Not Quite NFS, Soft Cache Consistency for NFS

Rick Macklem
University of Guelph

Abstract

There are some constraints inherent in the NFS™∈ protocol that result in performance limitations for high performance workstation environments. This paper discusses an NFS-like protocol named Not Quite NFS (NQNFS), designed to address some of these limitations. This protocol provides full cache consistency during normal operation, while permitting more effective client-side caching in an effort to improve performance. There are also a variety of minor protocol changes, in order to resolve various NFS issues. The emphasis is on observed performance of a preliminary implementation of the protocol, in order to show how well this design works and to suggest possible areas for further improvement.

1. Introduction

It has been observed that overall workstation performance has not been scaling with processor speed and that file system I/O is a limiting factor [Ousterhout90]. Ousterhout notes that a principal challenge for operating system developers is the decoupling of system calls from their underlying I/O operations, in order to improve average system call response times. For distributed file systems, every synchronous Remote Procedure Call (RPC) takes a minimum of a few milliseconds and, as such, is analogous to an underlying I/O operation. This suggests that client caching with a very good hit ratio for read type operations, along with asynchronous writing, is required in order to avoid delays waiting for RPC replies. However, the NFS protocol requires that the server be stateless[1] and does not provide any explicit mechanism for client cache consistency, putting constraints on how the client may cache data. This paper describes an NFS-like protocol that includes a cache consistency component designed to enhance client caching performance. It does provide full consistency under normal operation, but without requiring that hard state information be maintained on the server. Design tradeoffs were made towards simplicity and high performance over cache consistency under abnormal conditions. The protocol design uses a variation of Leases [Gray89] to provide state on the server that does not need to be recovered after a crash.

The protocol also includes changes designed to address other limitations of NFS in a modern workstation environment. The use of TCP transport is optionally available to avoid the pitfalls of Sun RPC over UDP transport when running across an internetwork [Nowicki89]. Kerberos [Steiner88] support is available to do proper user authentication, in order to provide improved security and arbitrary client to server user ID mappings. There are also a variety of other changes to accommodate large file systems, such as 64bit file sizes and offsets, as well as lifting the 8Kbyte I/O size limit. The remainder of this paper gives an overview of the protocol, highlighting performance related components, followed by an evaluation of resultant performance for the 4.4BSD implementation.

2. Distributed File Systems and Caching

Clients using distributed file systems cache recently-used data in order to reduce the number of synchronous server operations, and therefore improve average response times for system calls. Unfortunately, maintaining consistency between these caches is a problem whenever write sharing occurs; that is, when a process on a client writes to a file and one or more processes on other client(s) read the file. If the writer closes the file before any reader(s) open the file for reading, this is called sequential write sharing. Both the Andrew ITC file system [Howard88] and NFS [Sandberg85] maintain consistency for sequential write sharing by requiring the writer to push all the writes through to the server on close and having readers check to see if the file has been modified upon open. If the file has been modified, the client throws away all cached data for that file, as it is now stale. NFS implementations typically detect file modification by checking a cached copy of the file's modification time; since this cached value is often several seconds out of date and only has a resolution of one second, an NFS client often uses stale cached data for some time after the file has been updated on the server.

A more difficult case is concurrent write sharing, where write operations are intermixed with read operations. Consistency for this case, often referred to as "full cache consistency," requires that a reader always receives the

[1] The server must not require any state that may be lost due to a crash, to function correctly.

most recently written data. Neither NFS nor the Andrew ITC file system maintain consistency for this case. The simplest mechanism for maintaining full cache consistency is the one used by Sprite [Nelson88], which disables all client caching of the file whenever concurrent write sharing might occur. There are other mechanisms described in the literature [Kent87a, Burrows88], but they appeared to be too elaborate for incorporation into NQNFS (for example, Kent's requires specialized hardware). NQNFS differs from Sprite in the way it detects write sharing. The Sprite server maintains a list of files currently open by the various clients and detects write sharing when a file open request for writing is received and the file is already open for reading (or vice versa). This list of open files is hard state information that must be recovered after a server crash, which is a significant problem in its own right [Mogul93, Welch90].

The approach used by NQNFS is a variant of the Leases mechanism [Gray89]. In this model, the server issues to a client a promise, referred to as a "lease," that the client may cache a specific object without fear of conflict. A lease has a limited duration and must be renewed by the client if it wishes to continue to cache the object. In NQNFS, clients hold short-term (up to one minute) leases on files for reading or writing. The leases are analogous to entries in the open file list, except that they expire after the lease term unless renewed by the client. As such, one minute after issuing the last lease there are no current leases and therefore no lease records to be recovered after a crash, hence the term "soft server state."

A related design consideration is the way client writing is done. Synchronous writing requires that all writes be pushed through to the server during the write system call. This is the simplest variant, from a consistency point of view, since the server always has the most recently written data. It also permits any write errors, such as "file system out of space" to be propagated back to the client's process via the write system call return. Unfortunately this approach limits the client write rate, based on server write performance and client/server RPC round trip time (RTT).

An alternative to this is delayed writing, where the write system call returns as soon as the data is cached on the client and the data is written to the server sometime later. This permits client writing to occur at the rate of local storage access up to the size of the local cache. Also, for cases where file truncation/deletion occurs shortly after writing, the write to the server may be avoided since the data has already been deleted, reducing server write load. There are some obvious drawbacks to this approach. For any Sprite-like system to maintain full consistency, the server must "callback" to the client to cause the delayed writes to be written back to the server when write sharing is about to occur. There are also problems with the propagation of errors back to the client process that issued the write system call. The reason for this is that the system call has already returned without reporting an error and the process may also have already terminated. As well, there is a risk of the loss of recently written data if the client crashes before the data is written back to the server.

A compromise between these two alternatives is asynchronous writing, where the write to the server is initiated during the write system call but the write system call returns before the write completes. This approach minimizes the risk of data loss due to a client crash, but negates the possibility of reducing server write load by throwing writes away when a file is truncated or deleted.

NFS implementations usually do a mix of asynchronous and delayed writing but push all writes to the server upon close, in order to maintain open/close consistency. Pushing the delayed writes on close negates much of the performance advantage of delayed writing, since the delays that were avoided in the write system calls are observed in the close system call. Akin to Sprite, the NQNFS protocol does delayed writing in an effort to achieve good client performance and uses a callback mechanism to maintain full cache consistency.

3. Related Work

There has been a great deal of effort put into improving the performance and consistency of the NFS protocol. This work can be put in two categories. The first category are implementation enhancements for the NFS protocol and the second involve modifications to the protocol.

The work done on implementation enhancements have attacked two problem areas, NFS server write performance and RPC transport problems. Server write performance is a major problem for NFS, in part due to the requirement to push all writes to the server upon close and in part due to the fact that, for writes, all data and meta-data must be committed to non-volatile storage before the server replies to the write RPC. The Prestoserve™† [Moran90] system uses non-volatile RAM as a buffer for recently written data on the server, so that the write RPC replies can be returned to the client before the data is written to the disk surface. Write gathering [Juszczak94] is a software technique used on the server where a write RPC request is delayed for a short time in the hope that another contiguous write request will arrive, so that they can be merged into one write operation. Since the replies to all of the merged writes are not returned to the client until the write operation is completed, this delay does not violate the protocol. When write operations are merged, the number of disk writes can be reduced, improving server write performance. Although either of the above reduces write RPC response time for the server, it cannot be reduced to zero, and so, any client side caching mechanism that reduces write RPC load or client dependence on server RPC response time should still improve overall performance. Good client side caching should be complementary to these server techniques, although client performance improvements as a result of caching may be less dramatic when these techniques are used.

In NFS, each Sun RPC request is packaged in a UDP datagram for transmission to the server. A timer is started, and if a timeout occurs before the corresponding RPC reply is received, the RPC request is retransmitted. There are two problems with this model. First, when a retransmit timeout occurs, the RPC may be redone, instead of simply retransmitting the RPC request message to the server. A recent-request cache can be used on the server to minimize the negative impact of redoing RPCs [Juszczak89]. The second problem is that a large UDP datagram, such as a read request or write reply, must be fragmented by IP and if any one IP fragment is lost in transit, the entire UDP datagram is lost [Kent87]. Since entire requests and replies are packaged in a single UDP datagram, this puts an upper bound on the read/write data size (8 kbytes).

Adjusting the retransmit timeout (RTT) interval dynamically and applying a congestion window on outstanding requests has been shown to be of some help [Nowicki89] with the retransmission problem. An alternative to this is to use TCP transport to delivery the RPC messages reliably [Macklem90] and one of the performance results in this paper shows the effects of this further.

Srinivasan and Mogul [Srinivasan89] enhanced the NFS protocol to use the Sprite cache consistency algorithm in an effort to improve performance and to provide full client cache consistency. This experimental implementation demonstrated significantly better performance than NFS, but suffered from a lack of crash recovery support. The NQNFS protocol design borrowed heavily from this work, but differed from the Sprite algorithm by using Leases instead of file open state to detect write sharing. The decision to use Leases was made primarily to avoid the crash recovery problem. More recent work by the Sprite group [Baker91] and Mogul [Mogul93] have addressed the crash recovery problem, making this design tradeoff more questionable now.

Sun has recently updated the NFS protocol to Version 3 [SUN93], using some changes similar to NQNFS to address various issues. The Version 3 protocol uses 64bit file sizes and offsets, provides a Readdir_and_Lookup RPC and an access RPC. It also provides cache hints, to permit a client to be able to determine whether a file modification is the result of that client's write or some other client's write. It would be possible to add either Spritely NFS or NQNFS support for cache consistency to the NFS Version 3 protocol.

4. NQNFS Consistency Protocol and Recovery

The NQNFS cache consistency protocol uses a somewhat Sprite-like [Nelson88] mechanism, but is based on Leases [Gray89] instead of hard server state information about open files. The basic principle is that the server disables client caching of files whenever concurrent write sharing could occur, by performing a server-to-client callback, forcing the client to flush its caches and to do all subsequent I/O on the file with synchronous RPCs. A Sprite server maintains a record of the open state of files for all clients and uses this to determine when concurrent write sharing might occur. This *open state* information might also be referred to as an infinite-term lease for the file, with explicit lease cancellation. NQNFS, on the other hand, uses a short-term lease that expires due to timeout after a maximum of one minute, unless explicitly renewed by the client. The fundamental difference is that an NQNFS client must keep renewing a lease to use cached data whereas a Sprite client assumes the data is valid until canceled by the server or the file is closed. Using leases permits the server to remain "stateless," since the soft state information, which consists of the set of current leases, is moot after one minute, when all the leases expire.

Whenever a client wishes to access a file's data it must hold one of three types of lease: read-caching, write-caching or non-caching. The latter type requires that all file operations be done synchronously with the server via the appropriate RPCs.

A read-caching lease allows for client data caching but no modifications may be done. It may, however, be shared between multiple clients. Diagram 1 shows a typical read-caching scenario. The vertical solid black lines depict the lease records. Note that the time lines are nowhere near to scale, since a client/server interaction will normally take less than one hundred milliseconds, whereas the normal lease duration is thirty seconds. Every lease includes a *modrev* value, which changes upon every modification of the file. It may be used to check to see if data cached on the client is still current.

A write-caching lease permits delayed write caching, but requires that all data be pushed to the server when the lease expires or is terminated by an eviction callback. When a write-caching lease has almost expired, the client will attempt to extend the lease if the file is still open, but is required to push the delayed writes to the server if renewal fails (as depicted by diagram 2). The writes may not arrive at the server until after the write lease has expired on the client, but this does not result in a consistency problem, so long as the write lease is still valid on the server. Note that, in diagram 2, the lease record on the server remains current after the expiry time, due to the conditions mentioned in section 5. If a write RPC is done on the server after the write lease has expired on the server, this could be considered an error since consistency could be lost, but it is not handled as such by NQNFS.

Diagram 3 depicts how read and write leases are replaced by a non-caching lease when there is the potential for write sharing.

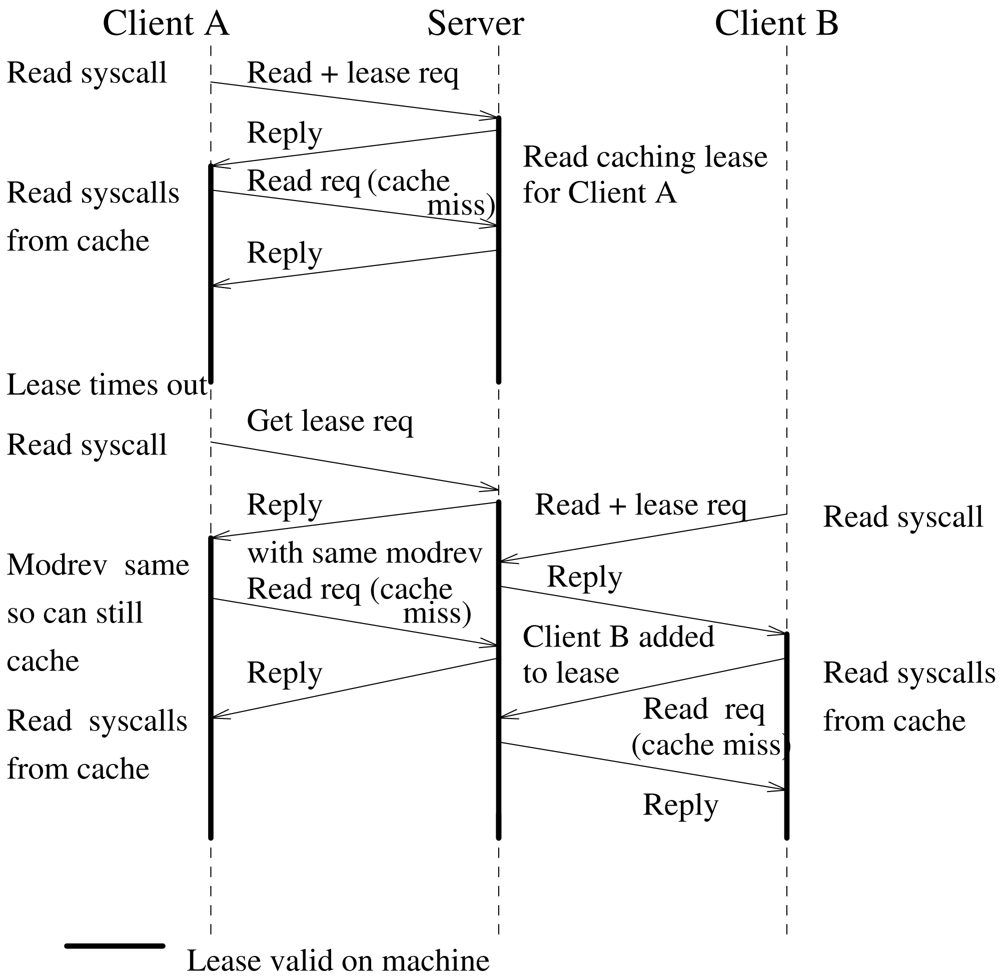


Diagram #1: Read Caching Leases

A write-caching lease is not used in the Stanford V Distributed System [Gray89], since synchronous writing is always used. A side effect of this change is that the five to ten second lease duration recommended by Gray was found to be insufficient to achieve good performance for the write-caching lease. Experimentation showed that thirty seconds was about optimal for cases where the client and server are connected to the same local area network, so thirty seconds is the default lease duration for NQNFS. A maximum of twice that value is permitted, since Gray showed that for some network topologies, a larger lease duration functions better. Although there is an explicit get_lease RPC defined for the protocol, most lease requests are piggybacked onto the other RPCs to minimize the additional overhead introduced by leasing.

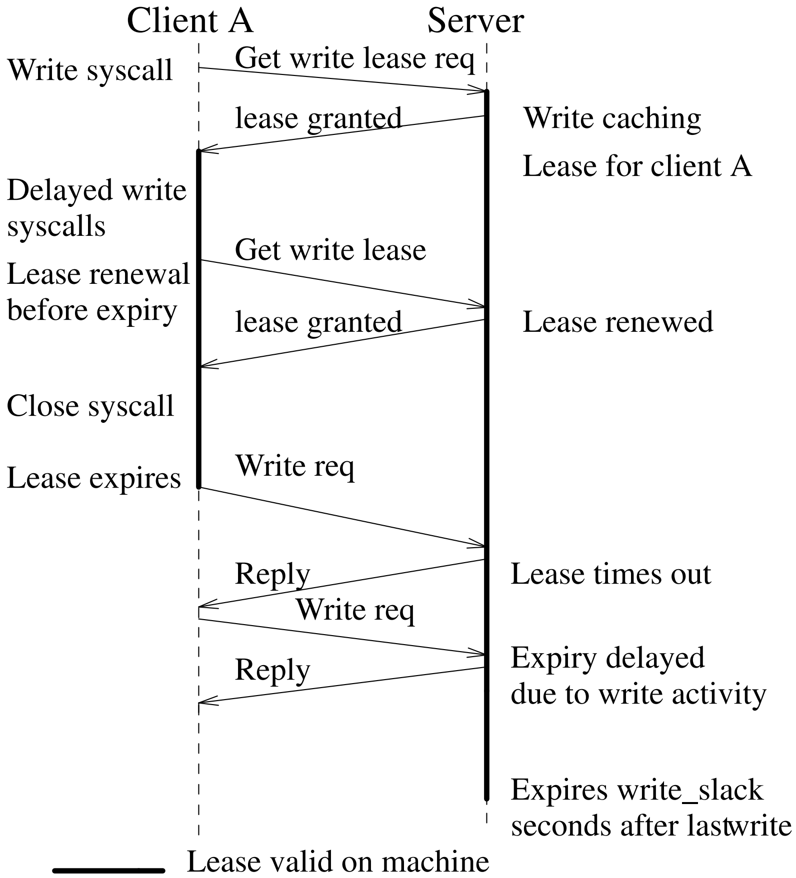


Diagram #2: Write Caching Lease

4.1 Rationale

Leasing was chosen over hard server state information for the following reasons:

1. The server must maintain state information about all current client leases. Since at most one lease is allocated for each RPC and the leases expire after their lease term, the upper bound on the number of current leases is the product of the lease term and the server RPC rate. In practice, it has been observed that less than 10% of RPCs request new leases and since most leases have a term of thirty seconds, the following rule of thumb should estimate the number of server lease records:

   Number of Server Lease Records = 0.1 * 30 * RPC rate

   Since each lease record occupies 64 bytes of server memory, storing the lease records should not be a serious problem. If a server has exhausted lease storage, it can simply wait a few seconds for a lease to expire and free up a record. On the other hand, a Sprite-like server must store records for all files currently open by all clients, which can require significant storage for a large, heavily loaded server. In [Mogul93], it is proposed that a mechanism vaguely similar to paging could be used to deal with this for Spritely NFS, but this appears to introduce a fair amount of complexity and may limit the usefulness of open records for storing other state information, such as file locks.

2. After a server crashes it must recover lease records for the current outstanding leases, which actually implies that if it waits until all leases have expired, there is no state to recover. The server must wait for the maximum lease duration of one minute, and it must serve all outstanding write requests resulting from terminated write-caching leases before issuing new leases. The one minute delay can be overlapped with file system consistency checking (eg. fsck). Because no state must be recovered, a lease-based server, like an NFS server, avoids the problem of state recovery after a crash.

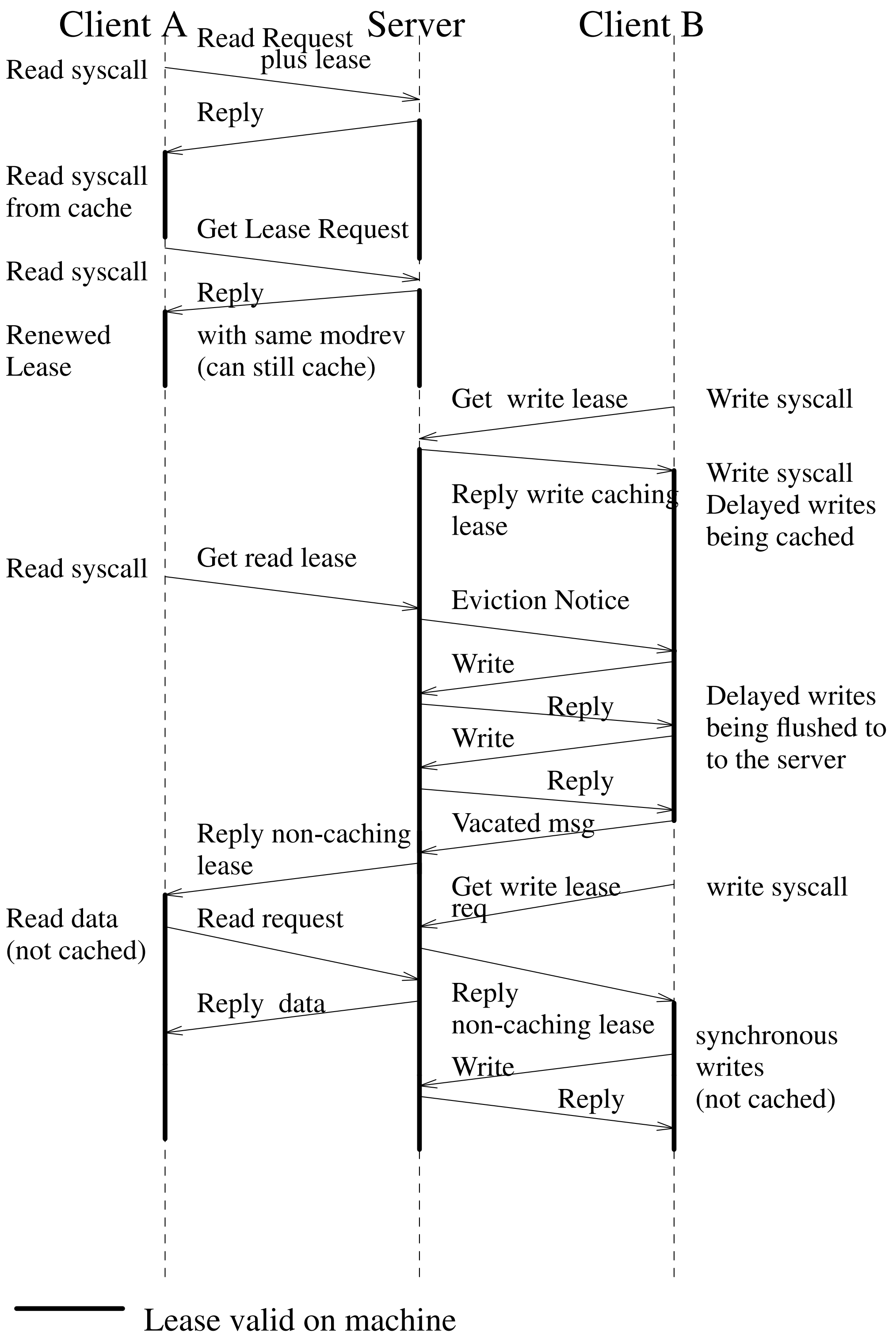


Diagram #3: Write sharing case

There can, however, be problems during crash recovery because of a potentially large number of write backs due to terminated write-caching leases. One of these problems is a "recovery storm" [Baker91], which could occur when the server is overloaded by the number of write RPC requests. The NQNFS protocol deals with this by replying with a return status code called try_again_later to all RPC requests (except write) until the write requests subside. At this time, there has not been sufficient testing of server crash recovery while under heavy server load to determine if the try_again_later reply is a sufficient solution to the problem. The other problem is that consistency will be lost if other RPCs are performed before all of the write backs for terminated write-caching leases have completed. This is handled by only performing write RPCs until no write RPC requests arrive for write_slack seconds, where write_slack is set to several times the client timeout retransmit interval, at which time it is assumed all clients have had an opportunity to send their writes to the server.

3. Another advantage of leasing is that, since leases are required at times when other I/O operations occur, lease requests can almost always be piggybacked on other RPCs, avoiding some of the overhead associated with the explicit open and close RPCs required by a Sprite-like system. Compared with Sprite cache consistency, this can result in a significantly lower RPC load (see table #1).

5. Limitations of the NQNFS Protocol

There is a serious risk when leasing is used for delayed write caching. If the server is simply too busy to service a lease renewal before a write-caching lease terminates, the client will not be able to push the write data to the server before the lease has terminated, resulting in inconsistency. Note that the danger of inconsistency occurs when the server assumes that a write-caching lease has terminated before the client has had the opportunity to write the data back to the server. In an effort to avoid this problem, the NQNFS server does not assume that a write-caching lease has terminated until three conditions are met:

1 - clock time > (expiry time + clock skew)
2 - there is at least one server daemon (nfsd) waiting for an RPC request
3 - no write RPCs received for leased file within write_slack after the corrected expiry time

The first condition ensures that the lease has expired on the client. The clock_skew, by default three seconds, must be set to a value larger than the maximum time-of-day clock error that is likely to occur during the maximum lease duration. The second condition attempts to ensure that the client is not waiting for replies to any writes that are still queued for service by an nfsd. The third condition tries to guarantee that the client has transmitted all write requests to the server, since write_slack is set to several times the client's timeout retransmit interval.

There are also certain file system semantics that are problematic for both NFS and NQNFS, due to the lack of state information maintained by the server. If a file is unlinked on one client while open on another it will be removed from the file server, resulting in failed file accesses on the client that has the file open. If the file system on the server is out of space or the client user's disk quota has been exceeded, a delayed write can fail long after the write system call was successfully completed. With NFS this error will be detected by the close system call, since the delayed writes are pushed upon close. With NQNFS however, the delayed write RPC may not occur until after the close system call, possibly even after the process has exited. Therefore, if a process must check for write errors, a system call such as *fsync* must be used.

Another problem occurs when a process on one client is running an executable file and a process on another client starts to write to the file. The read lease on the first client is terminated by the server, but the client has no recourse but to terminate the process, since the process is already in progress on the old executable.

The NQNFS protocol does not support file locking, since a file lock would have to involve hard, recovered after a crash, state information.

6. Other NQNFS Protocol Features

NQNFS also includes a variety of minor modifications to the NFS protocol, in an attempt to address various limitations. The protocol uses 64bit file sizes and offsets in order to handle large files. TCP transport may be used as an alternative to UDP for cases where UDP does not perform well. Transport mechanisms such as TCP also permit the use of much larger read/write data sizes, which might improve performance in certain environments.

The NQNFS protocol replaces the Readdir RPC with a Readdir_and_Lookup RPC that returns the file handle and attributes for each file in the directory as well as name and file id number. This additional information may then be loaded into the lookup and file-attribute caches on the client. Thus, for cases such as "ls -l", the *stat* system calls can be performed locally without doing any lookup or getattr RPCs. Another additional RPC is the Access RPC that checks for file accessibility against the server. This is necessary since in some cases the client user ID is mapped to a different user on the server and doing the access check locally on the client using file attributes and client credentials is not correct. One case where this becomes necessary is when the NQNFS mount point is using Kerberos authentication, where the Kerberos authentication ticket is translated to credentials on the server that are mapped to the client side user id. For further details on the protocol, see [Macklem93].

7. Performance

In order to evaluate the effectiveness of the NQNFS protocol, a benchmark was used that was designed to typify real work on the client workstation. Benchmarks, such as Laddis [Wittle93], that perform server load characterization are not appropriate for this work, since it is primarily client caching efficiency that needs to be evaluated. Since these tests are measuring overall client system performance and not just the performance of the file system, each sequence of runs was performed on identical hardware and operating system in order to factor out the system components affecting performance other than the file system protocol.

The equipment used for the all the benchmarks are members of the DECstation™† family of workstations using the MIPS™§ RISC architecture. The operating system running on these systems was a pre-release version of 4.4BSD Unix™‡. For all benchmarks, the file server was a DECstation 2100 (10 MIPS) with 8Mbytes of memory and a local RZ23 SCSI disk (27msec average access time). The clients range in speed from DECstation 2100s to a DECstation 5000/25, and always run with six block I/O daemons and a 4Mbyte buffer cache, except for the test runs where the buffer cache size was the independent variable. In all cases /tmp is mounted on the local SCSI disk[2], all machines were attached to the same uncongested Ethernet, and ran in single user mode during the benchmarks. Unless noted otherwise, test runs used UDP RPC transport and the results given are the average values of four runs.

The benchmark used is the Modified Andrew Benchmark (MAB) [Ousterhout90], which is a slightly modified version of the benchmark used to characterize performance of the Andrew ITC file system [Howard88]. The MAB was set up with the executable binaries in the remote mounted file system and the final load step was commented out, due to a linkage problem during testing under 4.4BSD. Therefore, these results are not directly comparable to other reported MAB results. The MAB is made up of five distinct phases:

1. Makes five directories (no significant cost)
2. Copy a file system subtree to a working directory
3. Get file attributes (stat) of all the working files
4. Search for strings (grep) in the files
5. Compile a library of C sources and archive them

Of the five phases, the fifth is by far the largest and is the one affected most by client caching mechanisms. The results for phase #1 are invariant over all the caching mechanisms.

7.1 Buffer Cache Size Tests

The first experiment was done to see what effect changing the size of the buffer cache would have on client performance. A single DECstation 5000/25 was used to do a series of runs of MAB with different buffer cache sizes for four variations of the file system protocol. The four variations are as follows:

Case 1: NFS - The NFS protocol as implemented in 4.4BSD

Case 2: Leases - The NQNFS protocol using leases for cache consistency

Case 3: Leases, Rdirlookup - The NQNFS protocol using leases for cache consistency and with the readdir RPC replaced by Readdir_and_Lookup

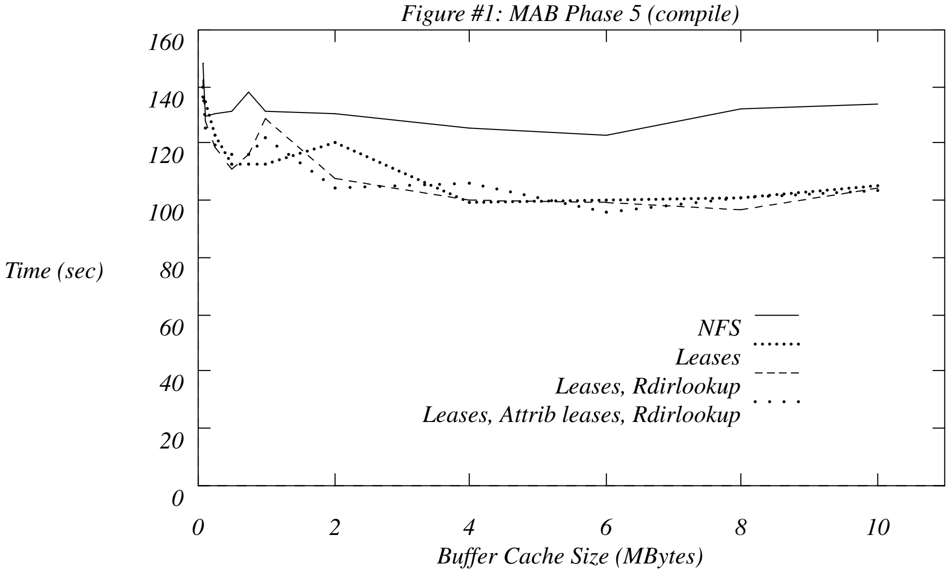
Case 4: Leases, Attrib leases, Rdirlookup - The NQNFS protocol using leases for cache consistency, with the readdir RPC replaced by the Readdir_and_Lookup, and requiring a valid lease not only for file-data access, but also for file-attribute access.

---

[2] Testing using the 4.4BSD MFS [McKusick90] resulted in slightly degraded performance, probably since the machines only had 16Mbytes of memory, and so paging increased.

As can be seen in figure 1, the buffer cache achieves about optimal performance for the range of two to ten megabytes in size. At eleven megabytes in size, the system pages heavily and the runs did not complete in a reasonable time. Even at 64Kbytes, the buffer cache improves performance over no buffer cache by a significant margin of 136-148 seconds versus 239 seconds. This may be due, in part, to the fact that the Compile Phase of the MAB uses a rather small working set of file data. All variants of NQNFS achieve about the same performance, running around 30% faster than NFS, with a slightly larger difference for large buffer cache sizes. Based on these results, all remaining tests were run with the buffer cache size set to 4Mbytes. Although I do not know what causes the local peak in the curves between 0.5 and 2 megabytes, there is some indication that contention for buffer cache blocks, between the update process (which pushes delayed writes to the server every thirty seconds) and the I/O system calls, may be involved.

Figure #1: MAB Phase 5 (compile)



7.2 Multiple Client Load Tests

During preliminary runs of the MAB, it was observed that the server RPC counts were reduced significantly by NQNFS as compared to NFS (table 1). (Spritely NFS and Ultrix™4.3/NFS numbers were taken from [Mogul93] and are not directly comparable, due to numerous differences in the experimental setup including deletion of the load step from phase 5.) This suggests that the NQNFS protocol might scale better with respect to the number of clients accessing the server. The experiment described in this section ran the MAB on from one to ten clients concurrently, to observe the effects of heavier server load. The clients were started at roughly the same time by pressing all the <return> keys together and, although not synchronized beyond that point, all clients would finish the test run within about two seconds of each other. This was not a realistic load of N active clients, but it did result in a reproducible increasing client load on the server. The results for the four variants are plotted in figures 2-5.

Table #1: MAB RPC Counts

RPC	Getattr	Read	Write	Lookup	Other	GetLease/Open-Close	Total
BSD/NQNFS	277	139	306	575	294	127	1718
BSD/NFS	1210	506	451	489	238	0	2894
Spritely NFS	259	836	192	535	306	1467	3595
Ultrix4.3/NFS	1225	1186	476	810	305	0	4002

For the MAB benchmark, the NQNFS protocol reduces the RPC counts significantly, but with a minimum of extra overhead (the GetLease/Open-Close count).

In figure 2, where a subtree of seventy small files is copied, the difference between the protocol variants is minimal, with the NQNFS variants performing slightly better. For this case, the Readdir_and_Lookup RPC is a slight hindrance under heavy load, possibly because it results in larger directory blocks in the buffer cache.

In figure 3, for the phase that gets file attributes for a large number of files, the leasing variants take about 50% longer, indicating that there are performance problems in this area. For the case where valid current leases are

Figure #2: MAB Phase 2 (copying)

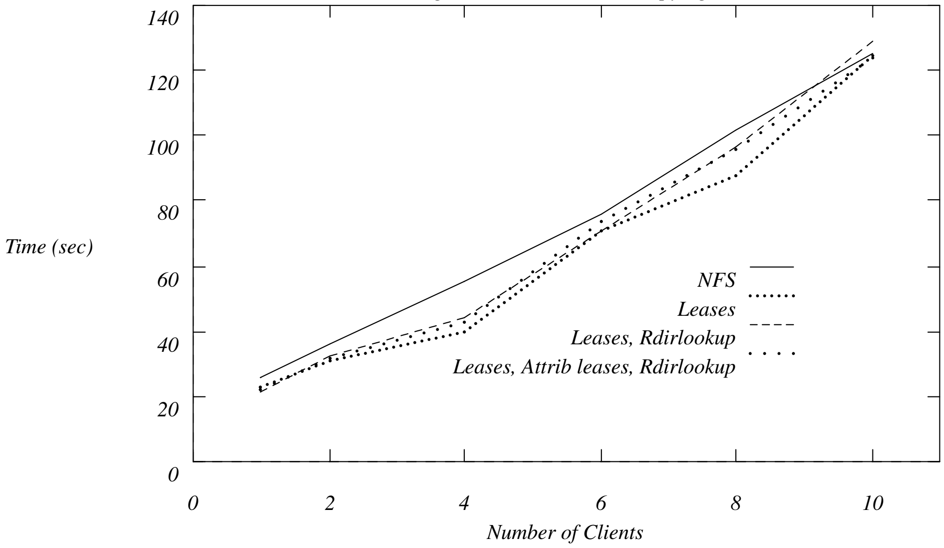


Figure #3: MAB Phase 3 (stat/find)

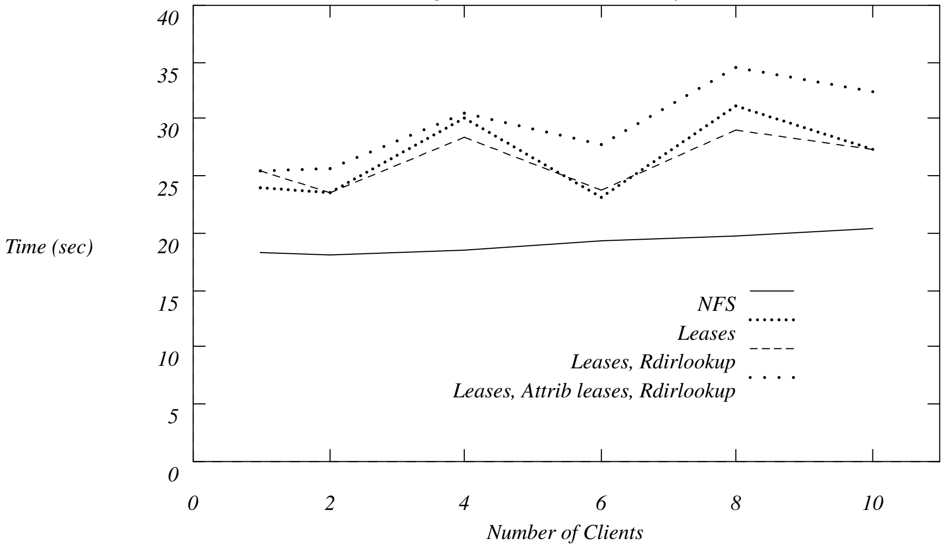


Figure #4: MAB Phase 4 (grep/wc/find)

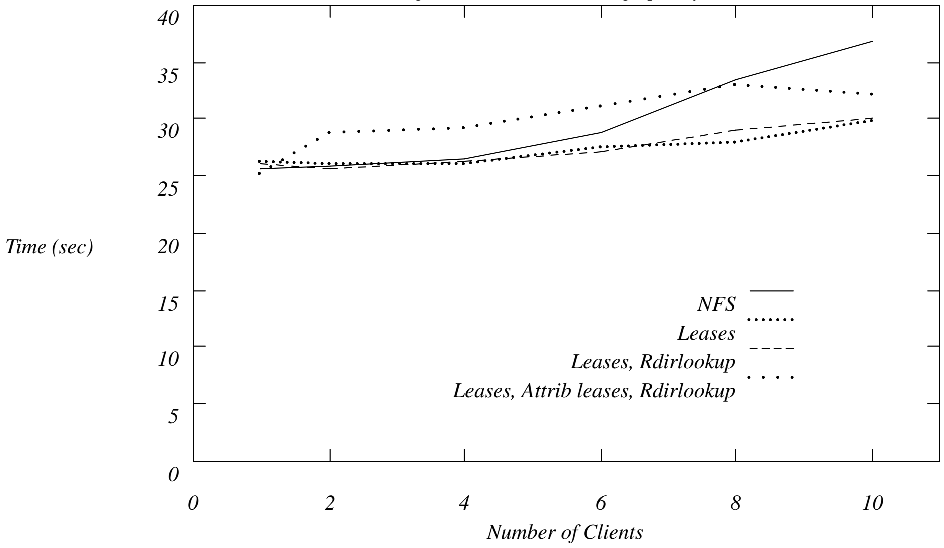
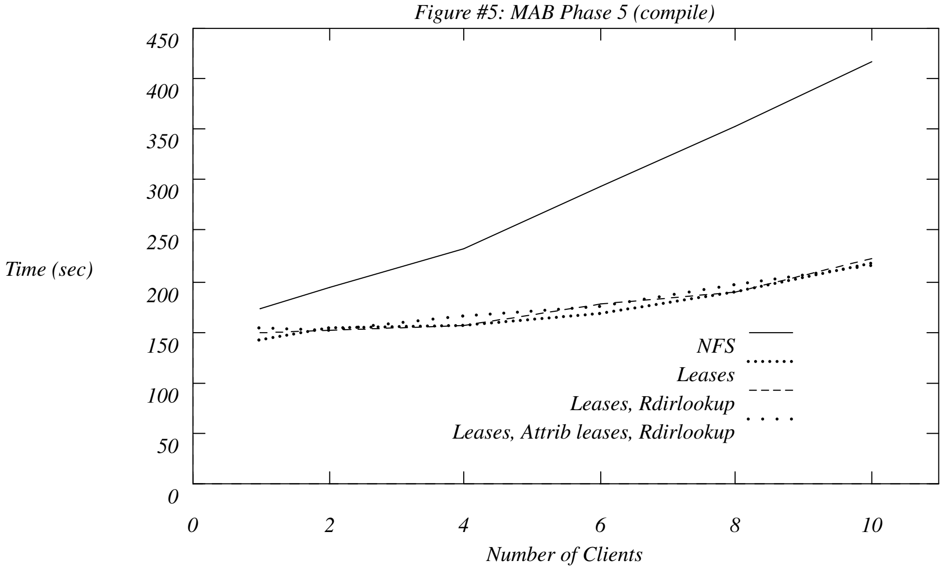


Figure #5: MAB Phase 5 (compile)



required for every file when attributes are returned, the performance is significantly worse than when the attributes are allowed to be stale by a few seconds on the client. I have not been able to explain the oscillation in the curves for the Lease cases.

For the string searching phase depicted in figure 4, the leasing variants that do not require valid leases for files when attributes are returned appear to scale better with server load than NFS. However, the effect appears to be negligible until the server load is fairly heavy.

Most of the time in the MAB benchmark is spent in the compilation phase and this is where the differences between caching methods are most pronounced. In figure 5 it can be seen that any protocol variant using Leases performs about a factor of two better than NFS at a load of ten clients. This indicates that the use of NQNFS may allow servers to handle significantly more clients for this type of workload.

Table 2 summarizes the MAB run times for all phases for the single client DECstation 5000/25. The *Leases* case refers to using leases, whereas the *Leases, Rdirl* case uses the Readdir_and_Lookup RPC as well and the *BCache Only* case uses leases, but only the buffer cache and not the attribute or name caches. The *No Caching* cases does not do any client side caching, performing all system calls via synchronous RPCs to the server.

Table #2: Single DECstation 5000/25 Client Elapsed Times (sec)

Phase	1	2	3	4	5	Total	% Improvement
No Caching	6	35	41	40	258	380	-93
NFS	5	24	15	20	133	197	0
BCache Only	5	20	24	23	116	188	5
Leases, Rdirl	5	20	21	20	105	171	13
Leases	5	19	21	21	99	165	16

7.3 Processor Speed Tests

An important goal of client-side file system caching is to decouple the I/O system calls from the underlying distributed file system, so that the client's system performance might scale with processor speed. In order to test this, a series of MAB runs were performed on three DECstations that are similar except for processor speed. In addition to the four protocol variants used for the above tests, runs were done with the client caches turned off, for worst case performance numbers for caching mechanisms with a 100% miss rate. The CPU utilization was measured, as an indicator of how much the processor was blocking for I/O system calls. Note that since the systems were running in single user mode and otherwise quiescent, almost all CPU activity was directly related to the MAB run. The results are presented in table 3. The CPU time is simply the product of the CPU utilization and elapsed running time and, as such, is the optimistic bound on performance achievable with an ideal client caching scheme that never blocks for I/O. As can be seen in the table, any caching mechanism achieves significantly better performance than when caching is disabled, roughly doubling the CPU utilization with a corresponding reduction in run time. For NFS, the

Table #3: MAB Phase 5 (compile)									
	DS2100 (10.5 MIPS)			DS3100 (14.0 MIPS)			DS5000/25 (26.7 MIPS)		
	Elapsed time	CPU Util(%)	CPU time	Elapsed time	CPU Util(%)	CPU time	Elapsed time	CPU Util(%)	CPU time
Leases	143	89	127	113	87	98	99	89	88
Leases, Rdirl	150	89	134	110	91	100	105	88	92
BCache Only	169	85	144	129	78	101	116	75	87
NFS	172	77	132	135	74	100	133	71	94
No Caching	330	47	155	256	41	105	258	39	101

CPU utilization is dropping with increase in CPU speed, which would suggest that it is not scaling with CPU speed. For the NQNFS variants, the CPU utilization remains at just below 90%, which suggests that the caching mechanism is working well and scaling within this CPU range. Note that for this benchmark, the ratio of CPU times for the DECstation 3100 and DECstation 5000/25 are quite different than the Dhrystone MIPS ratings would suggest.

Overall, the results seem encouraging, although it remains to be seen whether or not the caching provided by NQNFS can continue to scale with CPU performance. There is a good indication that NQNFS permits a server to scale to more clients than does NFS, at least for workloads akin to the MAB compile phase. A more difficult question is "What if the server is much faster doing write RPCs?" as a result of some technology such as Prestoserve or write gathering. Since a significant part of the difference between NFS and NQNFS is the synchronous writing, it is difficult to predict how much a server capable of fast write RPCs will negate the performance improvements of NQNFS. At the very least, table 1 indicates that the write RPC load on the server has decreased by approximately 30%, and this reduced write load should still result in some improvement.

Indications are that the Readdir_and_Lookup RPC has not improved performance for these tests and may in fact be degrading performance slightly. The results in figure 3 indicate some problems, possibly with handling of the attribute cache. It seems logical that the Readdir_and_Lookup RPC should be permit priming of the attribute cache improving hit rate, but the results are counter to that.

7.4 Internetwork Delay Tests

This experimental setup was used to explore how the different protocol variants might perform over internetworks with larger RPC RTTs. The server was moved to a separate Ethernet, using a MicroVAXII™ as an IP router to the other Ethernet. The 4.3Reno BSD Unix system running on the MicroVAXII was modified to delay IP packets being forwarded by a tunable N millisecond delay. The implementation was rather crude and did not try to simulate a distribution of delay times nor was it programmed to drop packets at a given rate, but it served as a simple emulation of a long, fat network[3] [Jacobson88]. The MAB was run using both UDP and TCP RPC transports for a variety of RTT delays from five to two hundred milliseconds, to observe the effects of RTT delay on RPC transport. It was found that, due to a high variability between runs, four runs was not suffice, so eight runs at each value was done. The results in figure 6 and table 4 are the average for the eight runs.

I found these results somewhat surprising, since I had assumed that stability across an internetwork connection would be a function of RPC transport protocol. Looking at the standard deviations observed between the eight runs, there is an indication that the NQNFS protocol plays a larger role in maintaining stability than the underlying RPC transport protocol. It appears that NFS over TCP transport is the least stable variant tested. It should be noted that the TCP implementation used was roughly at 4.3BSD Tahoe release and that the 4.4BSD TCP implementation was far less stable and would fail intermittently, due to a bug I was not able to isolate. It would appear that some of the recent enhancements to the 4.4BSD TCP implementation have a detrimental effect on the performance of RPC-type traffic loads, which intermix small and large data transfers in both directions. It is obvious that more exploration of this area is needed before any conclusions can be made beyond the fact that over a local area network, TCP transport provides performance comparable to UDP.

[3] Long fat networks refer to network interconnections with a Bandwidth X RTT product > 10^5 bits.

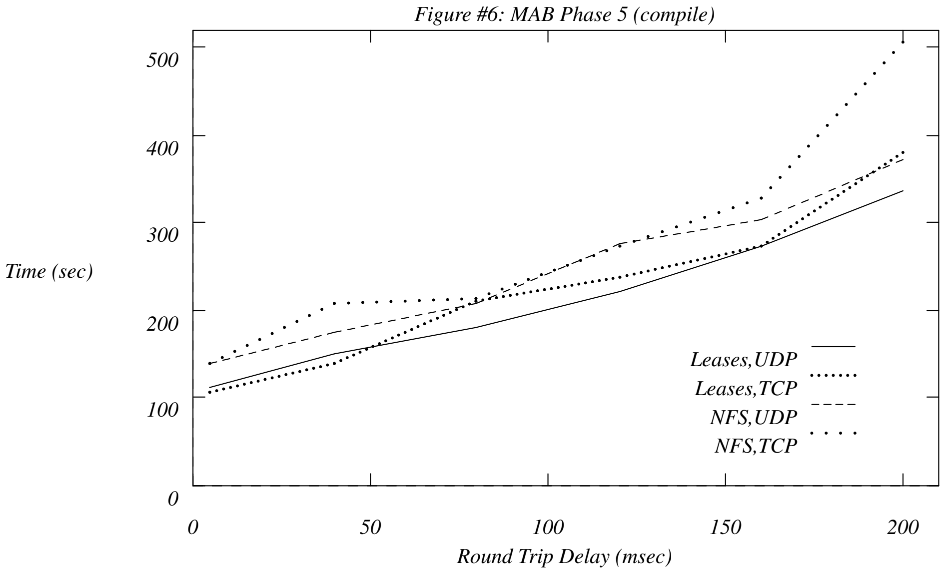


Table #4: MAB Phase 5 (compile) for Internetwork Delays

	NFS,UDP		NFS,TCP		Leases,UDP		Leases,TCP	
Delay (msec)	Elapsed time (sec)	Standard Deviation	Elapsed time (sec)	Standard Deviation	Elapsed time (sec)	Standard Deviation	Elapsed time (sec)	Standard Deviation
5	139	2.9	139	2.4	112	7.0	108	6.0
40	175	5.1	208	44.5	150	23.8	139	4.3
80	207	3.9	213	4.7	180	7.7	210	52.9
120	276	29.3	273	17.1	221	7.7	238	5.8
160	304	7.2	328	77.1	275	21.5	274	10.1
200	372	35.0	506	235.1	338	25.2	379	69.2

8. Lessons Learned

Evaluating the performance of a distributed file system is fraught with difficulties, due to the many software and hardware factors involved. The limited benchmarking presented here took a considerable amount of time and the results gained by the exercise only give indications of what the performance might be for a few scenarios.

The IP router with delay introduction proved to be a valuable tool for protocol debugging[4], and may be useful for a more extensive study of performance over internetworks if enhanced to do a better job of simulating internetwork delay and packet loss.

The Leases mechanism provided a simple model for the provision of cache consistency and did seem to improve performance for various scenarios. Unfortunately, it does not provide the server state information that is required for file system semantics, such as locking, that many software systems demand. In production environments on my campus, the need for file locking and the correct generation of the ETXTBSY error code are far more important that full cache consistency, and leasing does not satisfy these needs. Another file system semantic that requires hard server state is the delay of file removal until the last close system call. Although Spritely NFS did not support this semantic either, it is logical that the open file state maintained by that system would facilitate the implementation of this semantic more easily than would the Leases mechanism.

9. Further Work

The current implementation uses a fixed, moderate sized buffer cache designed for the local UFS [McKusick84] file system. The results in figure 1 suggest that this is adequate so long as the cache is of an appropriate size. However, a mechanism permitting the cache to vary in size has been shown to outperform fixed sized buffer caches

[4] It exposed two bugs in the 4.4BSD networking, one a problem in the Lance chip driver for the DECstation and the other a TCP window sizing problem that I was not able to isolate.

[Nelson90], and could be beneficial. It could also be useful to allow the buffer cache to grow very large by making use of local backing store for cases where server performance is limited. A very large buffer cache size would in turn permit experimentation with much larger read/write data sizes, facilitating bulk data transfers across long fat networks, such as will characterize the Internet of the near future. A careful redesign of the buffer cache mechanism to provide support for these features would probably be the next implementation step.

The results in figure 3 indicate that the mechanics of caching file attributes and maintaining the attribute cache's consistency needs to be looked at further. There also needs to be more work done on the interaction between a Readdir_and_Lookup RPC and the name and attribute caches, in an effort to reduce Getattr and Lookup RPC loads.

The NQNFS protocol has never been used in a production environment and doing so would provide needed insight into how well the protocol saisfies the needs of real workstation environments. It is hoped that the distribution of the implementation in 4.4BSD will facilitate use of the protocol in production environments elsewhere.

The big question that needs to be resolved is whether Leases are an adequate mechanism for cache consistency or whether hard server state is required. Given the work presented here and in the papers related to Sprite and Spritely NFS, there are clear indications that a cache consistency algorithm can improve both performance and file system semantics. As yet, however, it is unclear what the best approach to maintain consistency is. It would appear that hard state information is required for file locking and other mechanisms and, if so, it seems appropriate to use it for cache consistency as well.

10. Acknowledgements

I would like to thank the members of the CSRG at the University of California, Berkeley for their continued support over the years. Without their encouragement and assistance this software would never have been implemented. Prof. Jim Linders and Prof. Tom Wilson here at the University of Guelph helped proofread this paper and Jeffrey Mogul provided a great deal of assistance, helping to turn my gibberish into something at least moderately readable.

11. References

[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 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.

[Jacobson88] Van Jacobson and R. Braden, *TCP Extensions for Long-Delay Paths*, ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, October 1988, RFC-1072.

[Jacobson89] Van Jacobson, Sun NFS Performance Problems, *Private Communication,* November, 1989.

[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.

[Juszczak94] Chet Juszczak, Improving the Write Performance of an NFS Server, to appear in *Proc. Winter 1994 USENIX Conference,* San Francisco, CA, January 1994.

[Kazar88] Michael L. Kazar, Synchronization and Caching Issues in the Andrew File System, In *Proc. Winter 1988 USENIX Conference,* pg. 27-36, Dallas, TX, February 1988.

[Kent87] Christopher. A. Kent and Jeffrey C. Mogul, *Fragmentation Considered Harmful*, Research Report 87/3, Digital Equipment Corporation Western Research Laboratory, Dec. 1987.

[Kent87a] Christopher. A. Kent, *Cache Coherence in Distributed Systems*, Research Report 87/4, Digital Equipment Corporation Western Research Laboratory, April 1987.

[Macklem90] Rick Macklem, Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol, In *Proc. Winter 1991 USENIX Conference,* pg. 53-64, Dallas, TX, January 1991.

[Macklem93] Rick Macklem, The 4.4BSD NFS Implementation, In *The System Manager's Manual*, 4.4 Berkeley Software Distribution, University of California, Berkeley, June 1993.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry, A Fast File System for UNIX, *ACM Transactions on Computer Systems*, Vol. 2, Number 3, pg. 181-197, August 1984.

[McKusick90] Marshall K. McKusick, Michael J. Karels and Keith Bostic, A Pageable Memory Based Filesystem, In *Proc. Summer 1990 USENIX Conference,* pg. 137-143, Anaheim, CA, June 1990.

[Mogul93] Jeffrey C. Mogul, Recovery in Spritely NFS, Research Report 93/2, Digital Equipment Corporation Western Research Laboratory, June 1993.

[Moran90] Joseph Moran, Russel Sandberg, Don Coleman, Jonathan Kepecs and Bob Lyon, Breaking Through the NFS Performance Barrier, In *Proc. Spring 1990 EUUG Conference,* pg. 199-206, Munich, FRG, April 1990.

[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.

[Nelson90] Michael N. Nelson, *Virtual Memory vs. The File System*, Research Report 90/4, Digital Equipment Corporation Western Research Laboratory, March 1990.

[Nowicki89] Bill Nowicki, Transport Issues in the Network File System, In *Computer Communication Review*, pg. 16-20, March 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.

[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.

[Srinivasan89] V. Srinivasan and Jeffrey. C. Mogul, Spritely NFS: Experiments with Cache-Consistency Protocols, In *Proc. of the Twelfth ACM Symposium on Operating Systems Principals*, Litchfield Park, AZ, Dec. 1989.

[Steiner88] J. G. Steiner, B. C. Neuman and J. I. Schiller, Kerberos: An Authentication Service for Open Network Systems, In *Proc. Winter 1988 USENIX Conference,* pg. 191-202, Dallas, TX, February 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.

[SUN93] Sun Microsystems Inc., *NFS: Network File System Version 3 Protocol Specification*, Sun Microsystems Inc., Mountain View, CA, June 1993.

[Wittle93] Mark Wittle and Bruce E. Keith, LADDIS: The Next Generation in NFS File Server Benchmarking, In *Proc. Summer 1993 USENIX Conference,* pg. 111-128, Cincinnati, OH, June 1993.

Appendix: Protocol Details

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

RPC Information: Program Number 300105, Version 1

```
readdirlookres
NQNFSPROC_READDIRLOOK(fhandle file; nfscookie cookie; unsigned cnt; unsigned duration) = 18;
union readdirlookres switch (stat status) {
   case NFS_OK: struct {
            entry *entries; bool eof;
```

```
        } readdirlookok;
    default: void;
};
struct entry {
        unsigned cachable, duration; modifyrev rev; fhandle entry_fh;
        nqnfs_fattr entry_attrib; unsigned fid; filename name; nfscookie cookie;
        entry *nextentry;
};
```

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.

```
getleaseres NQNFSPROC_GETLEASE(fhandle file; cachetype rw; unsigned duration) = 19;
union getleaseres switch (stat status) {
    case NFS_OK: bool cachable; unsigned duration; modifyrev rev; nqnfs_fattr attr;
    default: void;
};
```

Gets a lease for "file" valid for "duration" seconds from when the lease was issued on the server.

void NQNFSPROC_EVICTED (fhandle file) = 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.

void NQNFSPROC_VACATED (fhandle file) = 20;

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

stat NQNFSPROC_ACCESS(fhandle file; bool read; bool write; bool exec) = 22;

The access RPC does permission checking on the server for the given type of access required by the client for the file.

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;
};
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.

Data Structures

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

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

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

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

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.

```
struct nqnfs_time {
        unsigned seconds, nano_seconds;
};
struct nqnfs_fattr {
        ftype type; unsigned mode, nlink, uid, gid; unsigned hyper size;
        unsigned blocksize, rdev; unsigned hyper bytes; unsigned fsid, fid;
        nqnfs_time atime, mtime, ctime; unsigned flags, gen; modifyrev rev;
};
struct nqnfs_sattr {
        unsigned mode, uid, gid; unsigned hyper size; nqnfs_time atime, mtime; unsigned flags, rev;
};
```

The attribute structures are modified from the NFS ones so that they store the file size as a 64bit quantity and the storage occupied as a 64bit number of bytes. They also have fields added for the 4.4BSD va_flags and va_gen as well as the file's modify rev level.

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.

```
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 attr;
              } 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".

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

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

```
union nqnfs_statfsres (stat status) {
case NFS_OK: struct {
                unsigned tsize, bsize, blocks, bfree, bavail, files, 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.

Rick is a support technician with the Department of Computing and Information Science at the University of Guelph. He provides operational support for a network of around 90 Novell PCs and 35 Unix systems of various flavours. He has been working with Unix kernels since 1979, including assorted ports and the NFS related work reported in this paper and still uses *ed* (really!). His spare time interests include horses, skiing and canoeing. Rick's electronic mail address is rick@snowhite.cis.uoguelph.ca.

∊ NFS is believed to be a trademark of Sun Microsystems, Inc.

† Prestoserve is a trademark of Legato Systems, Inc.

§ MIPS is a trademark of Silicon Graphics, Inc.

† DECstation, MicroVAXII and Ultrix are trademarks of Digital Equipment Corp.

‡ Unix is a trademark of Novell, Inc.