Design and Implementation of the Sun Network Filesystem

Russel Sandberg
David Goldberg
Steve Kleiman
Dan Walsh
Bob Lyon

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA. 94110
(415) 960-7293

Introduction

The Sun Network Filesystem (NFS) provides transparent, remote access to filesystems. Unlike many other remote filesystem implementations under UNIX†, the 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. The NFS is implemented on top of a Remote Procedure Call package (RPC) to help simplify protocol definition, implementation, and maintenance.

In order to build the NFS into the UNIX 4.2 kernel in a user transparent way, 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 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 kernel and the NFS virtual filesystem. We describe some interesting design issues and how they were resolved, and point out some of the shortcomings of the current implementation. We conclude with some ideas for future enhancements.

Design Goals

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

The overall design goals of the 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 be able to recover easily from server crashes.

Transparent Access
: We want to provide a system which allows programs to access remote files in exactly the same way as local files. No pathname parsing, no special libraries, no recompiling. Programs should not be able to tell whether a file is remote or local.

---

† UNIX is a trademark of Bell Laboratories.

UNIX Semantics Maintained on 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 want to use the NFS if it is no faster than the existing networking utilities, such as *rcp*, even if it is easier to use. Our design goal is to make NFS as fast as the Sun Network Disk protocol (ND[1]), or about 80% as fast as a local disk.

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 help 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 blocks until the server has completed the call and returned the results. This makes RPC very easy to use since it behaves like a local procedure call.

The 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. When state is maintained on the server, on the other hand, recovery is much harder. Both client and server need to be able 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.

Using a stateless protocol allows us to avoid complex crash recovery and simplifies the protocol. 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 can not tell the difference between a server that has crashed and recovered, and a server that is slow.

Sun's remote procedure call package is designed to be transport independent. New transport protocols can be "plugged in" to the RPC implementation without affecting the higher level protocol code. The NFS uses the ARPA 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 the 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.

[1] ND, the Sun Network Disk Protocol, provides block-level access to remote, sub-partitioned disks.

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 another RPC based 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 an External Data Representation (XDR) specification [3]. XDR defines the size, bytes order and alignment of basic data types such as string, integer, union, boolean and array. Complex structures can be built from the basic 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.

Server Side

Because the NFS server is stateless, as mentioned above, 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. This means that, 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 to make the server work 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 client side provides the transparent interface to the NFS. 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 schemes use *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 using the *mount* program. 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 filesystems interface in the kernel. 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 the NFS uses it.

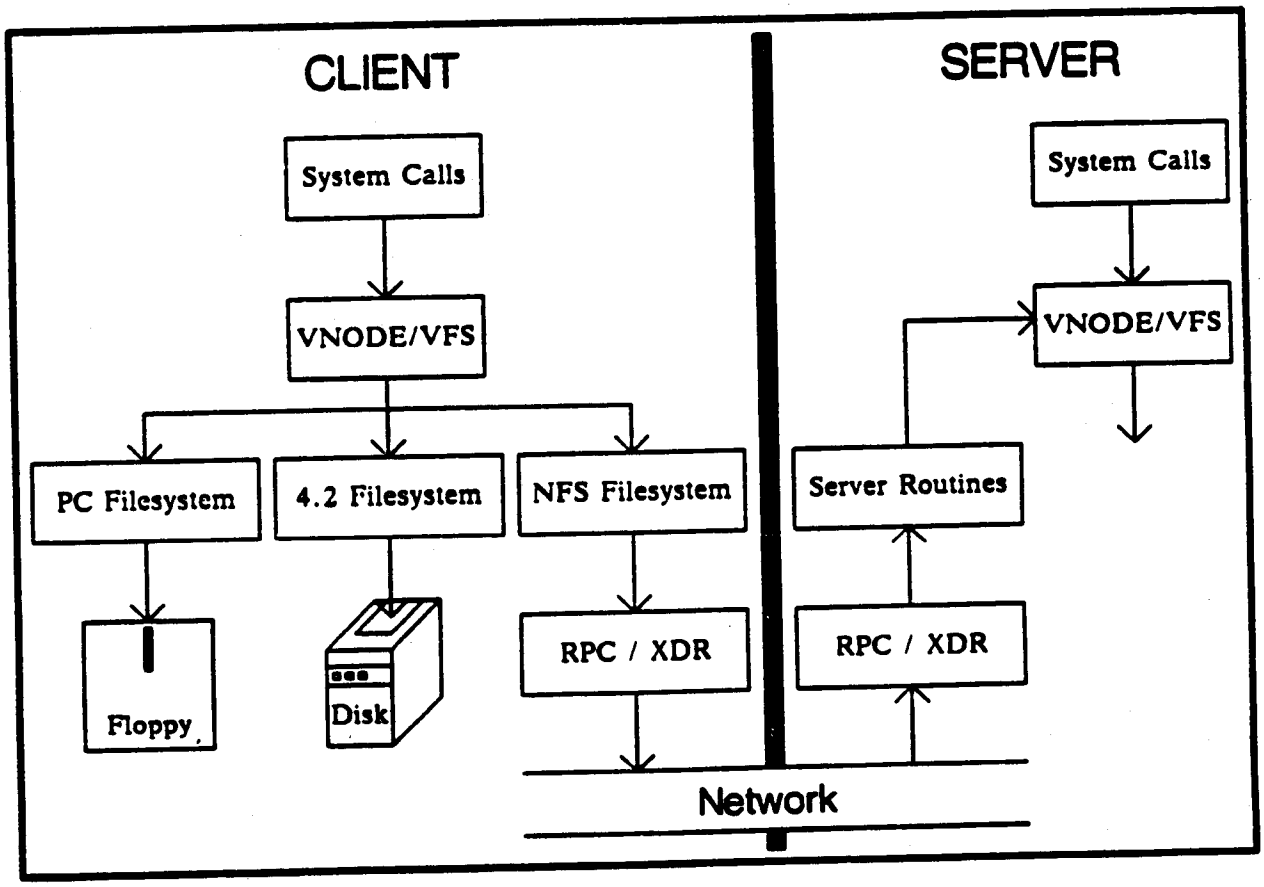

Figure 1

The Filesystem Interface

The VFS interface is implemented using a structure that contains the operations that can be done on a whole 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 just keeping a pointer so that the root vnode for each mounted filesystem can be released. The VFS of a mounted filesystem also contains a back pointer 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:

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(fsstatbuf)	Return filesystem statistics
sync(vfs)	Flush delayed write blocks

Vnode Operations

open(vnode, flags)	Mark file open
close(vnode, flags)	Mark file closed
rdwr(vnode, uio, rwflag, flags)	Read or write a file
ioctl(vnode, cmd, data, rwflag)	Do I/O control operation
select(vnode, rwflag)	Do select
getattr(vnode) returns(attr)	Return file attributes
setattr(vnode, attr)	Set file attributes
access(vnode, mode)	Check access permission
lookup(dvnode, name) returns(vnode)	Look up file name in a directory
create(dvnode, name, attr, excl, mode) returns(vnode)	Create a file
remove(dvnode, name)	Remove a file name from a directory
link(vnode, todvnode, toname)	Link to a file
rename(dvnode, name, todvnode, toname)	Rename a file
mkdir(dvnode, name, attr) returns(dvnode)	Create a directory
rmdir(dvnode, name)	Remove a directory
readdir(dvnode) returns(entries)	Read directory entries
symlink(dvnode, name, attr, to_name)	Create a symbolic link
readlink(vp) returns(data)	Read the value of a symbolic link
fsync(vnode)	Flush dirty blocks of a file
inactive(vnode)	Mark vnode inactive and do clean up
bmap(vnode, blk) returns(devnode, mappedblk)	Map block number
strategy(bp)	Read and write filesystem blocks
bread(vnode, blockno) returns(buf)	Read a block
brelse(vnode, buf)	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 is alleviated with a cache of directory vnodes.

Implementation

Implementation of the 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.), also had to be fixed because it depended on the global state from *namei*. *Direnter* also had to be modified to do directory locking during directory rename operations because inode locking is no longer available at this level, and vnodes are never locked.

To avoid having a fixed upper limit on the number of active vnode and VFS structures we added a memory allocator to the kernel so that these and other structures can be allocated and freed dynamically.

A new system call, *getdirentries*, was added to read directory entries from different types of filesystems. The 4.2 *readdir* library routine was modified to use the new system call so programs would not have to be rewritten. This change does, however, mean that programs that use *readdir* have to be relinked.

Beginning in March, the user level RPC and XDR libraries were ported to the kernel and we were able to make kernel to user and kernel to kernel RPC calls in June. We worked on RPC performance for about a month until the round trip time for a kernel to kernel **null** RPC call was 8.8 milliseconds. The performance tuning included several speed ups to the UDP and IP code in the kernel.

Once RPC and the vnode kernel were in place the implementation of NFS was simply a matter of writing the XDR routines to do the NFS protocol, implementing an RPC server for the NFS procedures in the kernel, and implementing a filesystem interface which translates vnode operations into NFS remote procedure calls. The first NFS kernel was up and running in mid August. At this point we had to make some modifications to the vnode interface to allow the NFS server to do synchronous **write** operations. This was necessary since unwritten blocks in the server's buffer cache are part of the "client's state".

Our first implementation of the MOUNT protocol was built into the NFS protocol. It wasn't until later that we broke the MOUNT protocol into a separate, user level RPC service. The MOUNT server is a user level daemon that is started automatically when a mount request comes in. It checks the file **/etc/exports** which contains a list of exported filesystems and the clients that can import them. If the client has import permission, the mount daemon does a **getfh** system call to convert a pathname into an fhandle which is returned to the client.

On the client side, the mount command was modified to take additional arguments including a filesystem type and options string. The filesystem type allows one *mount* command to mount any type of filesystem. The options string is used to pass optional flags to the different filesystem **mount** system calls. For example, the NFS allows two flavors of mount, soft and hard. A hard mounted filesystem will retry NFS calls forever if the server goes down, while a soft mount gives up after a while and returns an error. The problem with soft mounts is that most UNIX programs are not very good about checking return status from system calls so you can get some strange behavior when servers go down. A hard mounted filesystem, on the other hand, will never fail due to a server crash; it may cause processes to hang for a while, but data will not be lost.

In addition to the MOUNT server, we have added NFS server daemons These are user level processes that make an **nfsd** system call into the kernel, and never return. This provides a user context to the kernel NFS server which allows the server to sleep. Similarly, the block I/O daemon, on the client side, is a user level process that lives in the kernel and services asynchronous block I/O requests Because the RPC requests are blocking, a user context is necessary to wait for read-ahead and write-behind requests to complete. These daemons provide a temporary solution to the problem of handling parallel, synchronous requests in the kernel. In the future we hope to use a light-weight process mechanism in the kernel to handle these requests [4].

The NFS group started using the NFS in September, and spent the next six months working on performance enhancements and administrative tools to make the NFS easier to install and use. One of the advantages of the NFS was immediately obvious; as the *df* output below shows, a diskless workstation can have access to more than a Gigabyte of disk!

Filesystem	kbytes	used	avail	capacity	Mounted on
/dev/nd0	7445	5788	912	86%	/
/dev/ndp0	5691	2798	2323	55%	/pub
panic:/usr	27487	21398	3340	86%	/usr
fiat:/usr/src	345915	220122	91201	71%	/usr/src
panic:/usr/panic	148371	116505	17028	87%	/usr/panic
galaxy:/usr/galaxy	7429	5150	1536	77%	/usr/galaxy
mercury:/usr/mercury	301719	215179	56368	79%	/usr/mercury
opium:/usr/opium	327599	36392	258447	12%	/usr/opium

The Hard Issues

Several hard design issues were resolved during the development of the NFS. One of the toughest was deciding how we wanted to use the NFS. Lots of flexibility can lead to lots of confusion.

Root Filesystems

Our current NFS implementation does not allow shared NFS root filesystems. There are many hard problems associated with shared root filesystems that we just didn't have time to address. For example, many well-known, machine specific files are on the root filesystem, and too many programs use them. Also, sharing a root filesystem implies sharing `/tmp` and `/dev`. Sharing `/tmp` is a problem because programs create temporary files using their process id, which is not unique across machines. Sharing `/dev` requires a remote device access system. We considered allowing shared access to `/dev` by making operations on device nodes appear local. The problem with this simple solution is that many programs make special use of the ownership and permissions of device nodes.

Since every client has private storage (either real disk or ND) for the root filesystem, we were able to move machine specific files from shared filesystems into a new directory called `/private`, and replace those files with symbolic links. Things like `/usr/lib/crontab` and the whole directory `/usr/adm` have been moved. This allows clients to boot with only `/etc` and `/bin` executables local. The `/usr`, and other filesystems are then remote mounted.

Filesystem Naming

Servers export whole filesystems, but clients can mount any sub-directory of a remote filesystem on top of a local filesystem, or on top of another remote filesystem. In fact, a remote filesystem can be mounted more than once, and can even be mounted on another copy of itself! This means that clients can have different "names" for filesystems by mounting them in different places.

To alleviate some of the confusion we use a set of basic mounted filesystems on each machine and then let users add other filesystems on top of that. Remember though that this is just policy, there is no mechanism in the NFS to enforce this. User home directories are mounted on `/usr/servername`. This may seem like a violation of our goals because hostnames are now part of pathnames but in fact the directories could have been called `/usr/1`, `/usr/2`, etc. Using server names is just a convenience. This scheme makes workstations look more like timesharing terminals because a user can log in to any workstation and her home directory will be there. It also makes tilde expansion (~username is expanded to the user's home directory) in the C shell work in a network with many workstations.

To aviod the problems of loop detection and dynamic filesystem access checking, servers do not cross mount points on remote **lookup** requests. This means that in order to see the same

filesystem layout as a server, a client has to remote mount each of the server's exported filesystems.

Credentials, Authentication and Security

We wanted to use UNIX style permission checking on the server and client so that UNIX users would see very little difference between remote and local files. RPC allows different authentication parameters to be "plugged-in" to the packet header of each call so we were able to make the NFS use a UNIX flavor authenticator to pass uid, gid, and groups on each call. The server uses the authentication parameters to do permission checking as if the user making the call were doing the operation locally.

The problem with this authentication method is that the mapping from uid and gid to user must be the same on the server and client. This implies a flat uid, gid space over a whole local network. This is not acceptable in the long run and we are working on different authentication schemes. In the mean time, we have developed another RPC based service called the Yellow Pages (YP) to provide a simple, replicated database lookup service [5]. By letting YP handle `/etc/passwd` and `/etc/group` we make the flat uid space much easier to administrate.

Another issue related to client authentication is super-user access to remote files. It is not clear that the super-user on a workstation should have root access to files on a server machine through the NFS. To solve this problem the server maps user *root* (uid 0) to user *nobody* (uid −2) before checking access permission. This solves the problem but, unfortunately, causes some strange behavior for users logged in as *root*, since *root* may have fewer access rights to a file than a normal user.

Remote *root* access also affects programs which are set-uid *root* and need access to remote user files, for example *lpr*. To make these programs more likely to succeed we check on the client side for RPC calls that fail with EACCES and retry the call with the real-uid instead of the effective-uid. This is only done when the effective-uid is zero and the real-uid is something other than zero so normal users are not affected.

While restricting super-user access helps to protect remote files, the super-user on a client machine can still gain access by using *su* to change her effective-uid to the uid of the owner of a remote file.

Concurrent Access and File Locking

The NFS does not support remote file locking. We purposely did not include this as part of the protocol because we could not find a set of locking facilities that everyone agrees is correct. Instead we plan to build separate, RPC based file locking facilities. In this way people can use the locking facility with the flavor of their choice with minimal effort.

Related to the problem of file locking is concurrent access to remote files by multiple clients. In the local filesystem, file modifications are locked at the inode level. This prevents two processes writing to the same file from intermixing data on a single write. Since the server maintains no locks between requests, and a **write** may span several RPC requests, two clients writing to the same remote file may get intermixed data on long writes.

UNIX Open File Semantics

We tried very hard to make the NFS client obey UNIX filesystem semantics without modifying the server or the protocol. In some cases this was hard to do. For example, UNIX allows removal of open files. A process can open a file, then remove the directory entry for the file so that it has no name anywhere in the filesystem, and still read and write the file. This is a disgusting bit of UNIX trivia and at first we were just not going to support it, but it turns out that all of the programs that we didn't want to have to fix (*csh*, *sendmail*, etc.) use this for temporary files.

What we did to make open file removal work on remote files was check in the client VFS **remove** operation if the file is open, and if so **rename** it instead of removing it. This makes it (sort of) invisible to the client and still allows reading and writing. The client kernel then removes the new name when the vnode becomes inactive. We call this the 3/4 solution because if the client crashes between the **rename** and **remove** a garbage file is left on the server. An entry to *cron* can be added to clean up on the server.

Another problem associated with remote, open files is that access permission on the file can change while the file is open. In the local case the access permission is only checked when the file is opened, but in the remote case permission is checked on every NFS call. This means that if a client program opens a file, then changes the permission bits so that it no longer has read

permission, a subsequent **read** request will fail. To get around this problem we save the client credentials in the file table at open time, and use them in later file access requests.

Not all of the UNIX open file semantics have been preserved because interactions between two clients using the same remote file can not be controlled on a single client. For example, if one client opens a file and another client removes that file, the first client's **read** request will fail even though the file is still open.

Time Skew

Time skew between two clients or a client and a server can cause time associated with a file to be inconsistent. For example, *ranlib* saves the current time in a library entry, and *ld* checks the modify time of the library against the time saved in the library. When *ranlib* is run on a remote file the modify time comes from the server while the current time that gets saved in the library comes from the client. If the server's time is far ahead of the client's it looks to *ld* like the library is out of date. There were only three programs that we found that were affected by this, *ranlib*, *ls* and *emacs*, so we fixed them.

This is a potential problem for any program that compares system time to file modification time. We plan to fix this by limiting the time skew between machines with a time synchronization protocol.

Performance

The final hard issue is the one everyone is most interested in, performance.

Much of the time since the NFS first came up has been spent in improving performance. Our goal was to make NFS faster than the ND in the 1.1 Sun release (about 80% of the speed of a local disk). The speed we are interested in is not raw throughput, but how long it takes to do normal work. To track our improvements we used a set of benchmarks that include a small C compile, tbl, nroff, large compile, f77 compile, bubble sort, matrix inversion, make, and pipeline.

The graph below shows the speed of the first NFS kernel compared to various disks on the 1.1 release of the kernel. The NFS and ND benchmarks were run using a Sun-2 (68010 running at 10 Mhz with no wait states) model 100U for the client machine, and a Sun-2 120 for the server, with Sun 10 Megabit ethernet boards. The disk benchmarks were done on a Fujitsu Eagle with a Xylogics 450 controller, and a Micropolis 42-Megabyte drive with a SCSI controller. Notice that NFS performance is pretty bad, except in the case of matrix inversion, because there is essentially no filesystem work going on.

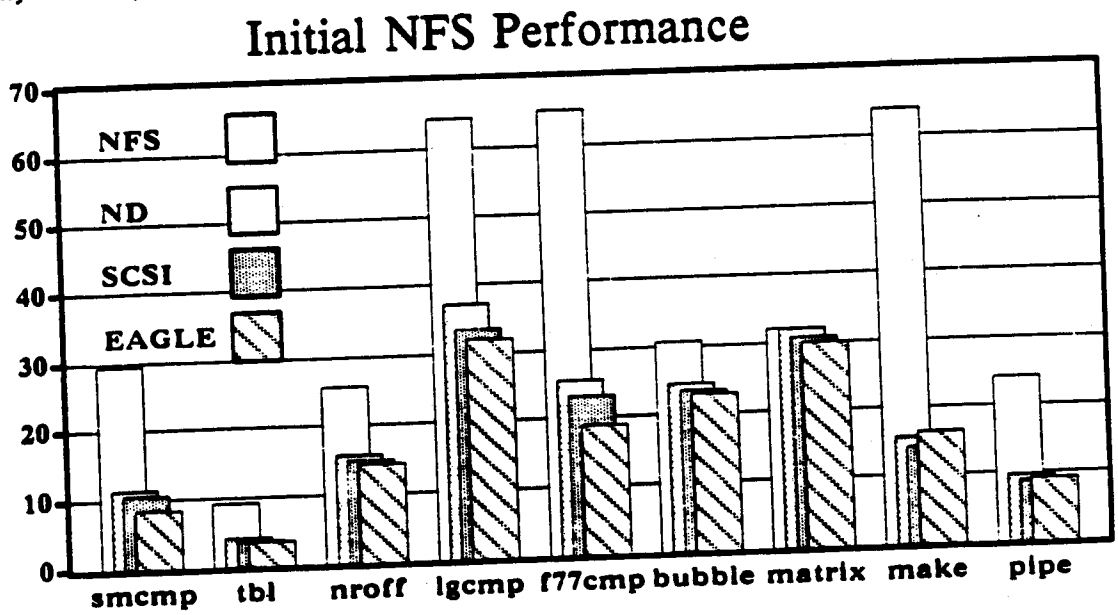

Figure 2

In our first attempt to increase performance we added buffer caching on the client side to decrease the number of **read** and **write** requests going to the server. To maintain cache

consistency, files are flushed on close. This helped a lot, but in reading and writing large files there were still too many requests going to the server. We were able to decrease the number of requests by changing the maximum UDP packet size from 2048 bytes to 9000 bytes (8k requests plus some overhead). This allows the NFS to send one large request and let the IP layer fragment and reassemble the packet. With a little work on the IP fragmentation code this turns out to be a big win in terms of raw throughput.

A *gprof* of the kernel, both client and server sides, showed that *bcopy* was a big consumer because the NFS and RPC kernel code caused three *bcopy*s on each side. We managed to trim that down to two copies on each side by doing XDR translation directly into, and out of, *mbuf* chains.

Using statistics gathered on the NFS server, we noticed that **getattr** (stat) accounted for 90% of the calls made to the server. In fact, the stat system call itself caused eleven RPC requests, seven of which were **getattr** requests on the same file. To speed up **getattr** we added a client side attribute cache. The cache is updated every time new attributes arrive from the server, and entries are discarded after three seconds for files or thirty seconds for directories. This caused the number of **getattr** requests to drop to about 10% of the total calls.

To make sequential read faster we added read-ahead in the server. This helped somewhat but it was noted that most of the **read** requests being done were in demand-loading executables, and these were not benefiting from read-ahead. To improve loading of executables we use two tricks. First, fill-on-demand clustering is used to group many small page-in requests into one large one. The second trick takes advantage of the fact that most small programs touch all of their pages before exiting. We treat 413 (paged in) programs as 410 (swapped in) if they are smaller than a fixed threshold size. This speeds up both the local and remote filesystems because loading a small program happens all at once, which allows read-ahead. This may sound like a hack but it can be thought of as a better initial estimate of the working set of small programs, since small programs are more likely to use all of their pages than none of them.

To make lookup faster we decided to add yet another cache to the client side. The directory name lookup cache holds the vnodes for remote directory names. This helps speed up programs that reference many files with the same initial path. The directory cache is flushed when the attributes returned in a NFS request do not match the attributes of the cached vnode.

Figure 3 shows the performance over the whole set of benchmarks for NFS compared to our performance goal (ND in the 1.1 release) and to an Eagle disk. Notice that the Eagle also got faster as a result of these improvements.

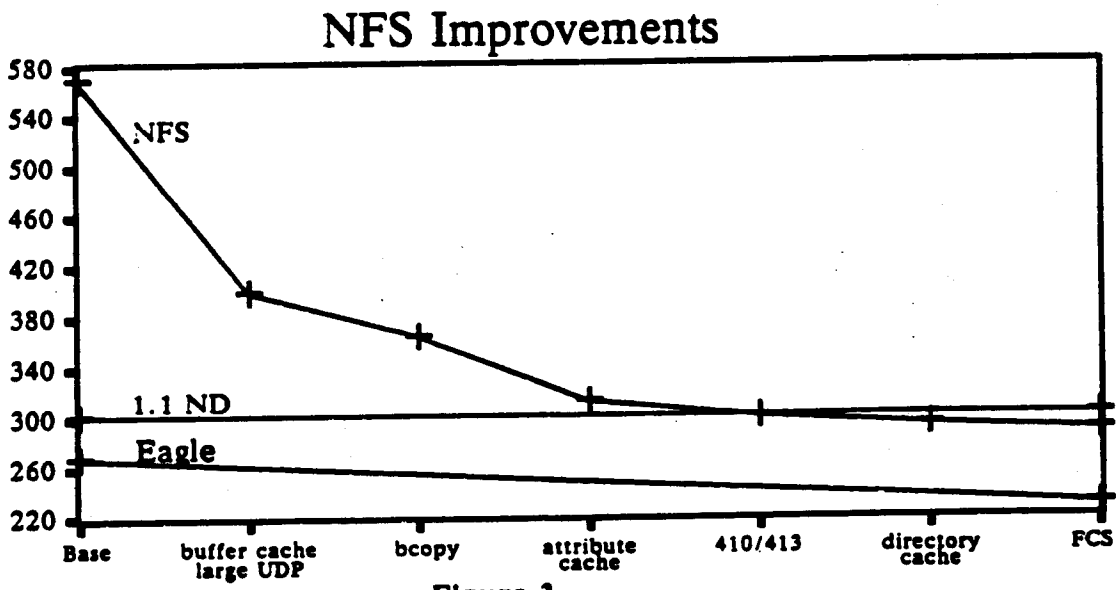

Figure 3

In Figure 4 below we give the benchmark numbers for the current release of the NFS. The biggest remaining problem area is *make*. The reason is that *stat*-ing lots of files causes one RPC call to the server for each file. In the local case the inodes for the whole directory end up in the buffer cache and then *stat* is just a memory reference. The other operation that is slow is write because it is synchronous on the server. Fortunately, the number of write calls in normal use is very small (about 5% of all calls to the server) so it is not noticeable unless the client does a large

write to a remote file. To speed up *make* we are considering modifying the **getattr** operation to return attributes for multiple files in one call.

Since many people in the UNIX community base performance estimates on raw transfer speed we also measured those. The current numbers on raw transfer speed are: 120 kilobytes/second for read (`cp bigfile /dev/null`) and 40 kilobytes/second for write. Figure 4, below, shows the same set of benchmarks as in Figure 2, this time run with the current NFS release.

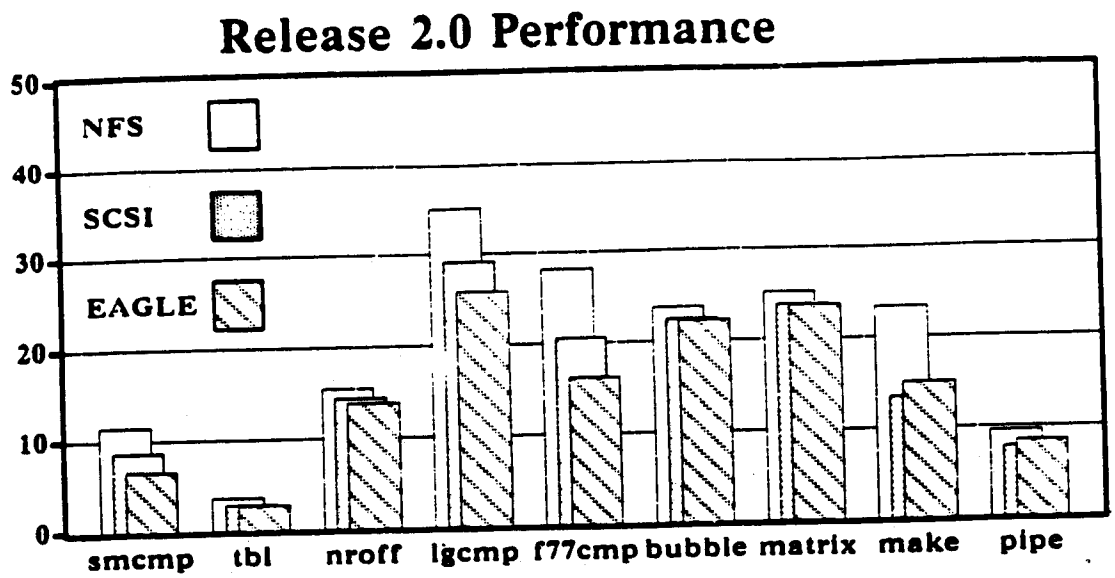

Figure 4

What Next?

These are some of the outstanding issues and new features of NFS that we will be working on in the future:

Full Diskless Operation
: One of the biggest problems with the current release is that diskless machines must use both ND and NFS. This makes administration twice as hard as it should be, and also makes our job twice as hard since we must support two protocols. We will be working on TFTP booting and additions to NFS to allow shared root filesystems, shared remote swapping, and remote device access. Together these will allow us to run diskless clients with only one remote file access method.

Remote File Locking
: We plan to build remote file locking services that are separate from the NFS service. Since file locking is inherently stateful (the server maintains the lock information) it will be built using the Sun status monitor service [6].

Other Filesystem Types
: The filesystem types that we have implemented so far are 4.2, NFS and a MS/DOS filesystem that runs on a floppy. We have barely scratched the surface of the usefulness of the filesystem interface. The interface could be used, for example, to implement filesystems to allow UNIX access to VMS or System V disk packs.

Performance
: We will continue our work on increasing performance, in particular, we plan to explore hardware enhancements to the server side since the server CPU speed is the bottleneck in the current implementation. We are currently considering building a low cost, stand-alone NFS server that would use a new filesystem type for higher performance and to allow automatic repair without operator intervention.

Better Security
: The NFS, like most network services, is prone to security problems because

programs can be written that impersonate a server There are also problems in the current implementation of the NFS with clients impersonating other clients. To improve security, we plan to build a better authentication scheme that uses public key encryption.

Automatic Mounting

We are considering building a new filesystem type which would give access to all of the exported filesystems in the network. The root directory would contain a directory for each accessable, remote filesystem. Adding protocol support for automatic redirection would allow a server to advise a client when a mount point has been reached, and the client could then automatically mount that remote filesystem. With this combination of new features clients could have access to all remote filesystems without having to explicitly do mounts.

Conclusions

We think that the NFS protocols along with RPC and XDR provide the most flexible method of remote file access available today. To encourage others to use NFS, Sun is making public all of the protocols associated with NFS. In addition, we have published the source code for the user level implementation of the RPC and XDR libraries. We are also working on a user level implementation of the NFS server which can easily be ported to different architectures.

Acknowledgements

There were many people throughout Sun who were involved in the NFS development effort. Bob Lyon led the NFS group and helped with protocol issues, Steve Kleiman implemented the filesystem interface in the kernel from Bill Joy's original design, Russel Sandberg ported RPC to the kernel and implemented the NFS virtual filesystem, Tom Lyon designed the protocol and provided far sighted inputs into the overall design, David Goldberg worked on many user level programs, Paul Weiss implemented the Yellow Pages, and finally, Dan Walsh is the one to thank for the performance of NFS.

I would like to thank Interleaf for making it possible for me to write this paper without using troff!

References

[1] B. Lyon, "Sun Remote Procedure Call Specification," Sun Microsystems, Inc. Technical Report, (1984).

[2] R. Sandberg, "Sun Network Filesystem Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[3] B. Lyon, "Sun External Data Representation Specification," Sun Microsystems, Inc. Technical Report, (1984).

[4] J. Kepecs, "Lightweight Processes for UNIX Implementation and Applications," Usenix (1985)

[5] P. Weiss, "Yellow Pages Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[6] J. M. Chang, "SunNet," Usenix (1985)