Development Issues: NFS vs. RFS

There are two distributed file systems available for UNIX® System V Release 4 (SVR4) — NFS and RFS. NFS (Network File System) was originally developed by Sun Microsystems Inc. and is a distributed file-system package that provides resource sharing in a heterogeneous network environment. Files and directories can be shared with other machines running the UNIX operating system as well as other operating systems ranging from MS-DOS™ to VMS. RFS (Remote File Sharing) was originally developed by AT&T and provides the ability to share resources with other machines running UNIX System V (i.e. a homogeneous environment). Neither is limited to a single network provider (e.g., TCP/IP or STARLAN). NFS is new and enhanced for System V. RFS has been enhanced as well. Both are good at what they do but depending on your application one may be more appropriate than the other. In this article I will describe the salient features and capabilities of each file-system, examine how they do what they do, and shed some light on their relative efficiency and appropriateness with respect to applications that may need to use them.

Network File System

NFS on SVR4 runs across any connectionless-mode (also known as datagram) network provider (e.g., UDP or STARLANDG). Connectionless-mode is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. It tends to be fast for relatively small chunks of data but there is a good deal of redundancy to handle packets that arrive out of sequence or don't arrive at all.

NFS uses the ''client-server'' model of processing. A server is a machine that physically owns the resources, i.e., directories, files and devices. The client is the machine that uses the resources. Provided it owns physical resources, a machine can be a server, client or both. Otherwise, it can only be a client. An NFS server can advertise or share an entire file-system, a partial file-system, or a single file. Special device files, as well as ordinary files, can be shared; however, peripheral devices such as modems and printers cannot. Furthermore, a client need not mount the entire resource. Clients are able to mount partial tree structures of a file-system hierarchy. For example, Machine-A, the server, advertises `/usr` using the `share` command:

```
share -F nfs /usr
```

A client, Machine-B, is free to mount `/usr` or something under `/usr` (such as `/usr/man`). This is done using the `mount` command:

```
mount -F nfs Machine-A:/usr/man /usr/man
```

Access to Machine-A's `/usr/man` is now transparent to users on Machine-B.

Security is primarily controlled by the server via the `share` command. Resources can be shared to all clients as read-only or read-write, or as read-only or read-write to specific clients. Unfortunately, NFS does not support the concept of a client access control list. That is, once a resource has been advertised it is mountable by every system on the network. Whether is can be mounted read-only or read-write is the extent of the servers' control.

Another aspect of security, known as user-id mapping, is also controlled by the server via the `share` command. However, this capability is limited. The only control the server has is whether to map a client user-id of `0` (`root`) to a valid server user-id or to a user-id known as *anonymous* (which is really the user-id for `nobody`). The mapping selected can be applied to one or more specific clients or to the entire network.

Some of the enhancements to NFS for SVR4 were to improve security. This version of NFS is based on a Secure version of the RPC (Remote Procedure Call) mechanism and is referred to as Secure NFS. Secure RPC is optional. It provides for a more robust authentication of clients and remote users. It uses DES (Data Encryption Standard) authentication — a scheme outside the scope of this discussion. The point here is that more secure mechanisms for NFS exist but using them may reduce the number of NFS systems with which your system can communicate.

NFS consists of some kernel-level code and several user-level standing-servers, known in UNIX parlance as *daemons*. Since NFS is a transparent file-system, system-calls like `read()` and `write()` are intercepted by the kernel-level NFS code and, depending on the operation, are directed to the appropriate user-level daemons on the server or the client. The daemons handle communication with the server. The server daemons deal directly with the server kernel.

An SVR4 NFS system — depending on whether it is a server, client, or both — consists of 5 different daemons. These processes are a very active part NFS. They are:

`nfsd`	The primary NFS daemon. It is used to handle client file-system requests.
`biod`	The asynchronous block I/O daemon. It is used on client systems to buffer read-ahead and write-behind.
`mountd`	The mount daemon. It is used on the server to process `mount()` requests.
`lockd`	The lock daemon. It is used on the server and client to process lock requests (e.g., `fcntl()` or `lockf()`) that are either sent by the local kernel or remotely by another lock daemon.
`statd`	The network status daemon. It is used on the server and client and interacts with `lockd` to provide crash and recovery functions for the locking services.

Remote File Sharing

RFS runs over a number of network providers (or protocols). Its major requirement is that the network provider support connection-mode service. Connection-mode is circuit-oriented and enables the transmission of data over an established connection in a reliable, sequenced manner. It also provides an identification procedure that avoids the overhead of address resolution and transmission during data transfer. In today's OSI mania world, there are few network providers that do not support this service.

As with NFS, RFS uses the client-server model and a machine may be a server, a client, or both. RFS is different from NFS in that it supports the concept of *name-servers* and *domains*. A domain is a group of one or more machines on a network. There can be one or more domains on a network. The name-server is a machine whose responsibility it is to provide clients with resource information for a given domain. The resource information that the name-server provides is what machines have what resources available and the address of those systems. There is only one primary name-server per RFS domain. There can be one or more secondary name-servers. The primary name-server replicates its data on each secondary. The secondary name-servers take over if the primary name-server crashes.

The domain name-server extension to the client-server model provides a convenient way to encapsulate resource and network data as well as control

security. The primary name-server contains the names, addresses and authentication information of all the members of its domain. Other members of the domain need only know the name and address of the primary name-server to find all other members and resources in the domain. When a member advertises its resources it does so with the name-server. When a member wants to know what resources other members are advertising it queries the name-server. When a member wants to mount an advertised resource it contacts the name server, gets the address of the server member and then calls it directly. The member server calls the name-server to get the client members authentication information, authenticates the client member and then the mount is completed. Thereafter, client and server communicate directly, not via the name-server. Resources can be shared with other domains. In such cases, the name-server also contains the names and addresses of the other domain's primary name-server.

An RFS server advertises its resources via the same command interface as NFS. Servers can share an entire file-system or partial file-systems. Any file-type can be shared including devices such as tape-drives, printers and modems. Individual files cannot be shared. They are implicitly shared as part of sharing a complete or partial file-system. Also, clients cannot mount partial tree structures of a file-system hierarchy. That is, a client can only mount exactly what is advertised.

RFS is different than NFS in that resources are advertised using *resource-identifiers* (a symbolic name) versus path-names. The resource-identifier consists of two parts, the domain name and the resource name. For example, Machine-A is in the domain `Orion` and advertises the resource `/usr` as SYS_A_USR.

```
share -F rfs /usr SYS_A_USR
```

The domain Machine-A resides in is implicit in the advertisement. The advertised resources of a domain can be queried using `nsquery` (name-server query). Members querying the advertised resources in their own domain simply do

```
nsquery
```

Members of a different domain can query the resources advertised in the `Orion` domain by using

```
nsquery Orion.[1]
```

Security for RFS is also controlled through the `share` command. Whether a resource can be mounted read-only or read-write can be specified as well as who can mount the resource. Access to a resource can be available to all, restricted to members of one domain, or to a single machine.

RFS also provides a simple authentication scheme for clients. It is based on passwords, like the `login` mechanism. Encrypted passwords are stored on the name-server and clients must provide their password before they can mount a resource.

User-id mapping features are very extensive. Transparent mapping is supported — a client user-id is mapped to the same numeric user-id on the server. Arbitrary mapping is supported — any client user-id can be mapped to any user-id on the server. Also, a particular mapping can be restricted to only members of a specific domain or a single machine. Furthermore, group-id mapping is supported to the same extent as user-id mapping.

Like NFS, RFS consists of some kernel-code and some user-level daemons. After a client has mounted a resource, the two kernels are in direct virtual-circuit reliable data communication with each other. All file system-calls like `read()` and `fcntl()` are intercepted by the kernel-level code but unlike NFS, RFS does not use any user-level daemons to complete the operation. The RFS user-level daemons are used to handle secondary functions like establishing network connections and name-server recovery.

File-System Semantics

NFS does not support complete file-system semantics. For example, the UFS and S5 file-systems supports device special files and named-pipes. Although an NFS server can advertise these file-types, the client mounting them does not receive physical access to these objects on the server system. A client that opens a remote named-pipe gets a pipe local to the client's virtual memory. That is, another client opening the same named-pipe is not in communication with the first client as one would expect if the pipe were global. Furthermore, a client opening a remote device, say a tape drive, is not accessing the tape drive on the server. If the open has meaning in the clients virtual memory then the open will

---

1. The "`.`" is required syntax.

succeed. Most likely it will fail.

RFS supports full file-system semantics with a few minor caveats. A client opening an advertised named-pipe will get a pipe in the virtual memory of the server system. A second client opening the same named-pipe will find itself talking to the first client. Furthermore, a client opening an advertised tape device will be able to operate the servers tape device as if it were local to the client.

The caveats to RFS's support of complete file-system semantics are for STREAMS operations and updates to mapped files. In SVR4, all character devices, including pipes, became STREAMS based. Now consider a client that has opened the remote named-pipe. The client may do `read()` and `write()` but `putmessage()` and `getmessage()` are disallowed. This is because the pipe is a STREAMS pipe on the server and can have STREAMS semantics on the server, but it is not a STREAMS pipe on the client and therefore it only has ''file'' semantics. This may seem esoteric, but if your application expects to do `getmessage()` and `putmessage()` across a pipe, consider resorting to `read()` and `write()` when the former functions fail with `errno` equal to `ENOSTR`. The server side can still do `getmessage()` and `putmessage()` and lo and behold, the message boundaries will be preserved.

Mapped files are supported in RFS (and NFS). The point here is that the semantics of file mapping is weaker when the file is remote. Because a mapped file can be updated by virtual memory access (this could be something as simple as a MOVE machine instruction) versus system-call, the consistency of the file is in jeopardy unless explicit care is taken using `memcntl()`. This applies to mapped files that are updated by more than one process. Read-only mapped files are quite safe.

Reading and Writing

By examining the flow of data in each file-system we should learn something about their relative efficiency. If it is assumed that data flow is best represented by synchronous `read()` and `write()` requests, then there is very little significant difference between the way NFS and RFS process these system calls. It is true that each file-system uses a different mechanism but the common denominator is that each client kernel intercepts the system call and routes the request to the server kernel. Therefore, a statement of relative performance cannot be made based on synchronous I/O.[2]

Caching and Consistency

Caching saves time by satisfying I/O requests locally before going to the server. The complement to caching is the correctness or consistency of the cache. Keeping the cache correct does cost something and can impact overall performance. Cache consistency is most expensive when many clients are all accessing and updating the same file. Not keeping the cache correct can cost something as well — incorrect data. Ironically an inconsistent cache is most expensive in the same scenario as a consistent one. The most simple statement that can be made from all this is that cache consistency directly relates to file consistency.

Both NFS and RFS cache data in the client kernel. The difference between the two is that NFS is very ‘‘state-less’’ and RFS is very ‘‘state-full’’. An NFS client’s decision to update the data in its cache is based the age of that data. The server does not inform the client when something the client is holding, such as an i-node or data block, has been updated. That is, the client and server relationship is state-less. Therefore, there tend to be small windows (measured in seconds) of opportunity when what the client thinks is the state of the file-system is incorrect. Thus when many clients concurrently update the same file on the same server, inconsistent data in the file can result.

How a client process does its `open()`’s, `read()`’s and `write()`’s has some effect on the consistency of files it modifies. Files `open()`’ed with the `O_SYNC` flag will effect the semantics of its `write()` operations. When the client kernel returns from the `write()` call, the data is guaranteed to be in the file buffer of the server kernel. Without using this flag, data may be in the buffer of the server kernel or the client kernel. Using `fsync()` has the same semantics as `O_SYNC`.

Locking the file or parts of it can also help alleviate file consistency problems. Unfortunately, locking in NFS tends to be slow. For example, a client process requests a lock via `fcntl()`. The client kernel asks the client `lockd` to get the lock. The client `lockd` calls the server `lockd` and asks it for the lock. The server `lockd` gets the lock from the server kernel. It then informs the client

---

2. One could argue the relative efficiency of the underlying transport mechanisms — connectionless-mode versus connection-mode — but this topic is outside the scope of this article.

which in turn informs the client kernel which informs the client process by returning from the `fcntl()` call. Now the process can read the data, update it, and then release it via the same mechanism and communication path.

RFS's caching is more reliable. The server maintains knowledge about what data-objects (e.g., i-nodes and data blocks) its clients are currently holding. When the server makes a change to one of these data objects it notifies the client that the data it has is no longer valid. The client can then request the new valid data. In any case the client kernel is always working with current and accurate data. Furthermore, files opened with the `O_SYNC` flag are guaranteed to have been physically updated on the server when the `write()` returns. Using `fsync()` has the same semantics as `O_SYNC`.

RFS also guarantees that `write()`'s to a file `open()`'ed with the `O_APPEND` flag will always be made to the end of the file and never conflict with any other client writing to the end of the same file. NFS does not support this.

Locking is simplified and efficient. The client process requests a lock via `fcntl()`. The client kernel asks the server kernel for a lock. The release is via the same mechanism and communication path.

Recovery

When a server crashes it may be important for the client to know. Both file-systems detect server crashes in essentially the same fashion — the server stops responding. For some clients this may bring everything to a halt. Fortunately, both NFS and RFS provide for auto-retry of mounting resources and allow for secondary servers. This is very important in a diskless client environment. If there is only one server then an entire network could be hung.

Also important is how client processes detect (or fail to detect) when the server is out to lunch. With NFS you have an option as to how you would like to detect server failure. Resources can be mounted either ‘‘hard’’ or ‘‘soft’’. Client processes accessing hard mounted resources block until the server comes back up and then continue processing as if nothing ever happened. Even locks that a client process had will be recovered when the server is rebooted. This is provided by the cooperative magic of the `lockd` and `statd` daemons of the client and server. Client processes accessing soft mounted resources will experience file access system-call failures (e.g., `read()` and `write()`) when the server fails to respond. In either case the client kernel knows when the server fails and prints a error on the console.

RFS does not provide for hard mounted resources. All resources are mounted soft. Server failures in RFS result in client system-call failures. When the server crashes, file access system-calls will fail with `errno` set to `ENOLINK`. Locks that the client possessed when the server crashed are lost. The upside to this is that all other clients have lost their locks as well.

One last note on crashes and recovery. For both file-systems, client crashes are semantically equivalent to a `close()` and `umount()` of all active and mounted resources.

Summary

Now that we are at the end of our examination of the similarities and differences of NFS and RFS, you are probably wondering which is the better distributed file-system. The problem is that that question really doesn't have an answer. In some applications RFS will be more than adequate and in others unusable — like when connectivity to other operating systems is required. As for NFS, lack of complete file-system semantics will make it unsuitable for applications requiring access to remote devices, but it is excellent for the support of diskless clients. It all depends on your application requirements. This may sound like a cliche' so let me leave you with a more useful observation: In SVR4, NFS and RFS can run concurrently on the same machine.

Acknowledgements

I would like to acknowledge the help of Steve Breitstein and Manoj Tangri of UNIX Systems Laboratories, Inc., for their help in preparing this article.

Bibliography

"UNIX System V Release 4: Programmer's Reference Manual," Prentice Hall, Englewood Cliffs, NJ, 1990.

"UNIX System V Release 4: Programmer's Guide: Networking Interfaces," Prentice Hall, Englewood Cliffs, NJ, 1990.

"UNIX System V Release 4: Network User's and Administrators Guide" Prentice Hall, Englewood Cliffs, NJ, 1990.