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.

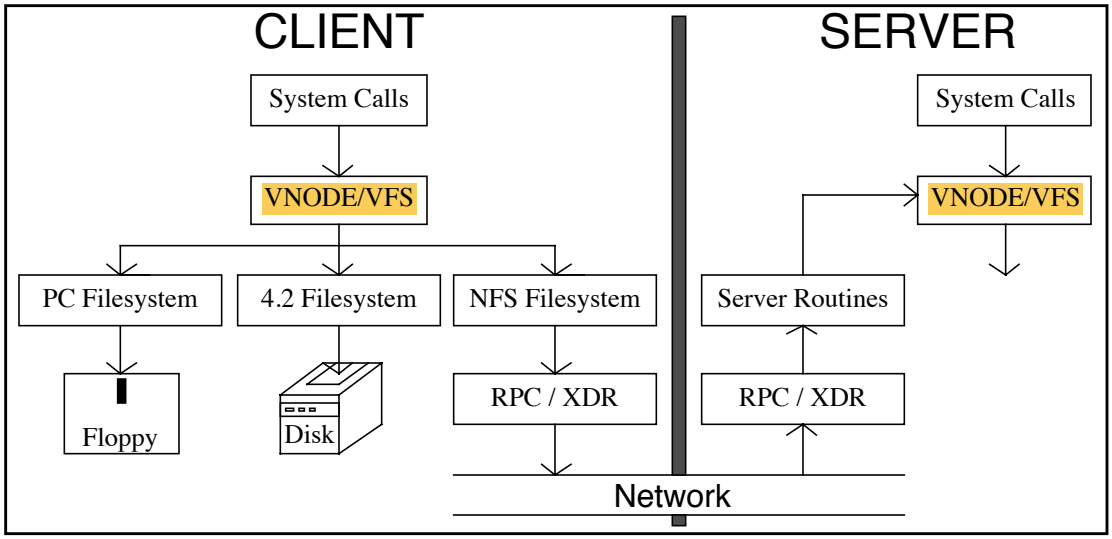

Figure 1

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

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. The memory allocator is also used by the kernel RPC implementation.

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 *getdirentries* 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 1984, the user level RPC and XDR libraries were ported from the user-level library 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 on a Sun-2 (68010). 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 by a mount request. It checks the file /etc/exports which contains a list of exported filesystems and the clients that can import them (see appendix 1). If the client has import permission, the mount daemon does a **getfh** system call to convert the pathname being imported 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 types at mount time. For example, NFS allows two flavors of mount, soft and hard. A hard mounted filesystem will retry NFS requests 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.

To allow automatic mounting at boot time and to keep track of currently mounted filesystems, the /etc/fstab and /etc/mtab file formats were changed to use a common ASCII format that is similar to the /etc/fstab format in Berkeley 4.2 with the addition of a type and an options field. The type field is used to specify filesystem type (nfs, 4.2, pc, etc.) and the options field is a comma separated list of option strings, such as rw, hard and nosuid (see appendix 1).

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. They provide 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 RPC requests block, 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].

We started using NFS at Sun in September 1984, and spent the next six months working on performance enhancements and administrative tools to make NFS easier to install and use. One of the advantages of NFS was immediately obvious; the *df* output below is from a diskless machine with 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 NFS. One of the toughest was deciding how we wanted to use NFS. Lots of flexibility can lead to lots of confusion.

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 that this is policy, there is no mechanism in 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 NFS clients look more like timesharing terminals because a user can log in to any machine and her home directory will be there. It also makes tilde expansion (where *~username* is expanded to the user's home directory) in the C shell work in a network with many machines.

To avoid 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

NFS uses UNIX style permission checking on the server and client so that UNIX users see very little difference between remote and local files. RPC allows different authentication parameters to be "plugged-in" to the message header so we are able to make 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 a network authentication method which allows users to "login" to the network [12]. This will provide a network-wide identity per user regardless of the user's identity on a particular machine. 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/hosts, /etc/passwd and /etc/group we make the flat uid space much easier to administer.

Another issue related to client authentication is super-user access to remote files. It is not clear that the super-user on a machine should have root access to files on a server machine through NFS. To solve this problem the server can map 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 remote file than a normal user.

Concurrent Access and File Locking

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 file locking facilities that everyone agrees is correct. Instead we have a separate, RPC based file locking facility. Because file locking is an inherently stateful service, the lock service depends on yet another RPC based service called the status monitor [6]. The status monitor keeps track of the state of the machines on a network so that the lock server can free the locked resources of a crashed machine. The status monitor is important to stateful services because it provides a common view of the state of the network.

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 NFS 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, but, in practice, this has never been necessary.

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 cannot 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 the times 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.

Time skew 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 development time of NFS has been spent in improving performance. Our goal was to make NFS comparable in speed to a small local disk The speed we were 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.

To improve the performance of NFS, we implemented the usual read-ahead and write-behind buffer caches on both the client and server sides. We also added caches on the client side for file attributes and directory names. To increase the speed of read and write requests, we increased the maximum size of UDP packets from 2048 bytes to 9000 bytes. We cut down the number of times data is copied by implementing a new XDR type that does XDR translation directly into and out of *mbufs* in the kernel.

With these improvements, a diskless Sun-3 (68020 at 16.67 Mhz.) using a Sun-3 server with a Fujitsu Eagle disk, runs the benchmarks faster than the same Sun-3 with a local Fujitsu 2243AS 84 Mega-byte disk on a SCSI interface.

The two remaining problem areas are **getattr** and **write**. The reason is that *stat*-ing files causes one RPC call to the server for each file. In the local case the inodes for a whole directory end up in the buffer cache and then *stat* is just a memory reference. The **write** operation is slow 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, see appendix 2) so it is not noticeable unless the client writes a large remote file.

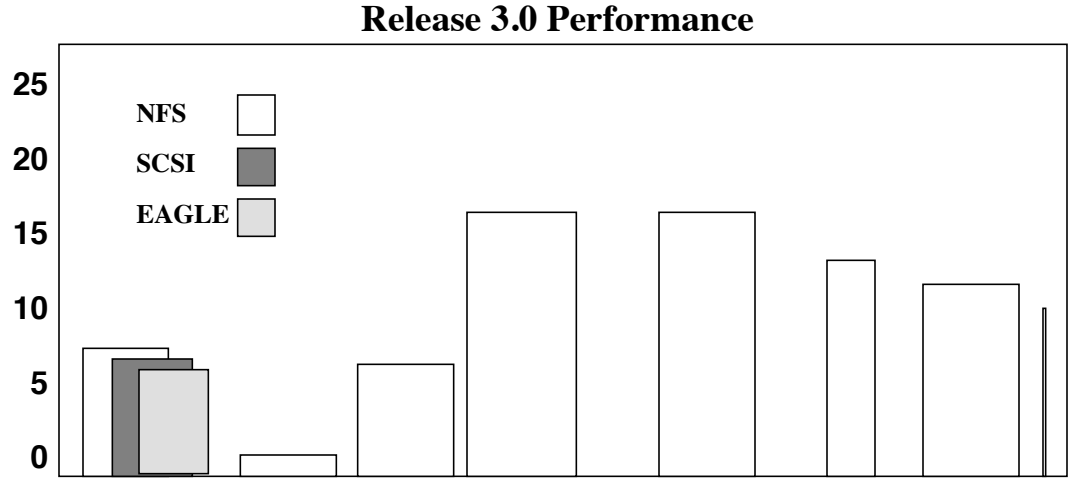

Figure 3

In Figure 3, above, we show some benchmark results comparing NFS and local SCSI disk performance for the current Sun software release. The scale on the left is unitless numbers. It is provided to make comparison easier.

Since many people base performance estimates on raw transfer speed we also measured those. The current numbers on raw transfer speed are: 250 kilobytes/second for read (cp bigfile /dev/null) and 60 kilobytes/second for write on a Sun-3 with a Sun-3 server.

Other Remote Filesystems

Why, you may ask, do we need NFS when we already have Locus[14], Newcastle Connection[15], RFS[8], IBIS[16] and EFS[10]. In most cases the answer is simple: NFS is designed to handle non-homogeneous machines and operating systems, it is fast, and you can get it today. Other than the Locus system, which provides file replication and crash recovery, the other remote filesystems are very similar to each other.

RFS vs NFS

The AT&T Remote Filesystem (RFS), which has been demonstrated at USENIX and UniForum conferences but not yet released, will provide much of the same functionality as NFS. It allows clients to mount filesystems from a remote server and access those files in a transparent way. The differences between them mostly stem from the basic design philosophies. NFS provides a general network service, while RFS provides a distributed UNIX filesystem [9]. This difference in philosophy shows up in many different areas of the designs.

Networking

RFS does not use standard network transport protocols, like UDP/IP. Instead it uses a special purpose transport protocol which has not been published, and implementations of it are not generally available. This protocol cannot easily be replaced because RFS depends on properties of the transport virtual circuit to determine when a machine has crashed. NFS uses the RPC layer to hide the underlying protocols, which makes it easy to support different transport protocols without having to change the NFS protocols.

RFS does not use a remote procedure call mechanism, instead it extends the semantics of UNIX system calls so that a system call which accesses a remote file goes over the network and continues execution on the server. When the system call is finished, the results are returned to the client. This protocol is complicated by the fact that both client and server can interrupt a remote system call. In addition, the system calls which deal with filenames had to be modified to handle a partial lookup on the server when a client mount point is encountered in the pathname. In this case the server looks up part of the name then returns control to the client to look up the rest.

Non-Homogeneous Machines and Operating Systems

While NFS currently runs on 16 different vendors hardware, and under Berkeley 4.2, Sun OS, DEC Ultrix, System V.2, VMS and MS/DOS, RFS will run only System V.3 based UNIX systems. The NFS design is based on the assumption that most installations have many different types of machines on their network, and that these machines run widely varying systems. The RFS protocol includes a canonical format for data to help support different machine architectures, but no attempt is made to support operating systems

other than System V.3. The NFS design does not try to predict the future. Instead, it includes enough flexibility to support evolving software, hardware, and protocols.

Flexibility

Because RFS is built on proprietary protocols with UNIX semantics built in, it is hard to imagine using those protocols from different operating systems. NFS, on the other hand, provides flexibility through the RPC layer. RPC allows different transport protocols, authentication methods, and server versions to be supported in a single implementation. This allows us, for example, to use an encrypted authentication method for maximum security among workstations, while still allowing access by PC's using a simpler authentication method. It also makes protocol evolution easier since clients and servers can support different versions of the RPC based protocols simultaneously.

RFS uses streams [7] to hide the details of underlying protocols. This should make it easy to plug in new transport protocols. Unfortunately, RFS uses the virtual circuit connection of the transport protocol to detect server and client crashes*. This means that even the reliable byte stream protocol TCP/IP cannot be plugged in because TCP connections do not go away when one end crashes unless there is data flowing at the time of the crash.

Crash Recovery

The RFS uses a stateful protocol. The server must maintain information about the current mount points of all of its clients, the open files, directories, and devices held by its clients, as well as the state of all client requests that are in progress. Because it would be very difficult and costly for the client to rebuild the server's state after a server crash, RFS does not do server crash recovery. A server or client crash is detected when the protocol connection fails, at which point all operations in progress to that machine are aborted. When an RFS server crashes it is roughly equivalent, from the client's point of view, to losing a local disk.

If server crashes are rare events doing no recovery is acceptable, however, keep in mind that network delays, breaks, or overloading usually cannot be distinguished from a machine crash. As networks grow the possibility of network failures increases, and as the connectivity of the network increases so does the chance of a client or server crash. We decided early in the design process that NFS must recover gracefully from machine and network problems. NFS does not need to do crash recovery on the server because the server maintains no state about its clients. Similarly, the client recovers from a server crash simply by resending a request.

Administration

There are two major differences between administration of NFS and RFS. The use of a uid mapping table on RFS servers removes the need for uniform uid to user mapping through out the network. NFS assumes a uniform uid space and we provide the Yellow Pages service to make distribution and central administration of system databases (like /etc/passwd and /etc/group) easier. NFS also has a MOUNT RPC service for each machine acting as a server. The exported filesystem information is maintained on each machine and made available by this service. RFS uses a centralized name service running on one machine on the network to keep track of advertised filesystems for all servers. A centralized name service was not acceptable in NFS because it forces all clients and servers to use the same protocol for exchanging mount information. By having a separate protocol for the MOUNT service we can support different filesystem access checking and different operating system dependent features of the mount operation.

UNIX Semantics

NFS does not support all of the semantics of UNIX filesystems on the client. Removing an open file, append mode writes, and file locking are not fully implemented by NFS. RFS does implement 100% of the UNIX filesystem semantics. However, if a server crashes or a filesystem is taken out of service, client applications can see error conditions which normally could only happen due to a disk failure. Since this is an error condition that is so severe that it usually means that the whole system has failed, most applications will not even try to recover.

Availability

NFS has been a product for more than a year. Source and support for NFS on Berkeley 4.2 BSD is available through Sun and Mt. Xinu, and for System V.2 through Lachman Associates, The Instruction Set, and Unisoft. RFS has not yet been released.

---

* The exclusive use of transport properties to drive session semantics is a common design flaw in many n
work applications.

Conclusion

For a small network of machines all running System V.3, RFS is the obvious choice for remote access to files since it will come with V.3 and it implements all of the UNIX semantics. For a large network or a network of mixed protocols, machine types, and operating systems, NFS is the better choice. It should be understood that NFS and RFS are not mutually exclusive. It will be possible to run both on a single machine.

Porting Experience

In the many ports of NFS to foreign hardware and systems we have found only a few places where additions to the protocol would be helpful. The IBM PC client side port was done almost exclusively from the protocol specification, and a simple, user-level server was also implemented from the specification.

NFS has been ported to five different operating systems, two of which are not UNIX based, and to many different types of machines. Each port had its own interesting problems.

The first port of NFS was to a VAX 750 running Berkeley 4.2 BSD. This was also the easiest port since our code is based on 4.2 UNIX. Modifying the kernel to use the vnode/VFS interface was the most time consuming part of the porting effort. Once the vnode/VFS interface was in, the NFS and RPC code pretty much just dropped in. Some libraries had to be updated, and programs that read directories had to be recompiled. The whole port took about two man-weeks to complete. This port was then used as the distribution source for later ports.

The System V.2 port was done in a joint effort by Lachman Associates and The Instruction Set on a VAX 750. In order to avoid having to port the Berkeley networking code to the System V kernel an Excelan board was used. The Excelan board handles the ethernet, IP, and UDP layers. A new RPC transport layer had to be implemented to interface to the Excelan board. Adding the vnode/VFS interface to the System V kernel was the hardest part of the port.

The port to the IBM PC, done by Geoff Arnold and Kim Kinnear at Sun, was complicated by the need to add a "redirector" layer to MS/DOS to catch system calls and redirect them. An implementation of UDP/IP also had to be added before RPC could be ported. The NFS client side implementation is written in assembler and occupies about 40K bytes of space. Currently, remote *read* operations are faster than a local hard disk access but remote *write* operations are slower. Over all, performance is about the same for remote and local access.

DEC has ported NFS to Ultrix on a Microvax II. This port was harder than the 4.2 port because the Ultrix release that was used is based on Berkeley 4.3beta. The most time consuming part of the port was, again, installing the vnode/VFS interface. This was complicated by the fact that Berkeley has made many changes to much of the kernel code that deals with inodes.

Another interesting port, while not a different operating system, was the Data General MV 4000 port. The DG machine runs System V.2 with Berkeley 4.2 networking and filesystem added. This made the RPC and vnode/VFS part of the port easy. The hard part was XDR. The MV 4000 has a word addressed architecture, and character pointers are handled very differently than word pointers. There were many places in the code, and especially in the XDR routines that assumed that (char *) == (int *).

As an aid to porting we have implemented a user-level version of the NFS server (UNFS). It uses the standard RPC and XDR libraries and makes system calls to handle remote procedure call requests. The UNFS can be ported to non-UNIX operating systems by changing the system calls and library routines that are used. Our benchmarks show it to be about 80% of the performance of a kernel based NFS server for a single client and server.

The VMS implementation is for the server side only. The basic port was done by Dave Kashtan at SRI. He started with the user-level NFS server and used the EUNICE UNIX-emulation libraries to handle the UNIX system calls. The RPC layer was ported to use a version of the Berkeley networking code that runs under VMS. Some caching was added to the libraries to speed up the system call emulation and to perform the mapping from UNIX permission checking to VMS permission checking.

At the UniForum conference in February 1986, all of the completed NFS ports were demonstrated. There were 16 different vendors and five different operating systems all sharing files over an ethernet.

Also at UniForum, IBM officially announced their RISC based workstation product, the RT. Before the announcement, NFS had already been ported to the RT under Berkeley 4.2 BSD by Mike Braca at Brown University.

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 has made 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.

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 Dan Walsh is the one to thank for the performance of NFS. The NFS consulting group, headed by Steve Isaac, has done an amazing job of getting NFS out to the world.

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).

[7] D.L. Presotto and D. M. Ritchie, "Interprocess Communication in the Eighth Edition UNIX System," USENIX Conference Proceedings, (June 1985).

[8] P. J. Weinberger, "The Version 8 Network File System," USENIX Conference Proceedings, (June 1985).

[9] M. J. Hatch, et al., "AT&T's RFS and Sun's NFS, A Comparison of Heterogeneous Distributed File Systems," UNIX World, (December 1985).

[10] C. T. Cole, et al., "An Implementation of an Extended File System for UNIX," USENIX Conference Proceedings, (June 1985).

[11] B. Taylor, "A protocol compiler for RPC," Sun Microsystems, Inc. Technical Report, (December 1985).

[12] B. Taylor, "A Secure Network Authentication Method for RPC," Sun Microsystems, Inc. Technical Report, (November 1985).

[13] S. R. Kleiman, "An Architecture for Multiple File Systems in Sun UNIX," Sun Microsystems, Inc. Technical Report, (October 1985).

[14] Popek, et al., "The LOCUS Distributed Operating System," Operating Systems Review ACM, (October 1983).

[15] D. R. Brownbridge, et al., "The Newcastle Connection or UNIXes of the World Unite!," Software -- Practice and Experience, (1982).

[16] W. F. Tichy, et al., "Towards a Distributed File System," USENIX Conference Proceedings, (June 1985).

Appendix 1

/etc/fstab **and** /etc/mtab **format**

The format of the filesystem database files /etc/fstab and /etc/mtab were changed to include type and options fields. The type field specifies which filesystem type this line refers to, and the options field specifies mount and run time options. The options field is a list of comma separated strings. This allows new options to be added, for example when a new filesystem type is created, without having to change the library routines that parse these files. The example below is the /etc/fstab file from a diskless machine.

(Filesystem	mount point	type	options)		
/dev/nd0	/	4.2	rw	1	1
/dev/ndp0	/pub	4.2	ro	0	0
speed:/usr.MC68010	/usr	nfs	ro,hard	0	0
#opium:/usr/opium	/usr/opium	nfs	rw,hard	0	0
speed:/usr.MC68020/speed	/usr/speed	nfs	rw,hard	0	0
panic:/usr/src	/usr/src	nfs	rw,soft,bg	0	0
titan:/usr/doctools	/usr/doctools	nfs	ro,soft,bg	0	0
panic:/usr/panic	/usr/panic	nfs	rw,soft,bg	0	0
panic:/usr/games	/usr/games	nfs	ro,soft,bg	0	0
wizard:/arch/4.3alpha	/arch/4.3	nfs	ro,soft,bg	0	0
sun:/usr/spool/news	/usr/spool/news	nfs	ro,soft,bg	0	0
krypton:/usr/release	/usr/release	nfs	ro,soft,bg	0	0
crayon:/usr/man	/usr/man	nfs	soft,bg	0	0
crayon:/usr/local	/usr/local	nfs	ro,soft,bg	0	0
topaz:/MC68010/db/release	/usr/db	nfs	ro,soft,bg	0	0
eureka:/usr/ileaf	/usr/ops	nfs	soft,bg	0	0
wells:/pe	/pe	nfs	rsize=1024	0	0

Mount Access Permission: the /etc/exports **File**

The file /etc/exports is used by the server's MOUNT protocol daemon to check client access to filesystems. The format of the file is <filesystem> <access-list>. If the access list is empty the filesystem is exported to everyone. The access-list consists of machine names and netgroups. Netgroups are like mail aliases, a single name refers to a group of machines. The netgroups database is accessed through the Yellow Pages. Below is and example /etc/exports file from a server.

(filesystem	access-list)
/usr	argon krypton
/usr/release	
/usr/misc	
/usr/local	
/usr/krypton	argon krypton phoenix sundae
/usr/3.0/usr/src	systems
/usr/src/pe	pe–users

Appendix 2

Below are the server NFS and RPC statistics collected from a typical server at Sun. Statistics are collected automatically each night, using the *nfsstat* command, and sent to a list of system administrators. The statistics are useful for load balancing and detecting network problems. Note that 1499689 calls/day = 62487 calls/hour = 17 calls/second, average over twenty four hours for one server!

Server rpc:

calls	badcalls	nullrecv	badlen	xdrcall
1499688	0	0	0	0

Server nfs:

calls	badcalls
1499688	0

null	getattr	setattr	root	lookup	readlink
0 0%	79897 5%	708 0%	0 0%	760709 50%	116712 7%

read	wrcache	write	create	remove	rename
452090 30%	0 0%	50151 3%	25394 1%	5605 0%	687 0%

link	symlink	mkdir	rmdir	readdir	fsstat
683 0%	83 0%	1 0%	1 0%	6960 0%	7 0%

Appendix 3

Sun Protocols in the ISO Open Systems Interconnect Model