implementation without affecting the higher level protocol code (see appendix 3). NFS currently uses the DARPA User Datagram Protocol (UDP) and Internet Protocol (IP) for its transport level. Since UDP is an unreliable datagram protocol, packets can get lost, but because the NFS protocol is stateless and NFS requests are idempotent, the client can recover by retrying the call until the packet gets through.

The most common NFS procedure parameter is a structure called a file handle (fhandle or fh) which is provided by the server and used by the client to reference a file. The fhandle is opaque, that is, the client never looks at the contents of the fhandle, but uses it when operations are done on that file.

An outline of the NFS protocol procedures is given below. For the complete specification see the *Sun Network Filesystem Protocol Specification* [2].

null() returns ()

Do nothing procedure to ping the server and measure round trip time.

lookup(dirfh, name) returns (fh, attr)

Returns a new fhandle and attributes for the named file in a directory.

create(dirfh, name, attr) returns (newfh, attr)

Creates a new file and returns its fhandle and attributes.

remove(dirfh, name) returns (status)

Removes a file from a directory.

getattr(fh) returns (attr)

Returns file attributes. This procedure is like a stat call.

setattr(fh, attr) returns (attr)

Sets the mode, uid, gid, size, access time, and modify time of a file. Setting the size to zero truncates the file.

read(fh, offset, count) returns (attr, data)

Returns up to *count* bytes of data from a file starting *offset* bytes into the file. **read** also returns the attributes of the file.

write(fh, offset, count, data) returns (attr)

Writes *count* bytes of data to a file beginning *offset* bytes from the beginning of the file. Returns the attributes of the file after the **write** takes place.

rename(dirfh, name, tofh, toname) returns (status)

Renames the file *name* in the directory *dirfh*, to *toname* in the directory *tofh*.

link(dirfh, name, tofh, toname) returns (status)

Creates the file *toname* in the directory *tofh*, which is a link to the file *name* in the directory *dirfh*.

symlink(dirfh, name, string) returns (status)

Creates a symbolic link *name* in the directory *dirfh* with value *string*. The server does not interpret the *string* argument in any way, just saves it and makes an association to the new symbolic link file.

readlink(fh) returns (string)

Returns the string which is associated with the symbolic link file.

mkdir(dirfh, name, attr) returns (fh, newattr)

Creates a new directory *name* in the directory *dirfh* and returns the new fhandle and attributes.

rmdir(dirfh, name) returns(status)

Removes the empty directory *name* from the parent directory *dirfh*.

readdir(dirfh, cookie, count) returns(entries)

Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, file id, and an opaque pointer to the next directory entry called a *cookie*. The *cookie* is used in subsequent **readdir** calls to start reading at a specific entry in the directory. A **readdir** call with the *cookie* of zero returns entries starting with the first entry in the directory.

statfs(fh) returns (fsstats)

Returns filesystem information such as block size, number of free blocks, etc.

New fhandles are returned by the **lookup**, **create**, and **mkdir** procedures which also take an fhandle as an argument. The first remote fhandle, for the root of a filesystem, is obtained by the client using the RPC based MOUNT protocol. The MOUNT protocol takes a directory pathname and returns an fhandle if the client has access permission to the filesystem which contains that directory. The reason for making this a separate protocol is that this makes it easier to plug in new filesystem access checking methods, and it separates out the operating system dependent aspects of the protocol. Note that the MOUNT protocol is the only place that UNIX pathnames are passed to the server. In other operating system implementations the MOUNT protocol can be replaced without having to change the NFS protocol.

The NFS protocol and RPC are built on top of the Sun External Data Representation (XDR) specification [3]. XDR defines the size, byte order and alignment of basic data types such as string, integer, union, boolean and array. Complex structures can be built from the basic XDR data types. Using XDR not only makes protocols machine and language independent, it also makes them easy to define. The arguments and results of RPC procedures are defined using an XDR data definition language that looks a lot like C declarations. This data definition language can be used as input to an XDR protocol compiler which produces the structures and XDR translation procedures used to interpret RPC protocols [11].

Server Side

Because the NFS server is stateless, when servicing an NFS request it must commit any modified data to stable storage before returning results. The implication for UNIX based servers is that requests which modify the filesystem must flush all modified data to disk before returning from the call. For example, on a **write** request, not only the data block, but also any modified indirect blocks and the block containing the inode must be flushed if they have been modified.

Another modification to UNIX necessary for our server implimentation is the addition of a generation number in the inode, and a filesystem id in the superblock. These extra numbers make it possible for the server to use the inode number, inode generation number, and filesystem id together as the fhandle for a file. The inode generation number is necessary because the server may hand out an fhandle with an inode number of a file that is later removed and the inode reused. When the original fhandle comes back, the server must be able to tell that this inode number now refers to a different file. The generation number has to be incremented every time the inode is freed.

Client Side

The Sun implementation of the client side provides an interface to NFS which is transparent to applications. To make transparent access to remote files work we had to use a method of locating remote files that does not change the structure of path names. Some UNIX based remote file access methods use pathnames like *host:path* or */../host/path* to name remote files. This does not allow real transparent access since existing programs that parse pathnames have to be modified.

Rather than doing a "late binding" of file address, we decided to do the hostname lookup and file address binding once per filesystem by allowing the client to attach a remote filesystem to a directory with the *mount* command. This method has the advantage that the client only has to deal with hostnames once, at mount time. It also allows the server to limit access to filesystems by checking client credentials. The disadvantage is that remote files are not available to the client until a mount is done.

Transparent access to different types of filesystems mounted on a single machine is provided by a new filesystem interface in the kernel [13]. Each "filesystem type" supports two sets of operations: the Virtual Filesystem (VFS) interface defines the procedures that operate on the filesystem as a whole; and the Virtual Node (vnode) interface defines the procedures that operate on an individual file within that filesystem type. Figure 1 is a schematic diagram of the filesystem interface and how NFS uses it.

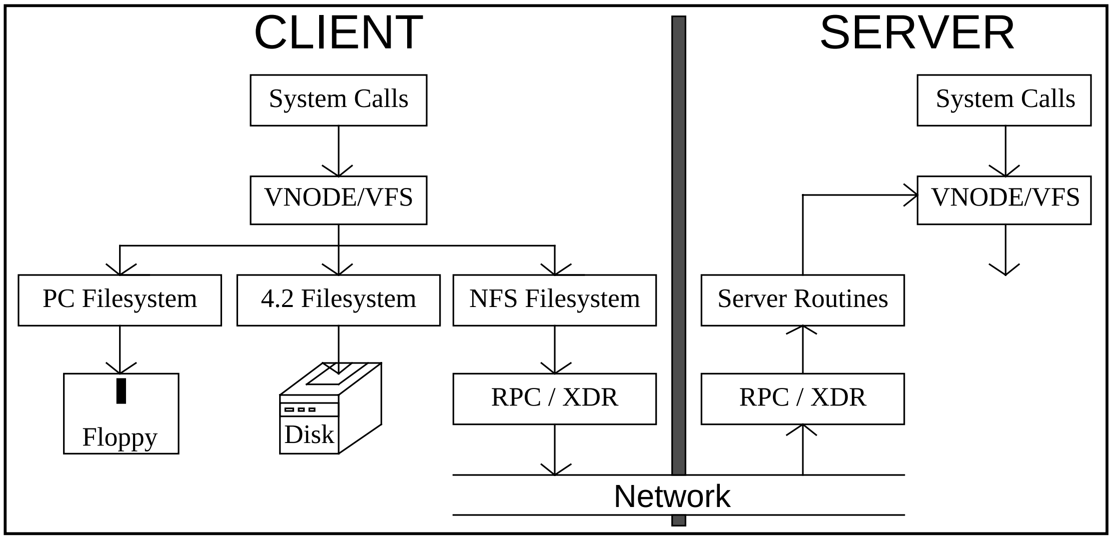

Figure 1

The Filesystem Interface

The VFS interface is implemented using a structure that contains the operations that can be done on a filesystem. Likewise, the vnode interface is a structure that contains the operations that can be done on a node (file or directory) within a filesystem. There is one VFS structure per mounted filesystem in the kernel and one vnode structure for each active node. Using this abstract data type implementation allows the kernel to treat all filesystems and nodes in the same way without knowing which underlying filesystem implementation it is using.

Each vnode contains a pointer to its parent VFS and a pointer to a mounted-on VFS. This means that any node in a filesystem tree can be a mount point for another filesystem. A **root** operation is provided in the VFS to return the root vnode of a mounted filesystem. This is used by the pathname traversal routines in the kernel to bridge mount points. The **root** operation is used instead of keeping a pointer so that the root vnode for each mounted filesystem can be released. The VFS of a mounted filesystem also contains a pointer back to the vnode on which it is mounted so that pathnames that include ".." can also be traversed across mount points.

In addition to the VFS and vnode operations, each filesystem type must provide **mount** and **mount_root** operations to mount normal and root filesystems. The operations defined for the filesystem interface are given below. In the arguments and results, vp is a pointer to a vnode, dvp is a pointer to a directory vnode and devvp is a pointer to a device vnode.

Filesystem Operations

mount(varies) System call to mount filesystem

mount_root( )	Mount filesystem as root
VFS Operations	
unmount(vfs)	Unmount filesystem
root(vfs) returns(vnode)	Return the vnode of the filesystem root
statfs(vfs) returns(statfsbuf)	Return filesystem statistics
sync(vfs)	Flush delayed write blocks
Vnode Operations	
open(vp, flags)	Mark file open
close(vp, flags)	Mark file closed
rdwr(vp, uio, rwflag, flags)	Read or write a file
ioctl(vp, cmd, data, rwflag)	Do I/O control operation
select(vp, rwflag)	Do select
getattr(vp) returns(attr)	Return file attributes
setattr(vp, attr)	Set file attributes
access(vp, mode)	Check access permission
lookup(dvp, name) returns(vp)	Look up file name in a directory
create(dvp, name, attr, excl, mode) returns(vp)	Create a file
remove(dvp, name)	Remove a file name from a directory
link(vp, todvp, toname)	Link to a file
rename(dvp, name, todvp, toname)	Rename a file
mkdir(dvp, name, attr) returns(dvp)	Create a directory
rmdir(dvp, name)	Remove a directory
readdir(dvp) returns(entries)	Read directory entries
symlink(dvp, name, attr, toname)	Create a symbolic link
readlink(vp) returns(data)	Read the value of a symbolic link
fsync(vp)	Flush dirty blocks of a file
inactive(vp)	Mark vnode inactive and do clean up
bmap(vp, blk) returns(devp, mappedblk)	Map block number
strategy(bp)	Read and write filesystem blocks
bread(vp, blockno) returns(buf)	Read a block
brelse(vp, bp)	Release a block buffer

Notice that many of the vnode procedures map one-to-one with NFS protocol procedures, while other, UNIX dependent procedures such as **open**, **close**, and **ioctl** do not. The **bmap**, **strategy**, **bread**, and **brelse** procedures are used to do reading and writing using the buffer cache.

Pathname traversal is done in the kernel by breaking the path into directory components and doing a **lookup** call through the vnode for each component. At first glance it seems like a waste of time to pass only one component with each call instead of passing the whole path and receiving back a target vnode. The main reason for this is that any component of the path could be a mount point for another filesystem, and the mount information is kept above the vnode implementation level. In the NFS filesystem, passing whole pathnames would force the server to keep track of all of the mount points of its clients in order to determine where to break the pathname and this would violate server statelessness. The inefficiency of looking up one component at a time can be alleviated with a cache of directory vnodes.

Implementation

Implementation of NFS started in March 1984. The first step in the implementation was modification of the 4.2 kernel to include the filesystem interface. By June we had the first “vnode kernel” running. We did some benchmarks to test the amount of overhead added by the extra interface. It turned out that in most cases the difference was not measurable, and in the worst case the kernel had only slowed down by about 2%. Most of the work in adding the new interface was in finding and fixing all of the places in the kernel that used inodes directly, and code that contained implicit knowledge of inodes or disk layout.

Only a few of the filesystem routines in the kernel had to be completely rewritten to use vnodes. *Namei*, the routine that does pathname lookup, was changed to use the vnode **lookup** operation, and cleaned up so that it doesn’t use global state. The *direnter* routine, which adds new directory entries (used by **create**, **rename**, etc.), was fixed because it depended on the global state from *namei*. *Direnter* was also modified

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE