by Peter Collinson, Hillside Systems

MICHELLE FRIESENHAHN WILBY

Network File Systems

I keep smashing the stupid connector that 3Com Corp. supplies to attach its very thin PCMCIA Ethernet card that slides into my laptop to the very thick BNC plug that connects the card into my Ethernet. The box in which the card came trumpets that 3Com has a new design for the connector; I presume people have complained about the facility to easily perform connector crunching. Well, 3Com in Ireland, the connector is still trivially smashable, and it really shouldn't be.

When I crunch the connector, I realize how dependent I have become on my LAN. Without the Ethernet connection, my laptop is useless. I cannot get any significant data into or out of it without the huge pain of using floppies. My son is pleased; it means that he can get on with the massively more important task of designing his new add-on level for the Dark Forces computer game to which he is addicted.

The main use of my LAN is file sharing. I use the network to carry the information that allows various systems to manage files stored on the disks of other computers. Of course, we think of the systems delivering the files as "servers" and those that access the files over the network as "clients." However, the distinction between the machines that are servers and those that are clients is blurred somewhat on my site, and I suspect the same is true at many sites.

I must confess I still have great feelings of delight when I access files on a remote system as if the files were sitting on a disk connected to the local machine. I like the whole notion. I enjoy the ease of use that mechanism permits. As time has gone on, I find that people to whom I explain the geography of my system are less surprised by the idea. Network file systems have become a fact of daily life for many people.

I'm using two different types of network protocols to provide remote file access on my network. My UNIX machines use NFS, the Network File System from Sun. My Windows NT and 95 systems want to talk LAN Manager protocols, and I use the excellent freeware Samba system from Andrew Tridgell and his team to support file access on the UNIX machines from my PC-based systems.

The NFS Protocol

If you use a UNIX workstation on a LAN with other UNIX workstations, it's a fair bet you will be using NFS to connect file systems together. The Solaris 2.5 system that I am running supports two versions of the NFS protocol. NFS Version 2 was implemented in 1984 and was first released with SunOS 2.0. It formed the basis of RFC 1094, dated March 1989. Version 3 was created in 1992, when a group of people from several companies got together to firm up the draft. After some pondering and discussion, the results were published for the world in June 1994 at the USENIX Conference in Boston. If you are wondering about Version 1 of NFS, well, that was internal to Sun and didn't escape.

Version 3, then, is the upstart newcomer, and is used between consenting machines. The choice of version is trans-

parent to the user of the system; if a machine cannot connect using Version 3, it will default back to Version 2.

The basic idea of NFS is simple. When you add a disk to the system on UNIX, you join it to the existing file system tree by the *mount* system call. The new disk forms a new branch of the tree structure. You can move into it using the `cd` command and access its files. With NFS, you do the same thing. You, as the client, issue a *mount* command that is sent to a remote server, and part of the file system tree on the server is joined to your local file store. The server will have a list of machines that are permitted to access its file store and will validate your request before passing back a "file handle."

On SunOS, the list is stored in a file called `/etc/exports`. On Solaris, the list is controlled by a call to the `share` command, and you'll find a set of these commands in `/etc/dfs/dfstab`. The file handle returned by the mount call is used in all further requests from the client to the server. Now, when a process on the client accesses a remote file with a read system call, for example, that system call is turned into a network request using the NFS protocol. The server checks the validity of the request, performs the desired operation and returns the result.

Once you have mounted the remote file system at some point in your file tree, whenever you attempt to open a file that is below the mount point, the system call that you emit will be translated into an NFS request and sent over the network to the server. The server will execute the request and will return a result to your kernel. In turn, your kernel will pass a system call result into your process as if the request was serviced by a local disk.

The NFS protocol assumes that the server does not retain any state about the client. For example, a normal UNIX read system call remembers how far a particular process has got in reading or writing a file. A sequence of read calls can be used to scan a file from beginning to end; there is no need to reposition file pointers before every read. NFS will store the "where we are now" state in the client, and when scanning a file, it's the job of the client to send appropriate read primitives, each containing position and size information.

So the server is not clever. It knows nothing about what the client is doing. Clients do tend to be clever, for efficiency reasons. They remember file positioning information so that they may present UNIX file semantics to the user process. They will also cache information so that it doesn't have to travel the network again.

It was an early aim of the NFS design that the remote file system should not be tied to UNIX, and that it would not be constrained to offer full UNIX file system semantics. So, for example, you cannot access a special device file on a remote system and expect that it will access the I/O device to which the special device interfaces. The NFS model was an "ideal" file system, with some UNIX overtones that I'll come to later. The aim of generating a system that would support different types of file system access paid off, I think. Sun must have sold a considerable number of PC-NFS licenses allowing PC systems to access shared file stores on UNIX servers.

An icon for the design was the notion of "statelessness" in the server. As we've seen, UNIX expects the kernel to retain state about the file that is open, the "where we are now" state. It was a great heresy to suggest that the server should maintain the state. At the time of the NFS design and early implementation, its competitor was the Remote File System (RFS) from AT&T, which aimed to provide a complete remote UNIX semantics, and did this by maintaining server state. Many loud debates, approaching the level of religious argument, ensued in public forums about the validity of the two approaches. For UNIX, RFS was probably the better approach, however, NFS won the contest by being an open standard and also because the world is not full of UNIX systems.

The NFS server is stateless. It's simply sent a transaction request and performs the operation. Each request is an independent event and, theoretically, file updates can occur in any order. Statelessness was really an original design criterion in NFS to avoid the need for crash recovery. When a server crashes, a client can just wait for it to come back and continue with operations as if nothing had happened. There is no state to be recovered and reloaded into the server. Cynics would say that it was a criterion because the early Sun systems were not exactly resilient: Because they tended to crash often, it was better to design something that avoided the recovery problem.

Stateless operation has a downside. File locking implies that state is retained on a file. This state needs to be kept close to the file–on the server–because several clients may be accessing the same file and need to see the locks. Getting file-locking operational on NFS took some time to implement. There is a separate lock manager to handle it.

UNIX Overtones

The NFS client/server has UNIX overtones because it adopted the UNIX file system tree model, an inevitable conclusion of its development. As a result, it doesn't easily supply file system features that UNIX doesn't support. For example, it doesn't provide support for file versioning, which might be needed to provide full remote support for a VMS file system. In general, people get by with this restriction. So far, no one has generated a widely used operating system whose file system differs wildly from the UNIX hierarchical structure.

POSIX and ANSI C have helped to sanctify the various values that a programmer might be expected to know about a file. These two standards rely heavily on UNIX, and so NFS has been able to interwork with several operating system clients that comply with the standards.

The details that are stored for each file in NFS map onto the traditional UNIX set of values. Each file has a set of file permissions that happen to be the same as the UNIX file permissions. NFS Version 2 doesn't support access control lists provided by other operating systems. The file permissions are applied to a user ID, group ID pair (UID, GID) that are also stored with the file. This pair of numbers needs to be the same for each user on your network. There's no translation mechanism to map local users on one machine to local users on another.

The need for (UID, GID) harmonization is one of the NFS features that I like the least. It is possible to provide a mapping scheme for the (UID, GID) pair, and non-UNIX servers sometimes do so. I well remember the nastiness of the renumbering operation that I had to perform on several machines to generate a single (UID, GID) space when the prospect of NFS loomed into view. The need to maintain a single local database of (UID, GID) pairs has given rise to systems that distribute databases around a site, known these days as NIS or NIS+.

These systems are really Band-Aids that don't solve the underlying problem. For one thing, the systems don't easily scale out from your local network to the wider real world. NFS really needs to provide for users to be local to a machine, but have shared access to files on other machines, and that access should be independent from the actual (UID, GID) values used on any particular machine.

You'll perhaps realize that something has to be done about superuser permissions, and this is one area where UID mapping does take place. Someone is superuser because the UID in the process that they are running is zero, which is what happens when they log in as root. When their process accesses a remote machine, it will tell the server that it is running with UID zero, which should allow them to have superuser privilege. However, the zero UID value is usually changed to the UID of the nobody user on the remote machine, meaning that they probably have less right to the files than a normal user.

Controls exist to map the UID to any value that seems sensible, and so it can be mapped back to the zero value. The conversion is controlled locally on the server, so the administration on a particular machine can decide which superusers on which external machines are permitted to have root access to their files.

NFS Version 3

So what does Version 3 give us? The Sun Answerbook doesn't help a lot. What follows is derived from the Boston USENIX paper (see below for details on the paper).

One of the big problems in Version 2 is the need for an NFS server to perform synchronous writes. When a client issues a write request, it sends an RPC call saying "write this data at such and such a position in the file." The server cannot reply to this RPC request saying "done" until the data is safely stored on magnetic media. If it says "OK" when it has it in memory but crashes before it manages to write it to disk, then the file will be in an inconsistent state because the client thinks that it has written some data that is not actually present on the disk. The server must do a "synchronous" write and not return any result until the data is safely stored on disk. The client has to wait until the write operation has completed, so runs in step with the speed of the network and the remote disk.

This has proved a huge bottleneck for NFS implementations. Some systems have provided an "unsafe" write mode where the data is retained in the memory of the server and the users hope that crashes are infrequent. To date, the most common solution for NFS servers to provide asynchronous writes has been the Prestoserve system, originally from Legato Systems Inc. Prestoserve is a kernel driver that uses an NVRAM disk cache sitting between the kernel and the disk. When the kernel initiates a write, the data is loaded into the cache and the kernel is told that the disk write has succeeded. The kernel tells the NFS server, and the server tells the client.

In time, the data finds its way onto the disk. If the system crashes, Prestoserve will write any unwritten data onto the disk before allowing the system to bootstrap, so the file system is consistent. The Prestoserve system also helps with normal local file system operations that require synchronous writes, for example, creating files.

Version 3 improves the write performance by allowing a client to choose to use asynchronous write transactions and later send a command that says "commit and write the data that you have to the disk." The operation is coupled with a "write verifier" value, an 8-bit number that is changed every time a server crashes. The idea is that the client can now determine whether a server has died and lost data that has been sent aimed at being written asynchronously.

Version 3 now allows NFS to operate using a TCP/IP connection to a remote machine rather than the previous UDP-based datagram system. The use of UDP goes back to the early obsession with transaction speed. The thinking was that TCP/IP imposed too much protocol overhead, and this would slow down NFS operation. The NFS designers then found that there was a need to supply many of the features of TCP/IP, such as reliability, error recovery, congestion control, timeouts and so on. The UDP code became a reimplementation of certain TCP/IP aspects, so why not just use TCP/IP in the first place? To be fair, the vast increase in processor speed since the original NFS design had made the TCP protocol overhead much less of a cycle stealer.

Version 3 provides some performance improvements by reducing the protocol overhead when returning directory information. There's a new primitive operation that returns a directory and the attributes of all the files it contains. The aim is to support the very common situation where a client will open a directory and then cycle through all the files that it contains looking for a file with a specific attribute. In NFS Version 2, this common operation sequence results in a flurry of network activity as the information for each individual file is retrieved one file at a time.

Version 3 provides some help for clients to maintain caches of information that is stored on the server. It's obviously a win if a file can be retained locally and supplied to the user processes without recourse to the network. Every reply for an operation that modifies data contains two sets of file attributes: a version taken before the operation was done and one taken after the operation. If the modification time in the preoperation attributes matches the client's copy, then the client knows that the local cache is consistent. Only the client has modified the file. The clients replace their local copy of the attributes with ones that existed after the operation and continue merrily along. If the preoperation times are different, then the client knows that its cached data is incorrect and can take steps to flush and reload it. This is a great improvement on the situation in Version 2 where a client was unsure who had altered a file with which it was dealing.

Finally, Version 3 supports 64-bit file pointers reflecting the change in our general expectation of the size of a large file. I can't say that I have a file on any of my systems that is bigger than 4,294,967,296 bytes, but undoubtedly some people are creating them.

As I said at the beginning, your Solaris 2.5+ system will use NFS Version 3 when it can, so most of these changes are invisible to users. Of course, the ability to interwork with older implementations was a design aim for Version 3.

Finally

This is another article where I intended writing about one thing and ended up talking about some subset of the topic. Never mind, another time. If you are interested in the NFS Version 3 paper, then you can find it in the *Proceedings of the USENIX Summer 1994 Technical Conference, Boston, MA,* ISBN 1-880446-62-6. I had hoped to say that you would find the paper online at the USENIX Web site (`http://www.usenix.org`) accessible only to members of the association, but sadly it's not there.

You'll find Samba at `http://samba.canberra.edu.au/pub/samba`. Another thing you will find on the Web are my son's Dark Forces Web pages: `http://www.hillside.co.uk/glyn/dark/dark.html`.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.