The 4.4BSD NFS Implementation

Rick Macklem
University of Guelph

ABSTRACT

The 4.4BSD implementation of the Network File System (NFS)[1] is intended to interoperate with other NFS Version 2 Protocol (RFC1094) implementations but also allows use of an alternate protocol that is hoped to provide better performance in certain environments. This paper will informally discuss these various protocol features and their use. There is a brief overview of the implementation followed by several sections on various problem areas related to NFS and some hints on how to deal with them.

Not Quite NFS (NQNFS) is an NFS like protocol designed to maintain full cache consistency between clients in a crash tolerant manner. It is an adaptation of the NFS protocol such that the server supports both NFS and NQNFS clients while maintaining full consistency between the server and NQNFS clients. It borrows heavily from work done on Spritely-NFS [Srinivasan89], but uses Leases [Gray89] to avoid the need to recover server state information after a crash.

1. NFS Implementation

The 4.4BSD implementation of NFS and the alternate protocol nicknamed Not Quite NFS (NQNFS) are kernel resident, but make use of a few system daemons. The kernel implementation does not use an RPC library, handling the RPC request and reply messages directly in *mbuf* data areas. NFS interfaces to the network using sockets via. the kernel interface available in *sys/kern/uipc_syscalls.c* as *sosend(), soreceive(),...* There are connection management routines for support of sockets for connection oriented protocols and timeout/retransmit support for datagram sockets on the client side. For connection oriented transport protocols, such as TCP/IP, there is one connection for each client to server mount point that is maintained until an umount. If the connection breaks, the client will attempt a reconnect with a new socket. The client side can operate without any daemons running, but performance will be improved by running nfsiod daemons that perform read-aheads and write-behinds. For the server side to function, the daemons portmap, mountd and nfsd must be running. The mountd daemon performs two important functions.

1) Upon startup and after a hangup signal, mountd reads the exports file and pushes the export information for each local file system down into the kernel via. the mount system call.

2) Mountd handles remote mount protocol (RFC1094, Appendix A) requests.

The nfsd master daemon forks off children that enter the kernel via. the nfssvc system call. The children normally remain kernel resident, providing a process context for the NFS RPC servers. The only exception to this is when a Kerberos [Steiner88] ticket is received and at that time the nfsd exits the kernel temporarily to verify the ticket via. the Kerberos libraries and then returns to the kernel with the results. (This only happens for Kerberos mount points as described further under Security.) Meanwhile, the master nfsd waits to accept new connections from clients using connection oriented transport protocols and passes the new sockets down into the kernel. The client side mount_nfs along with portmap and mountd are the only parts of the NFS subsystem that make any use of the Sun RPC library.

2. Mount Problems

There are several problems that can be encountered at the time of an NFS mount, ranging from a unresponsive NFS server (crashed, network partitioned from client, etc.) to various interoperability problems between different NFS implementations.

[1]Network File System (NFS) is believed to be a registered trademark of Sun Microsystems Inc.

On the server side, if the 4.4BSD NFS server will be handling any PC clients, mountd will require the **-n** option to enable non-root mount request servicing. Running of a pcnfsd[2] daemon will also be necessary. The server side requires that the daemons mountd and nfsd be running and that they be registered with portmap properly. If problems are encountered, the safest fix is to kill all the daemons and then restart them in the order portmap, mountd and nfsd. Other server side problems are normally caused by problems with the format of the exports file, which is covered under Security and in the exports man page.

On the client side, there are several mount options useful for dealing with server problems. In cases where a file system is not critical for system operation, the **-b** mount option may be specified so that mount_nfs will go into the background for a mount attempt on an unresponsive server. This is useful for mounts specified in *fstab(5)*, so that the system will not get hung while booting doing **mount -a** because a file server is not responsive. On the other hand, if the file system is critical to system operation, this option should not be used so that the client will wait for the server to come up before completing bootstrapping. There are also three mount options to help deal with interoperability issues with various non-BSD NFS servers. The **-P** option specifies that the NFS client use a reserved IP port number to satisfy some servers' security requirements.[3] The **-c** option stops the NFS client from doing a *connect* on the UDP socket, so that the mount works with servers that send NFS replies from port numbers other than the standard 2049.[4] Finally, the **-g=***num* option sets the maximum size of the group list in the credentials passed to an NFS server in every RPC request. Although RFC1057 specifies a maximum size of 16 for the group list, some servers can't handle that many. If a user, particularly root doing a mount, keeps getting access denied from a file server, try temporarily reducing the number of groups that user is in to less than 5 by editing /etc/group. If the user can then access the file system, slowly increase the number of groups for that user until the limit is found and then peg the limit there with the **-g=***num* option. This implies that the server will only see the first *num* groups that the user is in, which can cause some accessibility problems.

For sites that have many NFS servers, amd [Pendry93] is a useful administration tool. It also reduces the number of actual NFS mount points, alleviating problems with commands such as df(1) that hang when any of the NFS servers is unreachable.

3. Dealing with Hung Servers

There are several mount options available to help a client deal with being hung waiting for response from a crashed or unreachable[5] server. By default, a hard mount will continue to try to contact the server ''forever'' to complete the system call. This type of mount is appropriate when processes on the client that access files in the file system do not tolerate file I/O systems calls that return -1 with *errno == EINTR* and/or access to the file system is critical for normal system operation.

There are two other alternatives:

1) A soft mount (**-s** option) retries an RPC *n* times and then the corresponding system call returns -1 with errno set to EINTR. For TCP transport, the actual RPC request is not retransmitted, but the timeout intervals waiting for a reply from the server are done in the same manner as UDP for this purpose. The problem with this type of mount is that most applications do not expect an EINTR error return from file I/O system calls (since it never occurs for a local file system) and get confused by the error return from the I/O system call. The option **-x=***num* is used to set the RPC retry limit and if set too low, the error returns will start occurring whenever the NFS server is slow due to heavy load. Alternately, a large retry limit can result in a process hung for a long time, due to a crashed server or network partitioning.

2) An interruptible mount (**-i** option) checks to see if a termination signal is pending for the process when waiting for server response and if it is, the I/O system call posts an EINTR. Normally this results in the process being terminated by the signal when returning from the system call. This

[2] Pcnfsd is available in source form from Sun Microsystems and many anonymous ftp sites.

[3] Any security benefit of this is highly questionable and as such the BSD server does not require a client to use a reserved port number.

[4] The Encore Multimax is known to require this.

[5] Due to a network partitioning or similar.

feature allows you to ‘‘^C’’ out of processes that are hung due to unresponsive servers. The problem with this approach is that signals that are caught by a process are not recognized as termination signals and the process will remain hung.[6]

4. RPC Transport Issues

The NFS Version 2 protocol runs over UDP/IP transport by sending each Sun Remote Procedure Call (RFC1057) request/reply message in a single UDP datagram. Since UDP does not guarantee datagram delivery, the Remote Procedure Call (RPC) layer times out and retransmits an RPC request if no RPC reply has been received. Since this round trip timeout (RTO) value is for the entire RPC operation, including RPC message transmission to the server, queuing at the server for an nfsd, performing the RPC and sending the RPC reply message back to the client, it can be highly variable for even a moderately loaded NFS server. As a result, the RTO interval must be a conservation (large) estimate, in order to avoid extraneous RPC request retransmits.[7] Also, with an 8Kbyte read/write data size (the default), the read/write reply/request will be an 8+Kbyte UDP datagram that must normally be fragmented at the IP layer for transmission.[8] For IP fragments to be successfully reassembled into the IP datagram at the receive end, all fragments must be received within a fairly short ‘‘time to live’’. If one fragment is lost/damaged in transit, the entire RPC must be retransmitted and redone. This problem can be exaggerated by a network interface on the receiver that cannot handle the reception of back to back network packets. [Kent87a]

There are several tuning mount options on the client side that can prove useful when trying to alleviate performance problems related to UDP RPC transport. The options **-r=***num* and **-w=***num* specify the maximum read or write data size respectively. The size *num* should be a power of 2 (4K, 2K, 1K) and adjusted downward from the maximum of 8Kbytes whenever IP fragmentation is causing problems. The best indicator of IP fragmentation problems is a significant number of *fragments dropped after timeout* reported by the *ip:* section of a **netstat -s** command on either the client or server. Of course, if the fragments are being dropped at the server, it can be fun figuring out which client(s) are involved. The most likely candidates are clients that are not on the same local area network as the server or have network interfaces that do not receive several back to back network packets properly.

By default, the 4.4BSD NFS client dynamically estimates the retransmit timeout interval for the RPC and this appears to work reasonably well for many environments. However, the **-d** flag can be specified to turn off the dynamic estimation of retransmit timeout, so that the client will use a static initial timeout interval.[9] The **-t=***num* option can be used with **-d** to set the initial timeout interval to other than the default of 2 seconds. The best indicator that dynamic estimation should be turned off would be a significant number[10] in the *X Replies* field and a large number in the *Retries* field in the *Rpc Info:* section as reported by the **nfsstat** command. On the server, there would be significant numbers of *Inprog* recent request cache hits in the *Server Cache Stats:* section as reported by the **nfsstat** command, when run on the server.

The tradeoff is that a smaller timeout interval results in a better average RPC response time, but increases the risk of extraneous retries that in turn increase server load and the possibility of damaged files on the server. It is probably best to err on the safe side and use a large (>= 2sec) fixed timeout if the dynamic retransmit timeout estimation seems to be causing problems.

An alternative to all this fiddling is to run NFS over TCP transport instead of UDP. Since the 4.4BSD TCP implementation provides reliable delivery with congestion control, it avoids all of the above problems. It also permits the use of read and write data sizes greater than the 8Kbyte limit for UDP tran-

---

[6] Unfortunately, there are also some resource allocation situations in the BSD kernel where the termination signal will be ignored and the process will not terminate.

[7] At best, an extraneous RPC request retransmit increases the load on the server and at worst can result in damaged files on the server when non-idempotent RPCs are redone [Juszczak89].

[8] 6 IP fragments for an Ethernet, which has an maximum transmission unit of 1500bytes.

[9] After the first retransmit timeout, the initial interval is backed off exponentially.

[10] Even 0.1% of the total RPCs is probably significant.

sport.[11] NFS over TCP usually delivers comparable to significantly better performance than NFS over UDP unless the client or server processor runs at less than 5-10MIPS. For a slow processor, the extra CPU overhead of using TCP transport will become significant and TCP transport may only be useful when the client to server interconnect traverses congested gateways. The main problem with using TCP transport is that it is only supported between BSD clients and servers.[12]

5. Other Tuning Tricks

Another mount option that may improve performance over certain network interconnects is **-a=***num* which sets the number of blocks that the system will attempt to read-ahead during sequential reading of a file. The default value of 1 seems to be appropriate for most situations, but a larger value might achieve better performance for some environments, such as a mount to a server across a ‘‘high bandwidth * round trip delay’’ interconnect.

For the adventurous, playing with the size of the buffer cache can also improve performance for some environments that use NFS heavily. Under some workloads, a buffer cache of 4-6Mbytes can result in significant performance improvements over 1-2Mbytes, both in client side system call response time and reduced server RPC load. The buffer cache size defaults to 10% of physical memory, but this can be overridden by specifying the BUFPAGES option in the machine’s config file.[13] When increasing the size of BUFPAGES, it is also advisable to increase the number of buffers NBUF by a corresponding amount. Note that there is a tradeoff of memory allocated to the buffer cache versus available for paging, which implies that making the buffer cache larger will increase paging rate, with possibly disastrous results.

6. Security Issues

When a machine is running an NFS server it opens up a great big security hole. For ordinary NFS, the server receives client credentials in the RPC request as a user id and a list of group ids and trusts them to be authentic! The only tool available to restrict remote access to file systems with is the exports(5) file, so file systems should be exported with great care. The exports file is read by mountd upon startup and after a hangup signal is posted for it and then as much of the access specifications as possible are pushed down into the kernel for use by the nfsd(s). The trick here is that the kernel information is stored on a per local file system mount point and client host address basis and cannot refer to individual directories within the local server file system. It is best to think of the exports file as referring to the various local file systems and not just directory paths as mount points. A local file system may be exported to a specific host, all hosts that match a subnet mask or all other hosts (the world). The latter is very dangerous and should only be used for public information. It is also strongly recommended that file systems exported to ‘‘the world’’ be exported read-only. For each host or group of hosts, the file system can be exported read-only or read/write. You can also define one of three client user id to server credential mappings to help control access. Root (user id == 0) can be mapped to some default credentials while all other user ids are accepted as given. If the default credentials for user id equal zero are root, then there is essentially no remapping. Most NFS file systems are exported this way, most commonly mapping user id == 0 to the credentials for the user nobody. Since the client user id and group id list is used unchanged on the server (except for root), this also implies that the user id and group id space must be common between the client and server. (ie. user id N on the client must refer to the same user on the server) All user ids can be mapped to a default set of credentials, typically that of the user nobody. This essentially gives world access to all users on the corresponding hosts.

There is also a non-standard BSD **-kerb** export option that requires the client provide a KerberosIV rcmd service ticket to authenticate the user on the server. If successful, the Kerberos principal is looked up in the server’s password and group databases to get a set of credentials and a map of client userid to these credentials is then cached. The use of TCP transport is strongly recommended, since the scheme depends on the TCP connection to avert replay attempts. Unfortunately, this option is only usable between BSD

[11]Read/write data sizes greater than 8Kbytes will not normally improve performance unless the kernel constant MAXBSIZE is increased and the file system on the server has a block size greater than 8Kbytes.

[12]There are rumors of commercial NFS over TCP implementations on the horizon and these may well be worth exploring.

BUFPAGES is the number of physical machine pages allocated to the buffer cache. ie. BUFPAGES * NBPG = buffer cache size in bytes clients and servers since it is not compatible with other known ‘‘kerberized’’ NFS systems. To enable use of this Kerberos option, both mount_nfs on the client and nfsd on the server must be rebuilt with the -DKERBEROS option and linked to KerberosIV libraries. The file system is then exported to the client(s) with the **-kerb** option in the exports file on the server and the client mount specifies the **-K** and **-T** options. The **-m=***realm* mount option may be used to specify a Kerberos Realm for the ticket (it must be the Kerberos Realm of the server) that is other than the client's local Realm. To access files in a **-kerb** mount point, the user must have a valid TGT for the server's Realm, as provided by kinit or similar.

As well as the standard NFS Version 2 protocol (RFC1094) implementation, BSD systems can use a variant of the protocol called Not Quite NFS (NQNFS) that supports a variety of protocol extensions. This protocol uses 64bit file offsets and sizes, an *access rpc*, an *append* option on the write rpc and extended file attributes to support 4.4BSD file system functionality more fully. It also makes use of a variant of short term *leases* [Gray89] with delayed write client caching, in an effort to provide full cache consistency and better performance. This protocol is available between 4.4BSD systems only and is used when the **-q** mount option is specified. It can be used with any of the aforementioned options for NFS, such as TCP transport (**-T**) and KerberosIV authentication (**-K**). Although this protocol is experimental, it is recommended over NFS for mounts between 4.4BSD systems.[14]

7. Monitoring NFS Activity

The basic command for monitoring NFS activity on clients and servers is nfsstat. It reports cumulative statistics of various NFS activities, such as counts of the various different RPCs and cache hit rates on the client and server. Of particular interest on the server are the fields in the *Server Cache Stats:* section, which gives numbers for RPC retries received in the first three fields and total RPCs in the fourth. The first three fields should remain a very small percentage of the total. If not, it would indicate one or more clients doing retries too aggressively and the fix would be to isolate these clients, disable the dynamic RTO estimation on them and make their initial timeout interval a conservative (ie. large) value.

On the client side, the fields in the *Rpc Info:* section are of particular interest, as they give an overall picture of NFS activity. The *TimedOut* field is the number of I/O system calls that returned -1 for ‘‘soft’’ mounts and can be reduced by increasing the retry limit or changing the mount type to ‘‘intr’’ or ‘‘hard’’. The *Invalid* field is a count of trashed RPC replies that are received and should remain zero.[15] The *X Replies* field counts the number of repeated RPC replies received from the server and is a clear indication of a too aggressive RTO estimate. Unfortunately, a good NFS server implementation will use a ‘‘recent request cache’’ [Juszczak89] that will suppress the extraneous replies. A large value for *Retries* indicates a problem, but it could be any of:

- a too aggressive RTO estimate
- an overloaded NFS server
- IP fragments being dropped (gateway, client or server)

and requires further investigation. The *Requests* field is the total count of RPCs done on all servers.

The **netstat -s** comes in useful during investigation of RPC transport problems. The field *fragments dropped after timeout* in the *ip:* section indicates IP fragments are being lost and a significant number of these occurring indicates that the use of TCP transport or a smaller read/write data size is in order. A significant number of *bad checksums* reported in the *udp:* section would suggest network problems of a more generic sort. (cabling, transceiver or network hardware interface problems or similar)

There is a RPC activity logging facility for both the client and server side in the kernel. When logging is enabled by setting the kernel variable nfsrtton to one, the logs in the kernel structures nfsrtt (for the client side) and nfsdrt (for the server side) are updated upon the completion of each RPC in a circular manner. The pos element of the structure is the index of the next element of the log array to be updated. In other words, elements of the log array from *log*[pos] to *log*[pos - 1] are in chronological order. The include

---

[14] I would appreciate email from anyone who can provide NFS vs. NQNFS performance measurements, particularly fast clients, many clients or over an internetwork connection with a large ‘‘bandwidth * RTT’’ product.

[15] Some NFS implementations run with UDP checksums disabled, so garbage RPC messages can be received.

file <sys/nfsrtt.h> should be consulted for details on the fields in the two log structures.[16]

8. Diskless Client Support

The NFS client does include kernel support for diskless/dataless operation where the root file system and optionally the swap area is remote NFS mounted. A diskless/dataless client is configured using a version of the ‘‘swapvmunix.c’’ file as provided in the directory *contrib/diskless.nfs*. If the swap device == NODEV, it specifies an NFS mounted swap area and should be configured the same size as set up by diskless_setup when run on the server. This file must be put in the *sys/compile/<machine_name>* kernel build directory after the config command has been run, since config does not know about specifying NFS root and swap areas. The kernel variable mountroot must be set to nfs_mountroot instead of ffs_mountroot and the kernel structure nfs_diskless must be filled in properly. There are some primitive system administration tools in the *contrib/diskless.nfs* directory to assist in filling in the nfs_diskless structure and in setting up an NFS server for diskless/dataless clients. The tools were designed to provide a bare bones capability, to allow maximum flexibility when setting up different servers.

The tools are as follows:

- diskless_offset.c - This little program reads a ‘‘vmunix’’ object file and writes the file byte offset of the nfs_diskless structure in it to standard out. It was kept separate because it sometimes has to be compiled/linked in funny ways depending on the client architecture. (See the comment at the beginning of it.)
- diskless_setup.c - This program is run on the server and sets up files for a given client. It mostly just fills in an nfs_diskless structure and writes it out to either the "vmunix" file or a separate file called /var/diskless/setup.<official-hostname>
- diskless_boot.c - There are two functions in here that may be used by a bootstrap server such as tftpd to permit sharing of the ‘‘vmunix’’ object file for similar clients. This saves disk space on the bootstrap server and simplify organization, but are not critical for correct operation. They read the ‘‘vmunix’’ file, but optionally fill in the nfs_diskless structure from a separate "setup.<official-hostname>" file so that there is only one copy of "vmunix" for all similar (same arch etc.) clients. These functions use a text file called /var/diskless/boot.<official-hostname> to control the netboot.

The basic setup steps are:

- make a "vmunix" for the client(s) with mountroot() == nfs_mountroot() and swdevt[0].sw_dev == NODEV if it is to do nfs swapping as well (See the same swapvmunix.c file)
- run diskless_offset on the vmunix file to find out the byte offset of the nfs_diskless structure
- Run diskless_setup on the server to set up the server and fill in the nfs_diskless structure for that client. The nfs_diskless structure can either be written into the vmunix file (the -x option) or saved in /var/diskless/setup.<official-hostname>.
- Set up the bootstrap server. If the nfs_diskless structure was written into the ‘‘vmunix’’ file, any vanilla bootstrap protocol such as bootp/tftp can be used. If the bootstrap server has been modified to use the functions in diskless_boot.c, then a file called /var/diskless/boot.<official-hostname> must be created. It is simply a two line text file, where the first line is the pathname of the correct ‘‘vmunix’’ file and the second line has the pathname of the nfs_diskless structure file and its byte offset in it. For example:

  ```
  /var/diskless/vmunix.pmax
  /var/diskless/setup.rickers.cis.uoguelph.ca 642308
  ```
- Create a /var subtree for each client in an appropriate place on the server, such as /var/diskless/var/<client-hostname>/... By using the <client-hostname> to differentiate /var for each host, /etc/rc can be modified to mount the correct /var from the server.

[16] Unfortunately, a monitoring tool that uses these logs is still in the planning (dreaming) stage.

9. Not Quite NFS, Crash Tolerant Cache Consistency for NFS

Not Quite NFS (NQNFS) is an NFS like protocol designed to maintain full cache consistency between clients in a crash tolerant manner. It is an adaptation of the NFS protocol such that the server supports both NFS and NQNFS clients while maintaining full consistency between the server and NQNFS clients. This section borrows heavily from work done on Spritely-NFS [Srinivasan89], but uses Leases [Gray89] to avoid the need to recover server state information after a crash. The reader is strongly encouraged to read these references before trying to grasp the material presented here.

9.1. Overview

The protocol maintains cache consistency by using a somewhat Sprite [Nelson88] like protocol, but is based on short term leases[17] instead of hard state information about open files. The basic principal is that the protocol will disable client caching of a file whenever that file is write shared[18]. Whenever a client wishes to cache data for a file it must hold a valid lease. There are three types of leases: read caching, write caching and non-caching. The latter type requires that all file operations be done synchronously with the server via. RPCs. A read caching lease allows for client data caching, but no file modifications may be done. A write caching lease allows for client caching of writes, but requires that all writes be pushed to the server when the lease expires. If a client has dirty buffers[19] when a write cache lease has almost expired, it will attempt to extend the lease but is required to push the dirty buffers if extension fails. A client gets leases by either doing a **GetLease RPC** or by piggybacking a **GetLease Request** onto another RPC. Piggybacking is supported for the frequent RPCs Getattr, Setattr, Lookup, Readlink, Read, Write and Readdir in an effort to minimize the number of **GetLease RPCs** required. All leases are at the granularity of a file, since all NFS RPCs operate on individual files and NFS has no intrinsic notion of a file hierarchy. Directories, symbolic links and file attributes may be read cached but are not write cached. The exception here is the attribute file_size, which is updated during cached writing on the client to reflect a growing file.

It is the server's responsibility to ensure that consistency is maintained among the NQNFS clients by disabling client caching whenever a server file operation would cause inconsistencies. The possibility of inconsistencies occurs whenever a client has a write caching lease and any other client, or local operations on the server, tries to access the file or when a modify operation is attempted on a file being read cached by client(s). At this time, the server sends an **eviction notice** to all clients holding the lease and then waits for lease termination. Lease termination occurs when a **vacated the premises** message has been received from all the clients that have signed the lease or when the lease expires via. timeout. The message pair **eviction notice** and **vacated the premises** roughly correspond to a Sprite server→client callback, but are not implemented as an actual RPC, to avoid the server waiting indefinitely for a reply from a dead client.

Server consistency checking can be viewed as issuing intrinsic leases for a file operation for the duration of the operation only. For example, the **Create RPC** will get an intrinsic write lease on the directory in which the file is being created, disabling client read caches for that directory.

By relegating this responsibility to the server, consistency between the server and NQNFS clients is maintained when NFS clients are modifying the file system as well.[20]

The leases are issued as time intervals to avoid the requirement of time of day clock synchronization. There are three important time constants known to the server. The **maximum_lease_term** sets an upper bound on lease duration. The **clock_skew** is added to all lease terms on the server to correct for differing clock speeds between the client and server and **write_slack** is the number of seconds the server is willing to wait for a client with an expired write caching lease to push dirty writes.

The server maintains a **modify_revision** number for each file. It is defined as a unsigned quadword integer that is never zero and that must increase whenever the corresponding file is modified on the server. It is used by the client to determine whether or not cached data for the file is stale. Generating this value is

---

[17] A lease is a ticket permitting an activity that is valid until some expiry time.

[18] Write sharing occurs when at least one client is modifying a file while other client(s) are reading the file.

[19] Cached write data is not yet pushed (written) to the server.

[20] The NFS clients will continue to be *approximately* consistent with the server.

easier said than done. The current implementation uses the following technique, which is believed to be adequate. The high order longword is stored in the ufs inode and is initialized to one when an inode is first allocated. The low order longword is stored in main memory only and is initialized to zero when an inode is read in from disk. When the file is modified for the first time within a given second of wall clock time, the high order longword is incremented by one and the low order longword reset to zero. For subsequent modifications within the same second of wall clock time, the low order longword is incremented. If the low order longword wraps around to zero, the high order longword is incremented again. Since the high order longword only increments once per second and the inode is pushed to disk frequently during file modification, this implies $0 \le \text{Current} - \text{Disk} \le 5$. When the inode is read in from disk, 10 is added to the high order longword, which ensures that the quadword is greater than any value it could have had before a crash. This introduces apparent modifications every time the inode falls out of the LRU inode cache, but this should only reduce the client caching performance by a (hopefully) small margin.

9.2. Crash Recovery and other Failure Scenarios

The server must maintain the state of all the current leases held by clients. The nice thing about short term leases is that maximum_lease_term seconds after the server stops issuing leases, there are no current leases left. As such, server crash recovery does not require any state recovery. After rebooting, the server refuses to service any RPCs except for writes until write_slack seconds after the last lease would have expired[21]. By then, the server would not have any outstanding leases to recover the state of and the clients have had at least write_slack seconds to push dirty writes to the server and get the server sync'd up to date. After this, the server simply services requests in a manner similar to NFS. In an effort to minimize the effect of "recovery storms" [Baker91], the server replies **try_again_later** to the RPCs it is not yet ready to service.

After a client crashes, the server may have to wait for a lease to timeout before servicing a request if write sharing of a file with a cachable lease on the client is about to occur. As for the client, it simply starts up getting any leases it now needs. Any outstanding leases for that client on the server prior to the crash will either be renewed or expire via timeout.

Certain network partitioning failures are more problematic. If a client to server network connection is severed just before a write caching lease expires, the client cannot push the dirty writes to the server. After the lease expires on the server, the server permits other clients to access the file with the potential of getting stale data. Unfortunately I believe this failure scenario is intrinsic in any delay write caching scheme unless the server is required to wait **forever** for a client to regain contact[22]. Since the write caching lease has expired on the client, it will sync up with the server as soon as the network connection has been re-established.

There is another failure condition that can occur when the server is congested. The worst case scenario would have the client pushing dirty writes to the server but a large request queue on the server delays these writes for more than **write_slack** seconds. It is hoped that a congestion control scheme using the **try_again_later** RPC reply after booting combined with the following lease termination rule for write caching leases can minimize the risk of this occurrence. A write caching lease is only terminated on the server when there are have been no writes to the file and the server has not been overloaded during the previous write_slack seconds. The server has not been overloaded is approximated by a test for sleeping nfsd(s) at the end of the write_slack period.

9.3. Server Disk Full

There is a serious unresolved problem for delayed write caching with respect to server disk space allocation. When the disk on the file server is full, delayed write RPCs can fail due to "out of space". For NFS, this occurrence results in an error return from the close system call on the file, since the dirty blocks are pushed on close. Processes writing important files can check for this error return to ensure that the file was written successfully. For NQNFS, the dirty blocks are not pushed on close and as such the client may

---

[21] The last lease expiry time may be safely estimated as "boottime+maximum_lease_term+clock_skew" for machines that cannot store it in nonvolatile RAM.

[22] Gray and Cheriton avoid this problem by using a **write through** policy.

not attempt the write RPC until after the process has done the close which implies no error return from the close. For the current prototype, the only solution is to modify programs writing important file(s) to call fsync and check for an error return from it instead of close.

9.4. Protocol Details

The protocol specification is identical to that of NFS [Sun89] except for the following changes.

- RPC Information

  Program Number 300105
  Version Number 1

- Readdir_and_Lookup RPC

```
struct readdirlookargs {
        fhandle file;
        nfscookie cookie;
        unsigned count;
        unsigned duration;
};

struct entry {
        unsigned cachable;
        unsigned duration;
        modifyrev rev;
        fhandle entry_fh;
        nqnfs_fattr entry_attrib;
        unsigned fileid;
        filename name;
        nfscookie cookie;
        entry *nextentry;
};

union readdirlookres switch (stat status) {
case NFS_OK:
        struct {
                entry *entries;
                bool eof;
        } readdirlookok;
default:
        void;
};

readdirlookres
NQNFSPROC_READDIRLOOK(readdirlookargs) = 18;
```

  Reads entries in a directory in a manner analogous to the NFSPROC_READDIR RPC in NFS, but returns the file handle and attributes of each entry as well. This allows the attribute and lookup caches to be primed.

- Get Lease RPC

```
struct getleaseargs {
        fhandle file;
        cachetype readwrite;
        unsigned duration;
};

union getleaseres switch (stat status) {
case NFS_OK:
```

```
        bool cachable;
        unsigned duration;
        modifyrev rev;
        nqnfs_fattr attributes;
default:
        void;
};

getleaseres
NQNFSPROC_GETLEASE(getleaseargs) = 19;
```

Gets a lease for "file" valid for "duration" seconds from when the lease was issued on the server[23]. The lease permits client caching if "cachable" is true. The modify revision level and attributes for the file are also returned.

- Eviction Message

```
void
NQNFSPROC_EVICTED (fhandle) = 21;
```

This message is sent from the server to the client. When the client receives the message, it should flush data associated with the file represented by "fhandle" from its caches and then send the **Vacated Message** back to the server. Flushing includes pushing any dirty writes via. write RPCs.

- Vacated Message

```
void
NQNFSPROC_VACATED (fhandle) = 20;
```

This message is sent from the client to the server in response to the **Eviction Message**. See above.

- Access RPC

```
struct accessargs {
        fhandle file;
        bool read_access;
        bool write_access;
        bool exec_access;
};

stat
NQNFSPROC_ACCESS(accessargs) = 22;
```

The access RPC does permission checking on the server for the given type of access required by the client for the file. Use of this RPC avoids accessibility problems caused by client->server uid mapping.

- Piggybacked Get Lease Request

The piggybacked get lease request is functionally equivalent to the Get Lease RPC except that is attached to one of the other NQNFS RPC requests as follows. A getleaserequest is prepended to all of the request arguments for NQNFS and a getleaserequestres is inserted in all NFS result structures just after the "stat" field only if "stat == NFS_OK".

```
union getleaserequest switch (cachetype type) {
case NQLREAD:
case NQLWRITE:
        unsigned duration;
default:
        void;
```

[23] To be safe, the client may only assume that the lease is valid for ‘‘duration’’ seconds from when the RPC request was sent to the server.

```
};

union getleaserequestres switch (cachetype type) {
case NQLREAD:
case NQLWRITE:
        bool cachable;
        unsigned duration;
        modifyrev rev;
default:
        void;
};
```

The get lease request applies to the file that the attached RPC operates on and the file attributes remain in the same location as for the NFS RPC reply structure.

- Three additional "stat" values

  Three additional values have been added to the enumerated type "stat".

```
NQNFS_EXPIRED=500
NQNFS_TRYLATER=501
NQNFS_AUTHERR=502
```

The "expired" value indicates that a lease has expired. The "try later" value is returned by the server when it wishes the client to retry the RPC request after a short delay. It is used during crash recovery (Section 2) and may also be useful for server congestion control. The "authetication error" value is returned for kerberized mount points to indicate that there is no cached authentication mapping and a Kerberos ticket for the principal is required.

9.5. Data Types

- cachetype

```
enum cachetype {
        NQLNONE = 0,
        NQLREAD = 1,
        NQLWRITE = 2
};
```

  Type of lease requested. NQLNONE is used to indicate no piggybacked lease request.

- modifyrev

```
typedef unsigned hyper modifyrev;
```

  The "modifyrev" is a unsigned quadword integer value that is never zero and increases every time the corresponding file is modified on the server.

- nqnfs_time

```
struct nqnfs_time {
        unsigned seconds;
        unsigned nano_seconds;
};
```

  For NQNFS times are handled at nano second resolution instead of micro second resolution for NFS.

- nqnfs_fattr

```
struct nqnfs_fattr {
        ftype type;
        unsigned mode;
        unsigned nlink;
        unsigned uid;
        unsigned gid;
        unsigned hyper size;
        unsigned blocksize;
```

```
        unsigned rdev;
        unsigned hyper bytes;
        unsigned fsid;
        unsigned fileid;
        nqnfs_time atime;
        nqnfs_time mtime;
        nqnfs_time ctime;
        unsigned flags;
        unsigned generation;
        modifyrev rev;
};
```

The nqnfs_fattr structure is modified from the NFS fattr so that it stores the file size as a 64bit quantity and the storage occupied as a 64bit number of bytes. It also has fields added for the 4.4BSD va_flags and va_gen fields as well as the file's modify rev level.

- nqnfs_sattr

```
struct nqnfs_sattr {
        unsigned mode;
        unsigned uid;
        unsigned gid;
        unsigned hyper size;
        nqnfs_time atime;
        nqnfs_time mtime;
        unsigned flags;
        unsigned rdev;
};
```

The nqnfs_sattr structure is modified from the NFS sattr structure in the same manner as fattr.

The arguments to several of the NFS RPCs have been modified as well. Mostly, these are minor changes to use 64bit file offsets or similar. The modified argument structures follow.

- Lookup RPC

```
struct lookup_diropargs {
        unsigned duration;
        fhandle dir;
        filename name;
};

union lookup_diropres switch (stat status) {
case NFS_OK:
        struct {
                union getleaserequestres lookup_lease;
                fhandle file;
                nqnfs_fattr attributes;
        } lookup_diropok;
default:
        void;
};
```

The additional "duration" argument tells the server to get a lease for the name being looked up if it is non-zero and the lease is specified in "lookup_lease".

- Read RPC

```
struct nqnfs_readargs {
        fhandle file;
        unsigned hyper offset;
        unsigned count;
```

```
};
```

- Write RPC

```
struct nqnfs_writeargs {
        fhandle file;
        unsigned hyper offset;
        bool append;
        nfsdata data;
};
```

The "append" argument is true for apeend only write operations.

- Get Filesystem Attributes RPC

```
union nqnfs_statfsres (stat status) {
case NFS_OK:
        struct {
                unsigned tsize;
                unsigned bsize;
                unsigned blocks;
                unsigned bfree;
                unsigned bavail;
                unsigned files;
                unsigned files_free;
        } info;
default:
        void;
};
```

The "files" field is the number of files in the file system and the "files_free" is the number of additional files that can be created.

10. Summary

The configuration and tuning of an NFS environment tends to be a bit of a mystic art, but hopefully this paper along with the man pages and other reading will be helpful. Good Luck.

11. Bibliography

[Baker91] Mary Baker and John Ousterhout, Availability in the Sprite Distributed File System, In *Operating System Review*, (25)2, pg. 95-98, April 1991.

[Baker91a] Mary Baker, Private Email Communication, May 1991.

[Burrows88] Michael Burrows, Efficient Data Sharing, Technical Report #153, Computer Laboratory, University of Cambridge, Dec. 1988.

[Gray89] Cary G. Gray and David R. Cheriton, Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency, In *Proc. of the Twelfth ACM Symposium on Operating Systems Principals*, Litchfield Park, AZ, Dec. 1989.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham and Michael J. West, Scale and Performance in a Distributed File System, *ACM Trans. on Computer Systems*, (6)1, pg 51-81, Feb. 1988.

[Juszczak89] Chet Juszczak, Improving the Performance and Correctness of an NFS Server, In *Proc. Winter 1989 USENIX Conference*, pg. 53-63, San Diego, CA, January 1989.

[Keith90] Bruce E. Keith, Perspectives on NFS File Server Performance Characterization, In *Proc. Summer 1990 USENIX Conference*, pg. 267-277, Anaheim, CA, June 1990.

[Kent87] Christopher. A. Kent, *Cache Coherence in Distributed Systems*, Research Report 87/4, Digital Equipment Corporation Western Research Laboratory, April 1987.

[Kent87a] Christopher. A. Kent and Jeffrey C. Mogul, *Fragmentation Considered Harmful*, Research Report 87/3, Digital Equipment Corporation Western Research Laboratory, Dec. 1987.

[Macklem91] Rick Macklem, Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol, In *Proc. Winter USENIX Conference*, pg. 53-64, Dallas, TX, January 1991.

[Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, Caching in the Sprite Network File System, *ACM Transactions on Computer Systems* (6)1 pg. 134-154, February 1988.

[Nowicki89] Bill Nowicki, Transport Issues in the Network File System, In *Computer Communication Review*, pg. 16-20, Vol. 19, Number 2, April 1989.

[Ousterhout90] John K. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proc. Summer 1990 USENIX Conference*, pg. 247-256, Anaheim, CA, June 1990.

[Pendry93] Jan-Simon Pendry, 4.4 BSD Automounter Reference Manual, In *src/usr.sbin/amd/doc directory of 4.4 BSD distribution tape*.

[Reid90] Jim Reid, N(e)FS: the Protocol is the Problem, In *Proc. Summer 1990 UKUUG Conference*, London, England, July 1990.

[Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, Design and Implementation of the Sun Network filesystem, In *Proc. Summer 1985 USENIX Conference*, pages 119-130, Portland, OR, June 1985.

[Schroeder85] Michael D. Schroeder, David K. Gifford and Roger M. Needham, A Caching File System For A Programmer's Workstation, In *Proc. of the Tenth ACM Symposium on Operating Systems Principals*, pg. 25-34, Orcas Island, WA, Dec. 1985.

[Srinivasan89] V. Srinivasan and Jeffrey. C. Mogul, *Spritely NFS: Implementation and Performance of Cache-Consistency Protocols*, Research Report 89/5, Digital Equipment Corporation Western Research Laboratory, May 1989.

[Steiner88] Jennifer G. Steiner, Clifford Neuman and Jeffrey I. Schiller, Kerberos: An Authentication Service for Open Network Systems, In *Proc. Winter 1988 USENIX Conference*, Dallas, TX, February 1988.

[Stern] Hal Stern, *Managing NFS and NIS*, O'Reilly and Associates, ISBN 0-937175-75-7.

[Sun87] Sun Microsystems Inc., *XDR: External Data Representation Standard*, RFC1014, Network Information Center, SRI International, June 1987.

[Sun88] Sun Microsystems Inc., *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC1057, Network Information Center, SRI International, June 1988.

[Sun89] Sun Microsystems Inc., *NFS: Network File System Protocol Specification*, ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, March 1989, RFC-1094.