Why NFS Sucks

Olaf Kirch
SUSE/Novell Inc.
okir@suse.de

Abstract

This article will give an overview of NFS history, features and shortcomings, briefly compare it to other remote file systems, and present an outlook on future directions.

NFS is really *the* distributed file system in the Unix world – and at the same time it is probably also one of its most reviled components. For just about every Suse release, there's a bug in our bugzilla with a summary line of "NFS sucks". It even has a whole chapter of its own in the Unix Haters' Handbook. And having hacked quite a bit of NFS code over the course of 8 years, the author cannot help agreeing that NFS as a whole does have a number of warts.

This article is an attempt at answering why this is so. It will take a long look at some of the stranger features of NFS, why they came into existence, and how they affect stability, performance and POSIX conformance of the file system. The talk will also present some historical background, and compare NFS to other distributed file systems.

The author feels compelled to mention that this is not a complaint about the quality of the Linux NFS implementation, which is in fact pretty good.

1 History

One of the earliest networked file systems was RFS, the Remote File System included in SVR3. It was based on a concept called "Remote System Calls," where each system call was mapped directly to a call to the server system. This worked reasonably well, but was largely limited to SVR3 because of the SVR3 system call semantics.

Another problem with RFS was that it did not tolerate server crashes or reboots very well. Due to the design, the server had to keep a lot of state for every client, and, in fact, for every file opened by a client. This state could not be recovered after a server reboot, so when an RFS server went down, it usually took all its clients with it.

This early experience helps to understand some of the design decisions made in first NFS version developed by Sun in 1985. This was NFS version 2, and was first included in SunOS 2.0. Rumors have it that there was also a version 1, but it never got released to the world outside Sun.

NFSv2 attempted to address the shortcomings of RFS by making the server entirely stateless, and by defining a minimal set of *remote procedures* that provided a basic set of file system operations in a way that was a lot less operating system dependent than RFS. It also tried to be agnostic of the underlying file system, to a degree that it could be adapted to different Unix file systems with relative ease (doing the same for non-Unix file systems proved harder).

One of the shortcomings of NFSv2 was its lack of cache consistency. NFS makes no guarantees that all clients looking at a file or directory see exactly the same content at any given moment. Instead, each client sees a snapshot of a file's state from a hopefully not too distant past. NFS attempts to keep this snapshot in sync with the server, but if two clients operate on a single file simultaneously, changes made by one client usually do not become visible immediately on the other client.

In 1988, Spritely NFS was released, which extended the protocol to add a cache consistency mechanism to NFSv2. To achieve this, it sacrificed the server's statelessness, so that it was generally impossible for a client to recover from a server crash. Crash recovery for Spritely NFS wasn't added until 6 years later, in 1994.

At about the same time, NQNFS (the "Not-Quite NFS") was introduced in 4.4 BSD. NQNFS is a backward compatible protocol extension that adds the concept of leases to NFS, which is another mechanism to provide cache consistency. Unfortunately, NQNFS never gained wide acceptance outside the BSD camp.

In 1995, the specification of NFSv3 was published (written mostly by Rick Macklem, who also wrote NQNFS). NFSv3 includes several improvements over NFSv2, most of which can be categorized as performance enhancements. However, NFSv3 did not include any cache consistency mechanisms.

The year 1997 saw the publication of a standard called WebNFS, which was supposed to position NFS as an alternative to HTTP. It never gained any real following outside of Sun, and made a quiet exit after the Internet bubble burst.

The latest development in the NFS area is NFSv4, the first version of this standard was published in 2002. One of the major goals in the design of NFSv4 was to facilitate deployment over wide area networks (making it an "Internet Filesystem"), and to make the underlying file system model less Unix centric and provide better interoperability with Microsoft Windows. It is not an accident that the NFSv4 working group formed at about the same time as Microsoft started rebranding their SMB file sharing protocol as the "Common Internet File System."

2 NFS File Handles

One of the nice things about NFS is that it allows you to export very different types of file systems to the world. You're not stuck with a single file system implementation the way AFS does, for instance. NFS does not care if it is reiser, ext3 or XFS you export, a CD or a DVD.

A direct consequence of this is that NFS needs a fairly generic mechanism to identify the objects residing on a file system. This is what *file handles* are for. From the client's perspective, these are just opaque blobs of data, like a magic cookie. Only the server needs to understand the internal format of a file handle. In NFSv2, these handles were a fixed 32 bytes; NFSv3 makes them variable sized up to 64 bytes, and NFSv4 doubles that once more.

Another constraint is related to the statelessness paradigm: file handles must be *persistent*, i.e. when the server crashes and reboots, the file handles held by its clients must still be valid, so that the clients can continue whatever they were doing at that moment (e.g. writing to a file).

In the Unix world of the mid-80s and early 90s, a file handle merely represented an inode, and in fact in most implementations, the file handle just contained the device and inode number of the file it represented (plus some additional export identification we will ignore here). These handles went very well with the statelessness paradigm, as they remained valid across server reboots.

Unfortunately, this sort of mechanism does not work very well for all file systems; in fact, it is a fairly Unix centric thing to assume that files can be accessed by some static identifier, and without going through their file system path name. Not all file systems have a notion of a file independent from its path (the DOS and early Windows file systems kept the "inode" information inside the directory entry), and not all operating systems will operate on a disconnected inode. Also, the assumption that an inode number is sufficient to locate a file on disk was true with these older file systems, but that is no longer valid with more recent designs.

These assumptions can be worked around to some degree, but these workarounds do not come for free, and carry their own set of problems with them.

The easiest to fix is the inode number assumption – current Linux kernels allow file systems to specify a pair of functions that return a file handle for a given inode, and vice versa. This allows XFS, reiser and ext3 to have their own file handle representation without adding loads of ugly special case code to nfsd.

There is a second problem though, which proved much harder to solve. Some time in the 1.2 kernel or so, Linux introduced the concept of the directory cache, aka the *dcache*. An entry in the dcache is called a *dentry*, and represents the relation between a directory and one of its entries. The semantics of the dcache do not allow disconnected inode data floating around in the kernel; it requires that there is always a valid chain of dentries going from the root of the file system to the inode; and virtually all functions in the VFS layer expect a dentry as an argument instead of (or in addition to) the inode object they used to take.

This made things interesting for the NFS server, because the inode information is no longer sufficient to create something that the VFS layer is willing to operate on - now we also need a little bit of path information to reconstruct the dentry chain. For directories this is not hard, because each directory of a Unixish file system has a file named "`..`" that takes you to the parent directory. The NFS server simply has to walk up that chain until it hits the file system root. But for any other file system object, including regular files, there's no such thing, and thus the file handle needs to include an identifier for the parent directory as well.

This creates another interesting dilemma, which is that a file hard linked into several directories may be represented by different file handles, depending on the path it is accessed by. This is called *aliasing* and, depending on how well a client implementation handles this, may lead to inconsistencies in a client's attribute or data cache.

Even worse, a rename operation moving a file from one directory to another will invalidate the old file handle.

As an interesting twist, NFSv4 introduces the concept of volatile file handles. For these file handles, the server makes no promises about how long it will be good. At any time, the server may return an error code indicating to the client that it has to re-lookup the handle. It is not clear yet how well various NFSv4 are actually able to cope with this.

3 Write operations: As Slow as it Gets

Another problem with statelessness is how to prevent data loss or inconsistencies when the server crashes hard. For instance, a server may have acknowledged a client operation such as the creation of a file. If the server crashes before that change has been committed to disk, the client will never know, and it is in no position to replay the operation.

The way NFS solved this problem was to mandate that the server commits every change to disk before replying to the client. This is not that much of a problem for operations that usually happen infrequently, such as file creation or deletion. However, this requirement quickly becomes a major nuisance when writing large files, because each block sent to the server is written to disk separately, with the server waiting for the disk to do its job before it responds to the client.

Over the years, different ways to take the edge off this problem were devised. Several companies sold so-called “NFS accelerators," which was basically a card with a lot of RAM and a battery on it, acting as an additional, persistent cache between the VFS layer and the disk. Other approaches involved trying to flush several parallel writes in one go (also called write gathering). None of these solutions was entirely satisfactory, and therefore, virtually all NFS implementations provide an option for the administrator to turn off stable writes, trading performance for a (small) risk of data corruption or loss.

NFSv3 tries to improve this by introducing a new writing strategy, where clients send a large number of write operations that are not written to disk directly, followed by a “commit” call that flushes all pending writes to disk. This does afford a noticeable performance improvement, but unfortunately, it does not solve all problems.

On one hand, NFS clients are required to keep all dirty pages around until the server acknowledged the commit operation, beecause in case the server was rebooted, they need to replay all these write operations. This means, commit calls need to happen relatively frequently (once every few Megabytes). Second, a commit operation can become fairly costly – RAIDs usually like writes that cover one or more stripes, and it helps if the client is smart enough to align its writes in clusters of 128K or more. Second, some journaling file systems can have fairly big delays in sync operations. If there is a lot of write traffic, it is not uncommon for the NFS server to stall completely for several seconds because all of its threads service commit requests.

What’s more, some of the performance gain in using write/commit is owed to the fact that modern disk drives have internal write buffers, so that flushing data to the disk device really just sends data to the disk’s internal buffers, which is not sufficient for the type of guarantee NFS is trying to give. Forcing the block device to actually flush its internal write cache to disk incurs an additional delay.

4 NFS over UDP – Fragmentation

Another "interesting" feature of NFSv2 was that the original implementations supported only UDP as the transport protocol. NFS over TCP didn’t come into widespread use until the late 1990s.

There have been various issues with the use of UDP for NFS over the years. At one point, some operating system shipped with

UDP checksums turned off by default, presumably for performance reasons. Which is a rather bad thing to do if you're doing NFS over UDP, because you can easily end up with silent data corruption that you won't notice until it is way too late, and the last backup tape having a correct version of your precious file has been overwritten.

A more recent problem with UDP has to do with fragmentation. The lower bound for the NFS packet size that makes sense for reads and writes is given by the client's page size, which is 4096 for most architectures Linux runs on, and 8192 is a rather common choice these days. Unless you're using jumbograms (i.e. Ethernet frames of up to 9000 bytes), these packets get fragmented.

For those not familiar with IP fragmentation, here it is in a nutshell: if the sending system (or, in IPv4, any intermediate router) notices that an IP packet is too large for the network interface it needs to send this out to, it will break up the packet into several smaller pieces, each with a copy of the original IP header. In order so that the receiving system can tell which fragments go together, the sending system assigns each packet a 16bit identifier, the IPID. The receiver will lump all packets with matching source address, destination address and IPID into one *fragment chain*, and when it finds it has received all the pieces, it will stitch them together and hand them to the network stack for further processing. In case a fragment gets lost, there is a so-called *reassembly timeout*, defaulting to 30 seconds. If the fragment chain is not completed during that interval, it will simply be discarded.

The bad thing is, on today's network hardware, it is no big deal to send more than 65535 packets in under 30 seconds; in fact it is not uncommon for the IPID counter to wrap around in 5 seconds or less. Assume a packet A, containing an NFS READ reply is fragmented as say A_1, A_2, A_3, and fragment A_2 is lost. Then a few seconds later another NFS READ reply is transmitted, which receives the same IPID, and is being fragmented as B_1, B_2, B_3. The receiver will discard fragment B_1, because it already has a fragment chain for that IPID, and the part of the packet represented by B_1 is already there. Then it will receive B_2, which is exactly the piece of the puzzle that's missing, so it considers the fragment chain complete and reassembles a packet out of A_1, B_2, A_3.

Fortunately, the UDP checksum check will usually catch these botched reassemblies. But not all of them - it is just another 16bit quantity, so if the above happens a few thousand times, the probability of a matching checksum is decidedly non-zero. Depending on your hardware and test case, it is possible to reproduce silent data corruption within a few days or even a few hours.

Starting with kernel version 2.6.16, Linux has some code to protect from the ill side effects of IPID wraparound, by introducing some sort of sliding window of valid IPIDs. But that is really more of a band-aid than a real solution. The better approach is to use TCP instead, which avoids the problem entirely by not fragmenting at all.

5 Retransmitted Requests

As UDP is an unreliable protocol by design, NFS (or, more specifically, the RPC layer) needs to deal with packet loss. This creates all sorts of interesting problems, because we basically need to do all the things a reliable transport protocol does: retransmitting lost packets, flow control (if the NFS implementation supports sending several requests in parallel), and congestion avoidance. If you look at the RPC implementation in the Linux kernel, you will find a lot of things you may be familiar with from a TCP context, such as slow start, or estimators for round-trip times for more accurate timeouts.

One of the less widely known problems with NFS over UDP however affected the file system semantics. Consider a request to remove a directory, which the server dutifully performed and acknowledged. If the server's reply gets lost, the client will retransmit the request, which will fail unexpectedly because the directory it is supposed to remove no longer exists!

Requests that will fail if retransmitted are called *non-idempotent*. To prevent these from failing, a *request replay cache* was introduced in the NFS server, where replies to the most recent non-idempotent requests are cached. The NFS server identifies a retransmitted request by checking the reply cache for an entry with the same source address and port, and the same RPC transaction ID (also known as the *XID*, a 32bit counter).

This provides reasonable protection for NFS over UDP as long as the cache is big enough to hold replies for the client's maximum retransmit timeout. As of the 2.6.16 kernel, the Linux server's reply cache is rather too small, but there is work underway to rewrite it.

Interestingly, the reply cache is also useful when using TCP. TCP is not impacted the same way UDP is, since retransmissions are handled by the network transport layer. Still, TCP connections may break for various reasons, and the server may find the client retransmit a request after reconnecting.

There's a little twist to this story. The TCP protocol specification requires that the host breaking the connection does not reuse the same port number for a certain time (twice the *maximum segment lifetime*); this is also referred to as `TIME_WAIT` state. But usually you do not want to wait that long before reconnecting. That means the new TCP connection will originate from a different port, and the server will fail to find the retransmitted request in its cache.

To avoid that problem, the sunrpc code in recent Linux kernels works around this by using a little known method for disconnecting a TCP socket without going into `TIME_WAIT`, which allows it to reuse the same port immediately.

Strictly speaking, this is in violation of the TCP specification. While this avoids the problem with the reply cache, it remains to be seen whether this entails any negative side effects – for instance, how gracefully intermediate firewalls may deal with seeing SYN packets for a connection that they think ought to be in `TIME_WAIT`.

6 Cache Consistency

As mentioned in the first section, NFS makes no guarantees that all clients see exactly the same data at all times.

Of course, during normal operation, accessing a file will show you the content that is actually there, not some random gibberish. However, if two or more clients read and write the same file simultaneously, NFS makes no effort to propagate all changes to all clients immediately.

An NFS client is permitted to cache changes locally and send them to the server whenever it sees fit. This sort of lazy write-back greatly helps write performance, but the flip side is that everyone else will be blissfully unaware of these these change before they hit the server. To make things just a little harder, there is also no requirement for a client to transmit its cached write in any particular fashion, so dirty pages can (and often will be) written out in random order.

And even once the modified data arrives at the NFS server, not all clients will see this change immediately. This is because the NFS server does not keep track of who has a file open for reading and who does not (remember, we're stateless), so even if it wanted it cannot notify clients of such a change. Therefore, it is the client's job to do regular checks if its cached data is still valid.

So a client that has read the file once may continue to use its cached copy of the file until the next time it decides to check for a change. If that check reveals the file has changed, the client is required to discard any cached data and retrieve the current copy from the server.

The way an NFS client detects changes to a file is peculiar as well. Again, as NFS is stateless, there is no easy way to attach a monotonic counter or any other kind of versioning information to a file or directory. Instead, NFS clients usually store the file's modification time and size along with the other cache details. At regular intervals (usually somewhere between 3 to 60 seconds), it performs a so-called *cache revalidation*: The client retrieves the current set of file attributes from the server and compares the stored values to the current ones. If they match, it assumes the file hasn't changed and the cached data is still valid. If there is a mismatch, all cached data is discarded, and dirty pages are flushed to the server.

Unfortunately, most file systems store time stamps with second granularity, so clients will fail to detect any changes to a file if it happened within the same wall-clock second as their last revalidation. To compound the problem, NFS clients usually hold on to the data they have cached as long as they see fit. So once the cache is out of sync with the server, it will continue to show this invalid information until the data is evicted from the cache to make room, or until the file's modification time changes again and forces the client to invalidate its cache.

The only consistency guarantee made by NFS is called close-to-open consistency, which means that any changes made by you are flushed to the server on closing the file, and a cache revalidation occurs when you re-open it.

One can hardly fail to notice that there is a lot of handwaving in this sort of cache management. This model is adequate for environments where there is no concurrent read/write access by different clients on the same file, such as when exporting users' home directory, or a set of read-only data.

However, this fails badly when applications try to use NFS files concurrently, as some databases are known to do. This is simply not within the scope of the NFS standards, and while NFSv3 and NFSv4 do improve some aspects of cache consistency, these changes merely allow the client to cache more aggressively, but not necessarily more correctly. For instance, NFSv4 introduces the concept of delegations, which is basically a promise that the server will notify the client if some other host opens the file. This allows the client to cache all writes for as long as it holds the delegation; but after the server revokes it, everyone just falls back to the old NFSv3 behavior of mtime based cache revalidation.

There is no really good solution to this problem; all solutions so far either involve turning off caching to a fairly large degree, or extending the NFS protocol significantly.

Some documents recommend turning off caching entirely, by mounting the file system with the `noac` option, but this is really a desparate measure, because it kills performance completely.

Starting with the 2.6 kernel, the Linux NFS client supports `O_DIRECT` mode for file I/O, which turns off all read and write caching on a file descriptor. This is slightly better than using noac, as it still allows the caching of file attributes, but it means applications need to be modified and recompiled to use it. Its primary use is in the area of databases.

Another approach to force a file to show a consistent view across different clients is to use NFS file locking, because taking and releasing a lock acts as a cache synchronization point. In fact, in the Linux NFS client, the file unlock operation actually implies a cache invalidation – so this kind of synchronizyation is not exactly free of cost either.

Solutions involving changes to the NFS protocol include Spritely NFS and NQNFS; but these should probably considered as mostly research. It is questionable whether this gap in the NFS design will ever be addressed, or whether this is left for others to solve, such as OCFS2, GFS or Lustre.

7 POSIX Conformance

People writing applications usually expect the file system to "just work," and will get slightly upset if their application behaves differently on NFS than it does on a local file system. Of course, everyone will have a slightly different idea of what "just works" really is, but the POSIX standard is a reasonable approximation.

NFS never claimed to be fully POSIX compliant, and given its rather liberal cache consistency guarantees, it never will. But still, it attempts to conform to the standard as much as possible.

Some of the gymnastics NFS needs to go through in order to do so can be considered just funny when you look at them. For instance, consider the utimes call, which can be used by an application to set a file's modification time stamp. On some kernels, the command cp -p would not preserve the time stamp when copying files to NFS. The reason is the NFS write cache, which usually does not get flushed until the file is closed. The way cp -p does its job is by creating the output file and writing all data first; then it calls utimes to set the modification time stamp, and then closes the file. Now close would see that there were still pending writes, and flush them out to the server, clobbering the file's mtime as a result. The only viable fix for this is to make sure the NFS client flushes all dirty pages before performing the utimes update – in other words, utimes implies a fsync.

Some other cases are a bit stranger. One such case is the ability to write to an open unlinked file. POSIX says an application can open a file for reading and writing, unlink it, and continue to do I/O on it. The file is not supposed to go away until the last application closes it.

This is difficult to do over NFS, since traditionally, the NFS server has no concept of "open" files (this was added in NFSv4, however). So when a client removes a file, it will be gone for good, and the file handle is no longer valid – and and attempt to read from or write to that file will result in a “Stale file handle” error.

The way NFS traditionally kludges around this is by doing what has been dubbed a “silly rename.” When the NFS client notices during an unlink call that one or more applications still hold an open file descriptor to this file, it will not send a REMOVE call to the server. Instead, it will rename the file to some temporary file name, usually .nfsXXX where XXX is some hex number. This file will stay around until the last application closes its open file descriptor, and only then will the NFS client send the final REMOVE call to the server that gets rid of this renamed file.

This sounds like a rather smart sleight of hand, and it is – up to a point. First off, this does not work across different clients. But that should not come as a surprise given the lack of cache consistency.

Things get outright weird though if you consider what happens when someone tries to unlink such a `.nfsXXX` file. The Linux client does not allow this, in order to maintain POSIX semantics as much as possible. The undesirable side effect of this is that a `rm -rf` call will fail to remove a directory if it contains a file that is currently open to some application.

But the weirdest part of POSIX conformance is probably the handling of access control lists.

8 Access Control Lists

The POSIX.1e working group proposed a set of operating system primitives that were supposed to enhance the Unix security model. Their work was never finished, but they did create a legacy that kind of stuck – capabilities and access control lists (ACLs) being the promiment examples of their work.

Neither NFSv2 nor NFSv3 included support for ACLs in their design. When NFSv2 was designed, ACLs and mandatory access control were more or less an academic issue in the Unix world, so they were simply not part of the specification's scope.

When NFSv3 was designed, ACLs were already being used more or less widely, and acknowledging that fact, a new protocol operation named `ACCESS` was introduced, which lets the client query a user's permissions to perform a certain operation. This at least allows a client to perform the correct access decisions in the presence of access control lists on the server.

However, people who use ACLs usually want to be able to view and modify them, too, without having to log on to the server machine. NFS protocol versions 2 and 3 do not provide any mechanisms for queries or updates of ACLs at all, so different vendors devised their own side-band protocols that added this functionality. These are usually implemented as additional RPC programs available on the same port as the NFS server itself. According to various sources, there were at least four different ACL protocols, all of them mutually incompatible. So an SGI NFS client could do ACLs when talking to an SGI NFS server, or a Solaris client could do the same when talking to a Solaris server.

Over the course of a few years, it seems the Solaris ACL protocol has become the prevalent standard, if just by virtue of eliminating most of the competition. The Linux ACL implementation adopted this protocol as well.

NFSv4 adds support for access control lists. But in its attempt to be a cross-platform distributed file system, it adopted not the POSIX ACL model, but invented its own ACLs which are much closer to the Windows ACL model (which has richer semantics) than to the POSIX model. It's not entirely compatible with Windows ACLs either, though.

The result of this is that it is not really easy to do POSIX ACLs over NFSv4 either: there is a mapping of POSIX to NFSv4 ACLs, but it is not really one-to-one, and somewhat awkward.

The ironic part of the story is that Sun, which was one of the driving forces behind the NFSv4 standard, added an NFSv4 version to their ACL side band protocol which allows querying and updating of POSIX ACLs, without having to translate them to NFSv4 ACLs and back.

9 NFS Security

One of the commonly voiced complaints over NFS is the weak security model of the underlying RPC transport. And indeed, security has never been one of its strong points.

The default authentication mechanism in RPC is `AUTH_SYS`, also known as `AUTH_UNIX` because it basically conveys Unix style credentials, including user and group ID, and a list of supplementary groups the user is in. However, the server has no way to verify these credentials, it can either trust the client, or map all user and group IDs to some untrusted account (such as `nobody`).

Stronger security flavors for RPC have been around for a while, such as Sun's "Secure RPC," which was based on a Diffie-Hellman key management scheme and DES cryptography to validate a user's identity. Another security flavor that was used in some places relied on Kerberos 4 credentials. Both of them provided only a modicum of security however, as the credentials were not tied in any way to the packet payload, so that attackers could intercept a packet with valid credentials and massage the NFS request to do their own nefarious biddings. Moreover, the lack of high-resolution timers on average 1980s hardware meant that most clients would often generate several packets with identical time stamps; so the server had to accept these as legitimate – opening the door to replay attacks.

A few years ago, a new RPC authentication flavor based on GSSAPI was defined and standardized; it provides different levels of security, ranging from the old-style sort of authentication restricted to the RPC header, to integrity and/or privacy. And since GSSAPI is agnostic of the underlying security system, this authentication mechanism can be used to integrate NFS security with any security system that provides a GSSAPI binding.

The current Linux implementation of `RPCSEC_GSS` was developed as part of the NFSv4 project. It currently supports Kerberos 5, but work is underway to extend it to SPKM-3 and LIPKEY.

It is worth noting that GSS authentication is not an exclusive feature of NFSv4, it can be enabled separately of NFSv4, and can be used with older versions of the protocol as well. On the other hand, there remains some doubt as to whether there is really such a huge demand for stronger NFS security, despite the vocal criticism. Secure RPC was not perfect, but it has been available for ages on many platforms, and unlike Kerberos it was rather straightforward to deploy. Still there weren't that many site that seriously made use of it.

10 NFS File Locking

Another operation that was not in the scope of the original NFS specification is file locking. Nobody has put forth an explanation why that was so.

At some point, NFS engineers at Sun recognized that it would be very useful to be able to do distributed file locking, especially given the cache consistency semantics of the NFS protocol.

Subsequently, another side-band protocol called the *Network Lock Manager* (NLM for short) protocol was devised, which implements lock and unlock operations, as well as the ability to notify a client when a previously blocked lock could be granted. Traditionally, NLM requests are handled by the `lockd` service, which used to be a user land daemon, but was moved into the kernel later.

NLM has a number of shortcomings. Probably the most glaring one is that it was designed for POSIX locks only; BSD `flock` locks are not supported, since they have somewhat different semantics. It is possible to emulate these with NLM, but it's non-trivial, and so far few implementations do this.

Another shortcoming is that most implementations do not bother with using any kind of RPC security with NLM requests, so that a `lockd` implementation has no choice but to accept unauthenticated requests, at least as long as it wants to interoperate with other operating systems.

Now, file locking is inherently a stateful operation, which doesn't go together well with the statelessness paradigm of the NFS protocol. In order to address this, mechanisms for lock reclaim were added to the NLM – if a NFS server reboots, there is a so-called *grace period* during which clients can re-register all the locks they were holding with the server.

Obviously, in order to make this work, clients need to be notified when a server reboots. For this, yet another side-band protocol was designed, called *Network Status Monitor* or NSM. Calling it a status monitor is a bit of a misnomer, as this is purely a reboot notification service. NSM doesn't use any authentication either, and it its specification is a bit vague on how to identify hosts – either by address, which creates issues with multi-homed hosts, or by name, which requires that all machines have proper host names configured, and proper entries in the DNS (which surprisingly often is not the case).

NFSv4 does a lot better in this area, by finally integrating file locking into the protocol.

11 AFS

AFS, the Andrew File System, was originally developed jointly by Carnegie Mellon University and IBM. It was probably never a huge success outside academia and research installations, despite the fact that the Open Group made it the basis of the distributed file system for DCE (and charged an arm and a leg for it). Late in its life cycle, it was released by IBM under an open source license, which managed to breathe a little life back into it.

AFS is a very featureful distributed file system. Among other things, it provides good security through the use of Kerberos 4, location independent naming, and supports migration and replication. On the down side, it comes with its own server side storage file system, so that you cannot simply export your favorite journaling file system over AFS. Code portability, especially to 64bit platforms, and the sort of `#ifdef` accretion that can occur over the course of 20 years is also an issue.

12 CIFS

CIFS, the *Common Internet File System*, is what was colloquially referred to as SMBfs some time ago. Microsoft's distributed file system is session-based, and sticks closely to the file system semantics of windows file systems. Samba, and the Linux smbfs and cifs clients have demonstrated that it is possible for Unix platforms to interoperate with Windows machines using CIFS, but some things from the POSIX world remain hard to map to their Windows equivalents and vice versa, with Access Control Lists (ACLs) being the most notorious example.

CIFS provides some cache consistency through the use of op-locks. It is a stateful protocol, and crash recovery is usually the job of the application (we're probably all familiar with Abort/Retry/Ignore dialog boxes).

While CIFS was originally designed purely with Windows file system semantics in mind, it provides a protocol extension mechanisms which can be used to implement support for some POSIX concepts that cannot be mapped onto the CIFS model. This mechanism has been used successfully by the Samba team to provide better Linux to Linux operation over CIFS.

The Linux 2.6 kernel comes with a new CIFS implementation that is well along the way of replacing the old smbfs code. As of this writing, the cifs client seems to have overcome most of its initial stability issues, and a while it is still missing a few features, it looks very promising.

Without question, CIFS is the de-facto standard when it comes to interoperating with Windows machines. However, CIFS could be serious competition to NFS in the Linux world, too – the biggest obstacle in this arena is not a technical one, however, but the fact that it is is controlled entirely by Microsoft, who like to spring the occasional surprise or two on the open source world.

13 Cluster Filesystems

Another important area of development in the world of distributed file systems are clustered file systems such as Lustre, GFS and OCFS2. Especially the latter looks very interesting, as its kernel component is relatively small and seems to be well-designed.

Cluster file systems are currently no replacement for file systems such as NFS or CIFS, because they usually require a lot more in terms of infrastructure. Most of them do not scale very well beyond a few hundred nodes either.

14 Future NFS trends

The previous sections have probably made it abundantly clear that NFS is far from being the perfect distributed file system. Still, in the Linux-to-Linux networking world, it is currently the best we have, despite all its shortcomings.

It will be interesting to see if it will continue to play an important role in this area, or if it will be pushed aside by other distributed file systems.

Without doubt, NFSv4 will see wide-spread use in maybe a year from now. However, one should remain sceptical on whether it will actually meet its original goal of providing interoperability with the Windows world. Not because of any design shortcomings, but simply because CIFS is doing this already, and seems to be doing its job quite well. In the long term, it may be interesting to see if CIFS can take some bites out of the NFS pie. The samba developers certainly think so.

It is also an open question whether there is much incentive for end users to switch to NFSv4. In the operational area, protocol semantics haven't changed much; they mostly got more complex. If users get any benefits from NFSv4, it may not be from things like Windows interoperability, but from other new features of the protocol, such as support for replication and migration. It is worth noting, however, that while the NFSv4 RFC provides the hooks for informing clients about migration of a file system, it does not define the migration mechanisms themselves.

The adoption of `RPCSEC_GSS` will definitely be a major benefit in terms of security. While GSS with Kerberos may not see wide deployment, simply because of the administrative overhead of running a Kerberos service, other

GSS mechanisms such as LIPKEY may provide just the right trade-off between security and ease of use that make them worthwhile to small to medium sized networks.

Other interesting areas of NFS development in Linux include the RPC transport switch, which allows the RPC layer to use transports other than UDP and TCP over IPv4. The primary goals in this area are NFS over IPv6, and using Infiniband/RDMA as a transport.