Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol

Rick Macklem

University of Guelph

ABSTRACT

Since its introduction by Sun Microsystems in 1986, the NFS protocol has become the defacto standard distributed file system protocol for Unix based workstations. Most of these Unix implementations are based on the reference port provided by Sun Microsystems. Research published to date on NFS performance has focused on the problems of NFS server write performance and NFS server performance characterization. This paper discusses other performance and implementation aspects of NFS observed while tuning a rather different implementation of the Sun NFS protocol for Unix. Aspects of performance related to differences in caching mechanisms, the use of different RPC transport protocols and techniques that minimize memory to memory copying are explored. In particular, the notion that TCP transport would provide unacceptable performance for NFS RPCs is shown to be unfounded.

Introduction

There are several aspects of the 4.3BSD Reno implementation of Network File System (NFS)[1] that set it apart from the Sun reference port.[2] In the 4.3BSD Reno implementation, particular emphasis has been put on caching mechanisms, network transport layer independence and the avoidance of memory to memory copy operations. To minimize memory to memory copying and retain network transport layer independence, the NFS remote procedure call (RPC) requests and replies are handled directly in mbuf[3] data areas. Network transport independence permits experimentation with running NFS over other protocols, including TCP. As such, it was felt that by benchmarking this implementation of NFS, we could gain insight into various aspects of performance that have not yet been adequately addressed.

This paper describes the results of benchmarking and tuning in three major areas:

- Server CPU overheads
- Effects of transport protocols
- Effects of different caching mechanisms

In Section 1, a brief overview of the NFS protocol is presented, followed in Section 2 by an overview of the 4.3BSD Reno NFS implementation. Section 3 discusses techniques used to reduce server CPU overhead. Section 4 compares the performance of NFS over a variety of transport protocols operating on three different internetwork topologies. Following this, a comparison in Section 5 of 4.3BSD Reno NFS with Ultrix† NFS is used to identify significant differences related to caching mechanisms. The conclusion summarizes the results and suggests areas of distributed file system performance that

---

[1] NFS is a trademark of Sun Microsystems Inc.

[2] Along with the published NFS specification, Sun Microsystems licensed a reference port of NFS which forms the basis of most commercially available NFS systems. Since I have no access to this code, information about its structure was gleaned from a variety of publications. Apologies for any inaccuracies w.r.t. this port.

[3] **mbuf** is the Berkeley Unix structure for handling network buffers.

† MicroVAXII, DECstation and Ultrix are trademarks of Digital Equipment Corporation

require further investigation.

1. Overview of NFS Protocol

The NFS protocol is a remote procedure call (RPC) based distributed file system that does I/O at the level of logical blocks of files. These data blocks start at an arbitrary byte offset and range in size from 1 to 8192 bytes. The server is stateless, which implies that RPC requests are atomic operations where all request related information must be stored in the RPC request. The stateless server concept was used so that crash recovery is trivial. However, there are some obscure implications on performance in the areas of client cache consistency and write policy.[4] The write policy for NFS is **asynchronous** for full blocks and **delayed** when partial blocks are written. The delayed writes must be pushed when the file is closed and are also pushed every 30sec for most Unix[5] implementations. Cached data consistency is maintained with the server by checking that the file's modify time has not changed since the cached data was read from the server. Since most implementations also cache file attributes for a few seconds, this implies that cached data will be consistent with that of the server to within a few seconds. However, the stateless server does not know about any delayed writes to a file from other clients. By pushing delayed writes on close, NFS maintains a close/open consistency criteria when more than one client read/write shares a file. That is, a client opening file "X" for reading after another client that was writing to file "X" does a close, is guaranteed to see those changes.

The NFS RPCs are done using Sun RPC, which stores all fields of the requests and replies in an architecture-independent data format, called the external data representation (XDR). For the Sun reference port, a user mode runtime library that implements these layers, was ported into the kernel, and NFS was implemented using this library interface.

2. 4.3BSD Reno NFS Implementation

The 4.3BSD Reno NFS is implemented in the kernel without the use of any XDR or RPC interface layers. All NFS RPC requests and replies are constructed and decomposed directly in mbuf data areas using two macros **nfsm_build** and **nfsm_disect**. These two macros are then used by higher level functions and macros to access the fields of the NFS RPC request and reply packets. Most of the translation to/from XDR is done by inline code, except for a few special cases that are handled by functions. There were two reasons for this approach, namely to:

- Avoid the use of a buffer that would have to be copied into an mbuf list.
- Avoid the need for a special type of mbuf that might not work well with transport protocols other than UDP.

Once the request or reply has been converted into an mbuf list, the list is passed onto the socket interface code which deals with the vagaries of the various types of sockets. For datagram sockets, the client side provides round trip timeout (RTO) estimation and requests retransmission upon timeout. For stream sockets such as TCP, it maintains the connection and provides record marks between each RPC request/reply, along with concurrency control on the socket I/O routines.

Caching is done for **name lookups**, **data blocks** and **directory blocks**, using the VFS caching mechanisms which are discussed in greater detail in Section 4. The client side cache consistency is controlled by the file/directory **modify time**, and cached data is flushed whenever the modify time changes, as reported by the server. The file attributes are cached and time out five seconds after being updated from the server. This appears to be similar to the level of consistency that was observed experimentally on a SunOS NFS client.

---

[4] Write policy defines the client action when a write to a remote file is done. It may be **write through** which implies: do the write RPC and wait for the reply before returning from the system call. **Asynchronous**, implies start the write RPC but do not wait for its completion. **Delayed** means do the write RPC sometime later.

[5] Unix is a trademark of AT&T

3. Server Structural Changes and CPU Overhead

Most current NFS servers tend to be CPU bound, which makes minimizing server CPU overhead of interest. To study this, the kernel of a system that was running under heavy NFS server load was profiled to identify bottlenecks. It was observed that over a third of the CPU cycles were being used by the low level network interface handling code. In particular, the routine that copied the mbuf data areas to the network interface's transmit buffers was at the top of the CPU utilization list.

In an effort to reduce CPU overhead in the network interface code, two changes were made:

- Network interface buffer handling was modified to allow the mapping of a packet to two noncontiguous buffers for transmission, one for the IP fragment header and the other for the mapped mbuf data clusters. This allowed the *copying* of mbuf clusters to network interface buffers by page table entry swaps instead of by actual memory to memory copying.
- The network interface device driver was modified to remove the transmit interrupt service routine. Since this routine simply released buffers and updated I/O statistics, it was possible to disable transmit interrupts and perform the operations in the transmit startup routine, reducing the number of network interface interrupts. [Jacobson89] The transmit startup routine was also fine tuned by careful use of register variables and unrolling of loops.

After the above changes, CPU overhead was reduced by approximately 12%. Most of this was a reduction in memory to memory copying. Since memory to memory copy bandwidth has not grown with MIPS rate for many recent computer systems [Ousterhout90], this may be even more significant on newer hardware architectures.

At this point, the CPU bottlenecks were the network interface startup routine, the internet checksum calculation routine and the routine that copies data between the buffer cache and mbuf clusters. Since the first two bottlenecks have already been fine tuned, the only area that deserves further attention is the third. It may be possible to avoid the buffer cache to mbuf cluster copying by implementing a mechanism where page clusters in the buffer cache may be borrowed as mbuf page clusters and returned after network transmission. This was not done, due to the complexity of the code, but is a possible area for further work.

4. RPC Transport Issues

The NFS protocol normally runs on top of UDP transport where each RPC request and reply is packaged in one UDP datagram. Since UDP datagrams are not delivered reliably, the NFS client sets a retransmit timeout (RTO) when a request is sent and resends the request if the RPC reply is not received within the RTO. The initial RTO is set to a constant value defined at mount time and is backed off exponentially upon retransmits. This transport mechanism is adequate when the client and server reside on the same high bandwidth LAN cable, but performance has been observed to deteriorate over more complex internetwork connections. [Nowicki89] This problem stems in part from the fact that an 8Kbyte read or write RPC must be transmitted as IP fragments the size of the interconnect's Maximum Transmission Unit (MTU). (eg. 6 IP fragments for an Ethernet) There are serious problems with IP fragmentation, such as the need to resend the entire datagram if any one fragment is lost in transit, making this a poor transport mechanism for any but the most reliable network interconnects. [Kent87b] Since the 4.3BSD Reno implementation is transport layer independent, an experimental evaluation of performance over other transport mechanisms seemed appropriate.

The first alternate transport mechanism was a reliable virtual circuit (TCP) protocol with dynamic RTO estimation and congestion control. [Jacobson88a] Although others, [Chesson87] had suggested that CPU overheads might be excessive, it was felt that the advantages of reliable transport with congestion control might outweigh the increased CPU overheads when using congested internetwork connections. Early informal observations indicated that TCP ran well enough and led to further interest in the next alternative.

The other alternate transport mechanism used UDP, but with dynamic RTO estimation and a congestion window on outstanding requests modelled after that of TCP. The advantage of this approach over TCP is that it does not *break* the NFS protocol and works with existing NFS servers. Trace data of round trip time (RTT) for the NFS RPCs indicated that different RPCs had vastly different RTTs and that

the variance of *big* RPCs (Read, Write, Readdir) was higher than that of the *small* RPCs. (Getattr, Lookup) As such, it was decided to do separate RTO estimation on the four most frequent RPCs, **Read, Write, Getattr** and **Lookup** and use the constant value provided by the mount for the others. Since the *others* occur infrequently, it was felt that dynamic RTO estimation was impractical. Also, since most of these other RPCs are nonidempotent [Juszczak89], a conservative RTO is desired to minimize the risk of redoing the RPC. This design was somewhat similar to [Nowicki89] who used three timers and an overall estimation.[6]

The criteria for initial testing of these changes to UDP transport was that there should be no significant negative impacts when compared with the old UDP transport running on a single LAN. Early test runs showed that the retry rate for read RPCs was 2-4 times that of old UDP.

Two changes were made to bring the retry rate down:

- The calculation of RTO for the *big* RPCs (Read, Write, Readdir) was changed from "A+2D" to "A+4D" to allow for the large variances.[7]
- The RTO was recalculated on every NFS clock tick instead of at request transmission time, so that the most current values of A and D were used.

It was also found that "slow start" impacted performance and had to be removed from the code. As a result, the congestion window on the number of outstanding RPCs is simply incremented by one for each RTT upon reception of an RPC reply and divided by two upon a retransmit timeout.

The experiment consisted of running an NFS RPC load between a client and server interconnected in three ways:

1. Both machines on the same uncongested Ethernet
2. Machines on two Ethernets interconnected by an 80Mbit/sec token ring and two IP routers.
3. Machines on two Ethernets interconnected by an 80Mbit/sec token ring, a 56Kbs point to point link and three IP routers.

The NFS load was generated by the Nhfsstone[8] [Legato89] benchmark using two different load mixes: a 100% lookup RPC and a 50/50 lookup/read RPC. Since the object here was to measure the effects of different transport mechanisms, all that was required were *big* and *small* RPCs. Any RPCs that modify the underlying file system were avoided so that the subtree would remain stable and not require reloading between each test run. The 50/50 read/lookup load mix was selected since these are the most frequent *big* and *small* RPC's plus the fact that Nhfsstone requires a high percentage of lookup RPCs to function well. The 100% lookup load mix was chosen to allow factoring out of the effect of lookups on the above. Each point in graphs #1-5 represent a test of 30min, to avoid momentary variations caused by other network loads. There were two runs done for each of the *(transport, internetwork-configuration)* tuples and each of these is represented by a line on one of the graphs. Since these tests were run across production networks during off peak hours, the other network loads were realistic but were not controlled nor reproducible. As such, it is probably the shape of the curves that is more relevant than the RTTs of individual data points. In the case of the 56Kbps link, after hours involved almost no other loads.

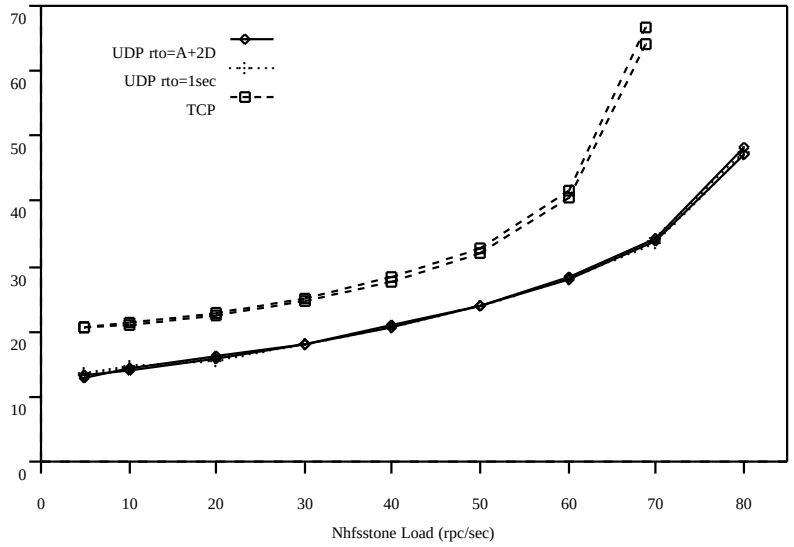
---

[6] My implementation was actually based on work done by Tom Talpey of the OSF. I was not aware of the work done by [Nowicki89] until later. It was not obvious to me what Nowicki meant by *overall estimation.*

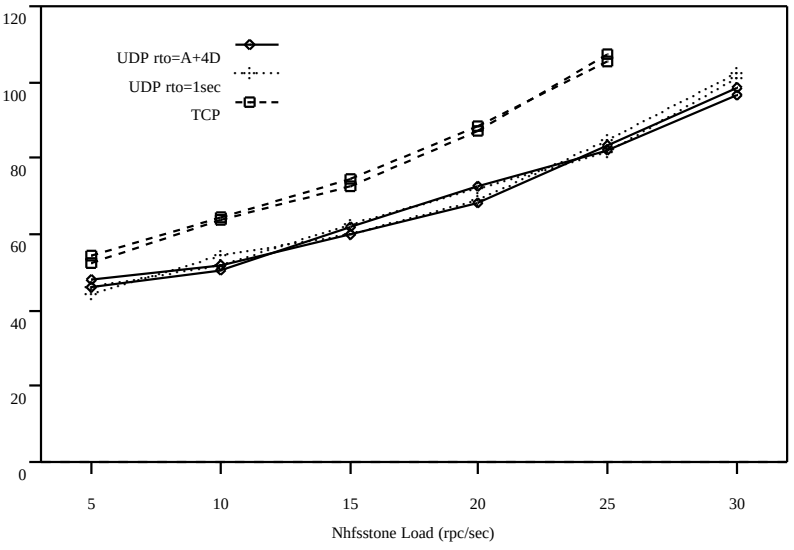
[7] A is the estimated mean and D the estimated mean deviation of RTT

[8] Nhfsstone is a trademark of Legato Systems Inc.

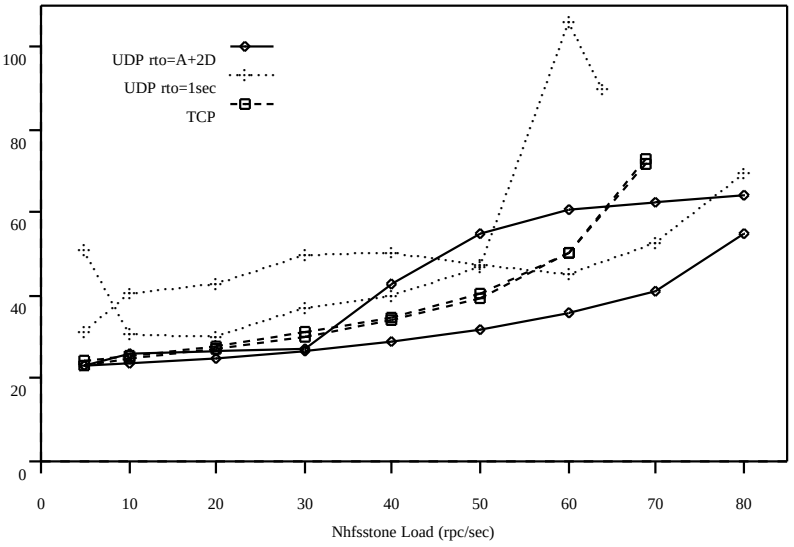
Graph #1, Lookup mix Ave RTT (Config #1, 2 runs on an Ethernet)



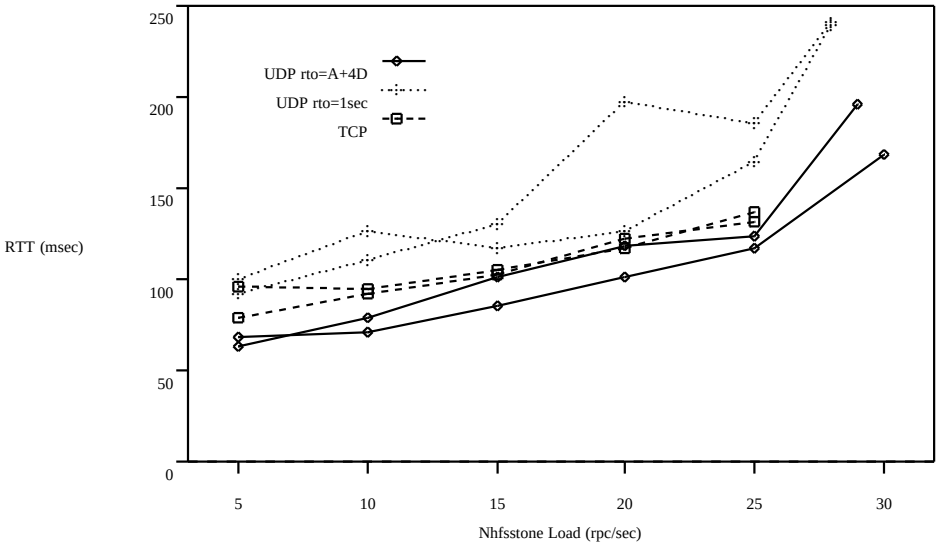
Graph #2, Read mix Ave RTT (Config. #1, 2 runs on an Ethernet)



Graph #3, Lookup mix Ave RTT (Config. #2, 2 runs across 80Mbit)



Graph #4, Read mix Ave RTT (Config. #2, 2 runs across 80Mbit)



Graph #5, Lookup mix Ave RTT (Config. #3, 2 runs across 56Kbit)

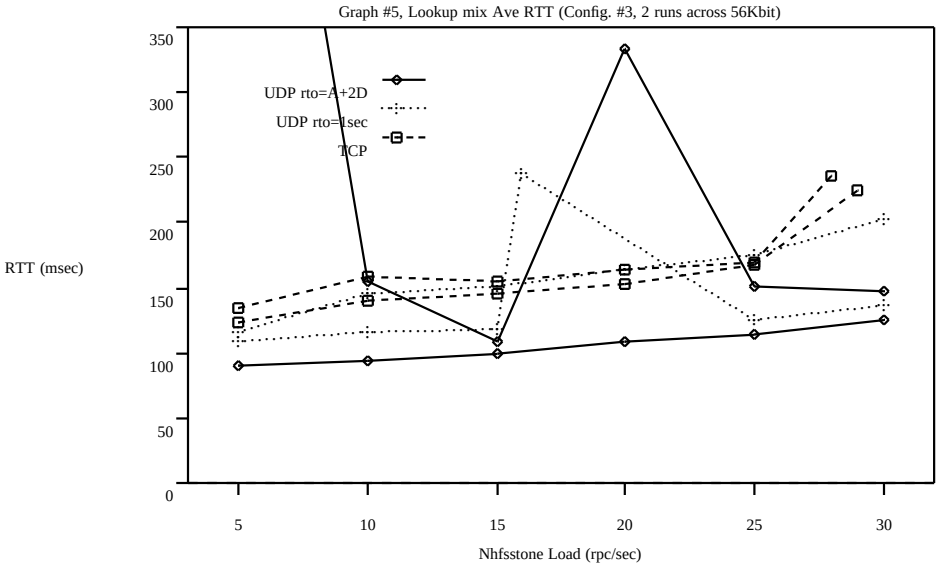
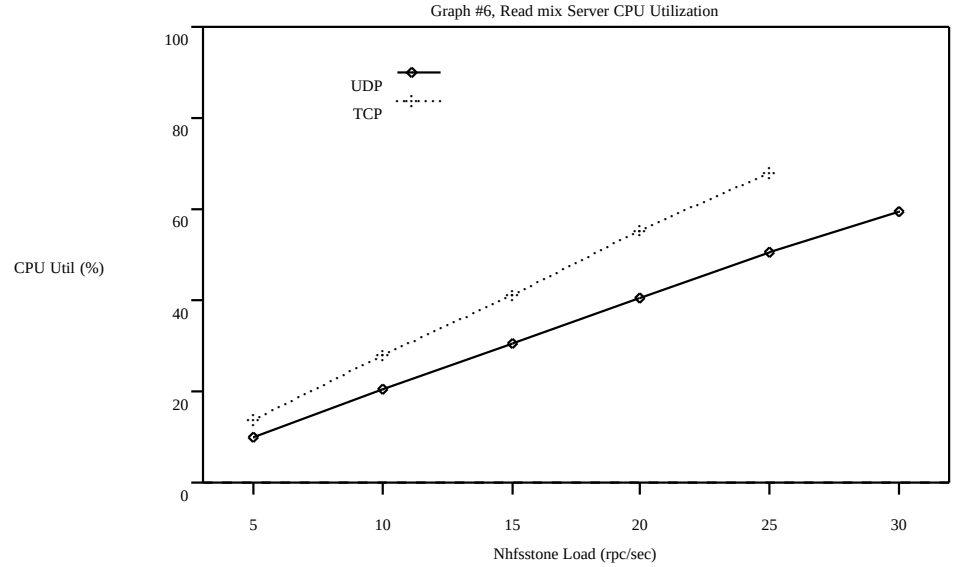
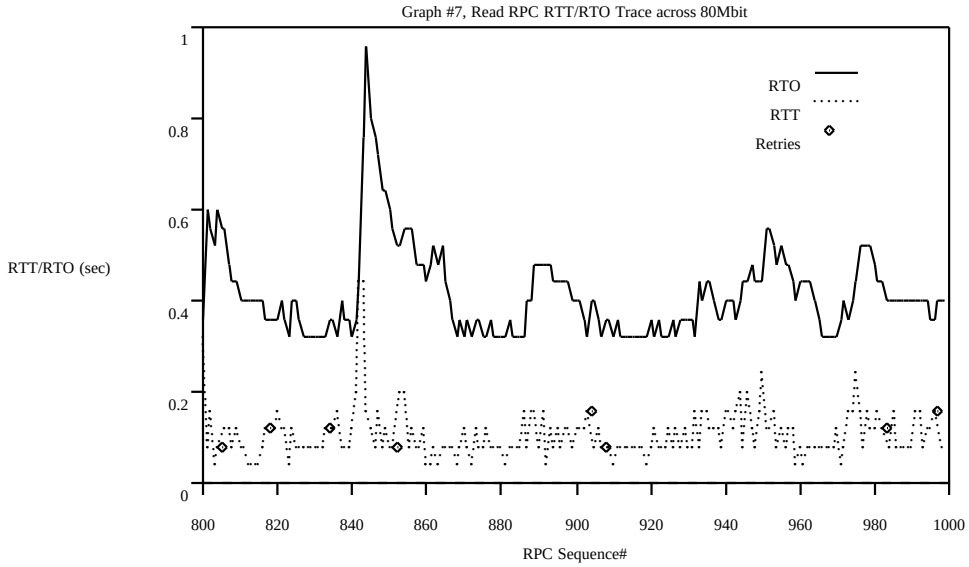


Table #1
Read Rate, large file (Kbytes/sec)

Config. / Transp.	#1 LAN	#2 80Mbit	#3 56Kbit
UDP rto=A+4D	202	154	6.21
UDP rto=1sec	198	117	1.77
TCP	177	106	6.38

Graph #6 compares the server CPU overhead of UDP and TCP for an Nhfsstone read RPC mix and Graph #7 is a sample trace of RTT and RTO equal A+4D for read RPCs.





Contrasting Graph #1 with #3 and #5, Graph #2 with #4 and examining Table #1, a variety of observations can be made:

- Graphs #1-2

  When both the client and server were on the same LAN, the method used to set RTO for UDP is not relevant. The RTTs for TCP are higher by a fixed amount of approximately 7msec for lookups and 10msec for the read mix until the server is under heavy load. For the read mix, much of this increased delay can be attributed directly to the higher CPU overhead associated with TCP. (7msec/rpc)[9] As for lookups, the increase in CPU overhead is only 1msec/rpc for TCP, so there must be some other factor introducing real time delay. (Possibly more packets through the DEQNA Ethernet interface, which is *real slow*)

- Graphs #3-4

  When the client and server were interconnected through the 80Mbit token ring and gateways, the differences start to become apparent. The TCP curves are almost identical, indicating a high degree of stability. The curves for UDP with dynamic RTO estimation are somewhat more variable than

[9] The test machines were 0.9MIPS MicroVAXIIs and as such a small amount of processing takes several msec. (Also See Graph #6)

TCP but with equal or better average RTTs, due to the lower CPU overheads. However, the curves for UDP with a fixed 1sec RTO are more erratic, due to the long delays before retransmits. At first glance, this would suggest that 1sec was too large, but examination of RTT trace data had peaks for Read RPCs at close to 1sec, which suggests that lowering the constant would not be advisable. The read rates for UDP with fixed RTO and TCP are almost the same, suggesting that the gains resulting from congestion control are cancelled out by the delay introduced by the higher CPU overhead. In this case, the simple congestion control added to UDP has improved read rate by about 30% over the other transport methods.

- When running across the 56Kbps link, the tests could only be run for the lookup mix.[10] As graph #5[11] and Table #1 indicate, UDP with a fixed 1sec RTO did not perform as well as either of the others. In this case, TCP performed consistently well and UDP with dynamic RTO estimation and congestion avoidance was often equal to TCP, but at times became unstable. The advantage of providing congestion control for this kind of network interconnect becomes apparent when you look at the read rates in Table #1. The read rates for TCP and UDP with dynamic RTO and congestion control are over three times that of UDP with fixed RTO.

There is another aspect of UDP transport for NFS and that is the choice of read/write data size. All of the above tests were run with the default 8Kbytes, but there are situations where decreasing the read/write size might improve performance. The difference in read rate between the two versions of UDP transport across the 56Kbyte link suggests that a congestion avoidance scheme may be sufficient for most situations, so that this is not normally required. Decreasing the read/write size increases the number of RPCs and as such should be considered as a last ditch action when all else fails. Since the trick here is to avoid IP fragment loss, it may be possible to adjust the size dynamically, based on the IP fragment drop rate. (This has not yet been tried, but is an area for further work.)[12]

5. Client Side Caching Issues

The 4.3BSD Reno NFS implementation uses several caching mechanisms that are believed to be somewhat different from those of the Sun NFS reference port. The 4.3BSD Reno VFS[13] layer buffer cache is used by the NFS client to cache regular file blocks, directory blocks and symbolic links. There are references to these cached blocks hanging directly off of the vnodes. For writing of partial buffers there is no need to preread the blocks from the server, since there are additional fields in the buf[14] structure for keeping track of the "dirty" region within the buffer. File attributes are cached in the associated vnode[15] structure and there is also a VFS layer name lookup cache in 4.3BSD Reno. Any performance gains that could be related to differences in the caching mechanisms could suggest future work related to caching mechanisms.

An experimental mount flag that disables all of the NFS cache consistency mechanisms was implemented. Although operating NFS in this way is not practical in a production environment, it was done to allow determination of an optimistic bound on performance of a system with a cache consistency protocol.[16] This is of interest, since distributed file systems such as Sprite have been observed to outperform NFS.[17] [Nelson88] Another issue related to caching is, what is the best write policy?

---

[10] The upper bound on the number of 8Kbyte reads over a 7Kbyte link is < 1/sec.

[11] For one of the UDP rto=A+4D runs, the Ave RTT for 5rpc/sec was 721msec and therefore off the graph.

[12] [Nowicki89] describes some difficulties w.r.t. dynamically adjusting read/write size, but does not explain how they resolved the problems.

[13] VFS refers to the Virtual File System described in [Karels86].

[14] **buf** is the Berkeley Unix structure for handling block I/O buffers.

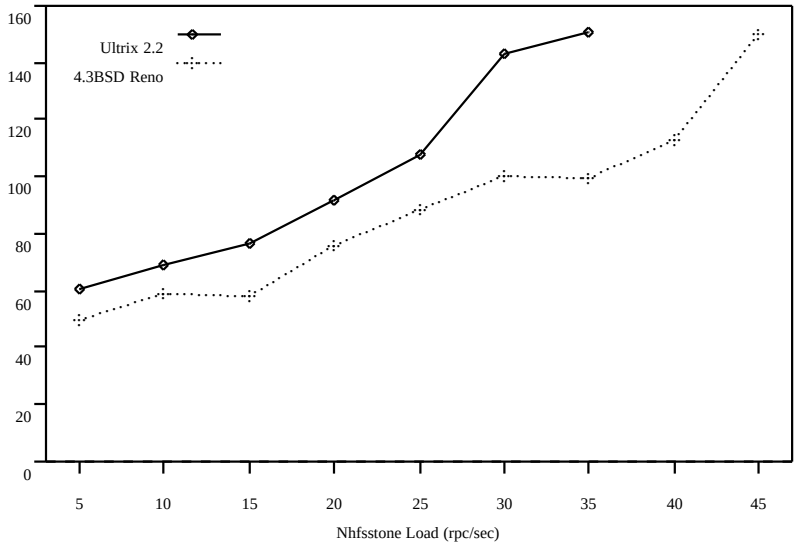
[15] **vnode** is the structure in Berkeley Unix for a file object. See [Karels86]

[16] Essentially a cache consistency protocol without overheads.

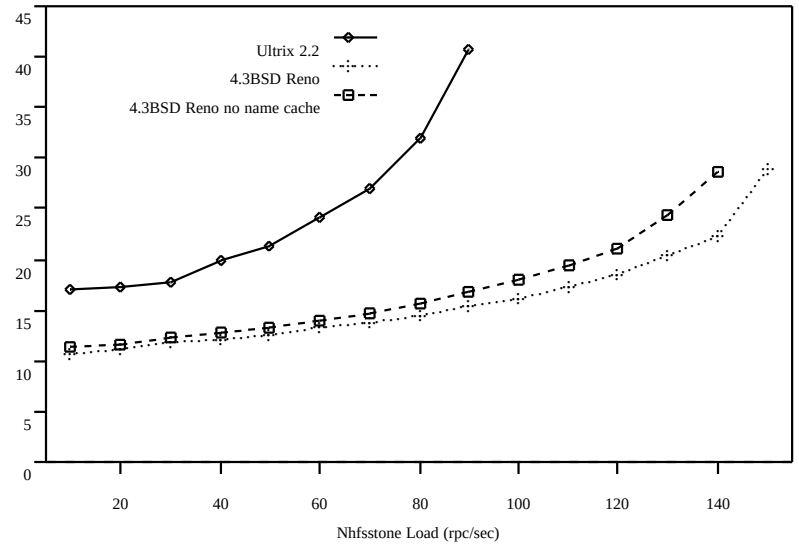
[17] Srinivasan et al were not able to achieve the performance gains of Sprite by adding a sprite like cache consistency protocol to NFS. They believed that a major reason for this was the large number of lookup RPCs that predominated. Since 4.3BSD Reno's name lookup cache reduces the number of lookup RPCs significantly, better performance improvements might be expected.

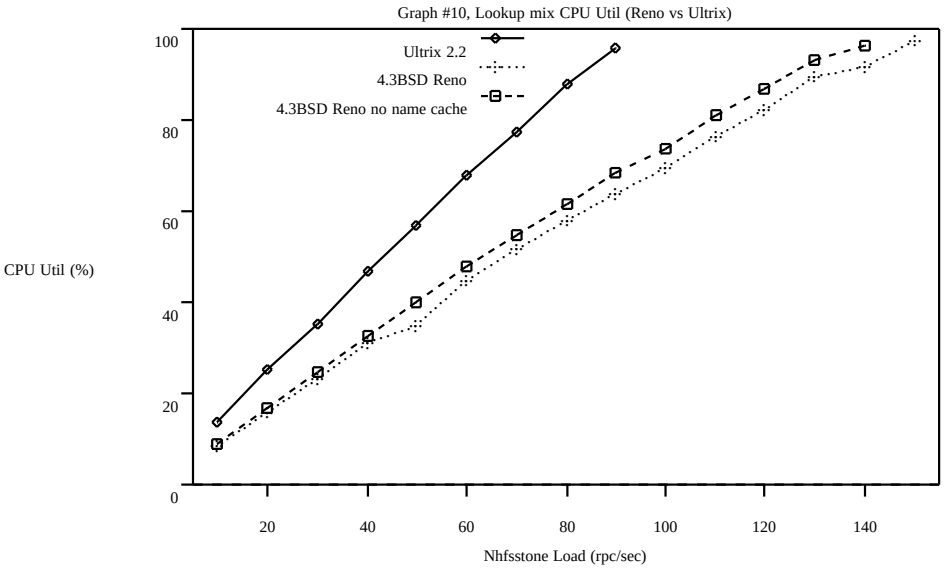
The performance of a distributed file system implementation is closely coupled with network interface, disk I/O subsystem and processor performance. As such, use of identical hardware is required to isolate hardware related performance effects. A comparison with an implementation based on the Sun reference port was performed by benchmarking a MicroVAXII running both 4.3BSD Reno and Ultrix Version2.2. With these two systems as servers, Nhfsstone loads were run on them from a client on the same LAN.

Graph #8, Default mix Ave RTT (Reno vs Ultrix)



Graph #9, Lookup mix Ave RTT (Reno vs Ultrix)





Graph #7 showed significant differences between 4.3BSD Reno and Ultrix. In order to isolate the basis of the differences, I ran further tests with 100% lookup and 50/50 read/lookup load mixes. The most significant difference was the lookup RPC performance, as seen in Graphs #8-9. An obvious explanation for the difference was the VFS name lookup cache on the 4.3BSD Reno server. However, disabling this cache only reduced the performance of 4.3BSD Reno by a small fraction of the difference observed compared to Ultrix. A possible explanation for the remainder of the difference is that on 4.3BSD Reno, the directory blocks in the server's buffer cache are chained directly off of the vnodes, reducing the CPU overhead for buffer cache searches.

For client side testing, the Modified Andrew Benchmark [Osterhout90] was used with both systems mounting the same file system on the same server. Since almost any *real work* is CPU bound on a MicroVAXII, the RPC counts in Table #3 are of more interest than the running times.

Table #2
Mod Andrew Bench MicroVAXII client (sec)

OS/Phase	I-IV	V
Reno	145	1253
Reno-TCP	143	1265
Reno-nopush	132	1208
Ultrix2.2	184	1183

Table #3
Mod Andrew Bench MicroVAXII client (RPC counts)

RPC	Reno	Reno-noconsist	Ultrix2.2
Getattr	822	780	877
Setattr	22	22	22
Read	1050	619	691
Write	501	340	703
Lookup	872	918	1782
Readdir	146	144	150
Other	127	128	127
Total	3540	2951	4352

The biggest differences between 4.3BSD Reno and Ultrix were the number of lookups and the number of reads. The VFS name lookup cache on 4.3BSD Reno has reduced the number of lookup RPCs by 50%. This is significant, since lookup RPCs are usually the largest percentage of RPCs observed on production servers. The number of write RPCs was reduced by over 50% by disabling cache consistency. This

implies a big reduction in server load, since every write RPC requires 1-3 disk writes on the server. The number of read RPCs for 4.3BSD Reno was 50% higher than Ultrix, and this can be traced to the fact that the 4.3BSD Reno NFS pushes all "dirty" blocks to the server before it starts reading a file. The argument for this is that after doing a write RPC, the modify time has changed, but the client cannot tell whether this modify was due to changes it made or to writes just done by other clients to the same file. It appears that Ultrix assumes that other clients are not writing to the same file at the same time and therefore regards data in the cache as still valid. As such, it may be worthwhile to rethink the above consistency criteria for 4.3BSD Reno.

The Modified Andrew Benchmark was also run on a DECstation 3100 against both servers to see what effect the server differences would have on real work.

Table #4
Mod Andrew Bench DS3100 client (sec)

OS/Phase	I-IV	V
Reno	88	180
Ultrix2.2	123	226

The results in Table #4 show a difference of 20-30% between the two servers.

To look at the effects of different write policies, the Create-Delete benchmark [Ousterhout90] was run with and without cache consistency. The tests were run with zero, four and sixteen biods,[18] to simulate different levels of asynchronous I/O concurrency. With no biods running, the write policy becomes *write through*.

Table #5
Create-Delete Bench 4.3BSD Reno MicroVAXII (msec)

Config	No data	10Kbytes	100Kbytes
Local	120	216	1170
write thru	210	475	2401
async,4biod	216	470	1940
async,16biod	210	464	2094
delay wrt.	216	468	2230
no consist	218	244	329

When maintaining close/open consistency by pushing writes on close, the only time that selection of write policy is significant is for large files. For the 100Kbyte file, it was observed that an *asynchronous write* policy was about 20% faster than *write through* or *delayed write*. However, there is a big improvement if you do not **push writes on close**[19] due to the fact that there is usually no need for the write system call to block waiting for write RPCs to complete. Also, the number of write RPCs is dramatically higher for asynchronous writes than for the delayed write without push on close, (Table #3) suggesting that there is a good argument for this approach based on reduced server load. (Also see [Nelson88]) Note however, that to do this for a production environment would require the addition of some sort of **cache consistency protocol** to NFS.

Conclusions

The performance of an NFS implementation is influenced by caching performance for the client and caching plus CPU overhead for the server. Most current NFS servers have observed loads that are lookup RPC dominant. A good lookup name cache on the client can reduce the lookup RPC load significantly, causing the performance of the read/write RPCs to become more dominant. The read/write RPC performance of a server can be significantly improved by minimization of memory to memory copying and tuning of the low level network interface handling code.

[18] **biod** is a daemon that does asynchronous I/O for client NFS

[19] **don't push writes on close** is the major effect of disabling cache consistency

Two of the major limitations of NFS are actually a result of the implementation of Sun RPC on UDP transport. The *at least once* semantics of these RPC's can result in faulty behaviour on a heavily loaded server, due to the repetition of non-idempotent RPCs. Also, the simple timeout/retransmit scheme used to achieve reliability is inadequate for all but the most reliable client/server interconnects. Serious degradation of performance has been observed across even a single IP gateway. Early evidence suggests that UDP transport can be improved by dynamic RTO estimation and a congestion window modelled after that used by TCP. It has also been found that TCP performs fairly well as an NFS transport mechanism, with an increase in CPU overhead of about 20% over UDP.

A cache consistency protocol would reduce the number of write RPCs by at least half.

Future Directions

As CPU speed increases, real work becomes less CPU bound and more sensitive to I/O performance [Ousterhout90]. As such, a performance evaluation of the client side running on a 20 MIPS workstation could yield further insight into appropriate client side caching mechanisms. In particular, with a reduction in lookup RPC rate due to a name lookup cache, it may be possible to achieve higher performance gains from a Sprite like cache consistency protocol than was observed by Srinivasan et al. [Srinivasan89] The experimental mount option for *don't do cache consistency* permits determination of an optimistic bound on the performance gains of such a Sprite like cache consistency protocol but does not solve the problem. A cache consistency protocol that is crash and network partition tolerant is still needed. A question here is whether full cache coherency is required or simply a mechanism for doing a *delayed write without push on close* policy safely.[20]

More work needs to be done on good transport mechanisms for RPC's. An improved timeout/retransmit scheme for UDP would be a first step, since there are so many NFS/UDP servers out there today. However, in the future I believe that UDP needs to be replaced as a transport mechanism for RPC's.

It would be desirable to construct some sort of experimental test bed to explore performance issues related to many gateway hops and *long fat pipes*. [Jacobson88b] Such a testbed could be used for experimentation with tranport mechanisms and caching techniques better suited to large delay paths. To achieve good performance in these internetworks, the number of times that an I/O system call blocks for an RPC reply[21] must be minimized. This would be achieved in part by a cache consistency protocol. However, I think that you must also do more cache preloading. There are many possibilities here. For reads, you might either increase the size of the read RPCs or the level of read-aheads[22] or both, so that most read system calls find the data already in the local cache. I think that you also need a way of doing many name lookups per RPC, possibly by adding a **readdir_and_lookup_files** RPC to the protocol.

Acknowledgements

I am indebted to the people at the Computer Systems Research Group of the University of California Berkeley for their friendly poking and prodding over the past couple of years. Professors Jim Linders and Tom Wilson here at Guelph provided much needed guidance and support. The OSF has provided both moral and financial support for this activity and DEC helped immensely by providing a 50% equipment grant. I must also thank the personnel of Computing Services and Communications Services on the Guelph campus for permitting the use of their facilities for benchmarking.

---

[20] This is not meant to imply that a *delayed write without push on close* protocol that retains close/open consistency criteria, handles *disk full* errors and server crashes is simple.

[21] [Osterhout90] refers to this as decoupling I/O.

[22] Normally Unix does a read-ahead of 1 block. By increasing the level of read-ahead, I mean doing a read-ahead of the next 2-4 blocks.

Appendix. Experimental Details

All tests were performed on identical hardware, MicroVAXIIs' with RD53 disks and a DEQNA ethernet interface attached to either a lightly loaded Ethernet or the internetworks described in Section 4. Although not representative of current hardware, these systems demonstrate relatively well balanced performance (ie. slow CPU, slow disks and a slow network interface) and were the only systems available that would run both 4.3BSD Reno and a vendor implementation of NFS based on the Sun reference port. The emphasis was placed on the four RPCs *getattr, lookup, read and write*, since these make up a majority of most NFS RPC workloads. For client side benchmarking, the same server was always used. For the Modified Andrew Benchmark, the DECstation 3100 was selected as it has a sufficiently fast processor so that real work, such as C compilation, is not entirely CPU bound. For comparisons between 4.3BSD Reno and the vendor kernel *(Ultrix Version 2.2)*, the kernels were configured with identically sized buffer caches and file systems.

The percentage of idle CPU as reported by *iostat(1)* was observed to be erratic during early test runs. The cause of this was found to be a hardware constraint of the MicroVAXII, which masks off clock interrupts during peripheral interrupts. To avoid this problem, all kernels were patched with a counter inside the idle loop to allow for an accurate measure of CPU utilization. This is a particularly handy bit of instrumentation, since it does not have any adverse effect on real performance due to the fact that the instrumentation overhead is only incurred when the CPU is idle.

Two caveats were identified in the Nhfsstone server characterization benchmark as follows:

1) The Nhfsstone benchmark uses long file names to defeat client name caching, but this can also defeat server name caching. This will tend to bias against servers with good lookup name caches. To determine the extent of this problem, the lookup benchmark was run against a server with and without name caching enabled.[23]

2) The Nhfsstone benchmark chooses a file at random and then performs a random operation on it in proportion to its load mix. Since most load mixes have a small proportion of writes *(8% is the default)*, starting with empty test directories causes most files to remain empty during the test interval. This implies that most reads are performed on empty files and biases the results against a server with good read performance. Further, as testing continues, more files are written reducing the number of empty files. This results in the average RTT increasing over time, due to the fact that fewer of the reads are of empty files. To avoid this side effect, the subtree was preloaded with an identical set of files before each test.

References

[Cheriton86]
David R. Cheriton, VMTP: A Transport Protocol for the Next Generation of Communication Systems, In *Proc. SIGCOMM 86 Symposium on Communications Architectures and Protocols*, pg. 406-415, Stowe VT, August 1986.

[Chesson87]
G. Chesson, Protocol Engine Design, In *Proc. Summer 1987 USENIX Conference*, Phoenix, AZ, June 1987.

[Jacobson88a]
Van Jacobson, Congestion Avoidance and Control, In *Proc. SIGCOMM 88 Symposium on Communication Architectures and Protocols*, pg. 314-329, Stanford, CA, August 1988.

[Jacobson88b]
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.

[23] 4.3BSD Reno name caches file names up to 31 characters, which is longer than the names used by the Nhfsstone benchmark.

[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.

[Karels86]
Michael J. Karels and Marshall Kirk McKusick, Toward a Compatible Filesystem Interface, In *Proc. EUUG Conference*, September 1986.

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

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

[Kent87b]
Christopher A. Kent and Jeffrey C. Mogul, Fragmentation Considered Harmful, In *Proc. SIGCOMM 87 Workshop on Frontiers in Computer Communications Technology*, pg. 390-401, Stowe, VT, August 1987.

[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.

[Nhfsstone]
Nhfsstone NFS load generating program, Legato Systems Inc., Palo Alto, CA, 94306.

[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.

[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.

[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.

[RFC1094]
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.