Improving the Performance and Correctness of an NFS Server

Chet Juszczak

Digital Equipment Corporation
110 Spit Brook Road ZK03-3/U14
Nashua, New Hampshire 03062
(603) 881-0386
chet@decvax.dec.com

ABSTRACT

The Network File System (NFS) utilizes a stateless protocol between clients and servers; the major advantage of this statelessness is that NFS crash recovery is very easy. An NFS client simply continues to send a request until it gets a response from the server. However, this client retry model also has disadvantages: a server can receive multiple copies of the same request. The processing of duplicate requests is an expense of server effort that is better spent elsewhere. Worse than that, it can result in incorrect results. This paper describes a *work avoidance* technique that utilizes a cache on the server to avoid the needless processing of duplicate client requests. An implementation of this technique has resulted in a significant increase in server bandwidth. A beneficial side effect is that it can help avoid the destructive re-application of non-idempotent operations. It can be used in any NFS server implementation, requires no client modifications, and in no way violates the NFS crash recovery design.

1. Introduction

The Network File System (NFS)[1] has become a standard in the UNIX[2] industry. The NFS utilizes a stateless protocol between clients and servers. This means that an NFS server is not required to keep any information (state) about a client request after it has been performed. Each NFS request contains all the information necessary for the server to perform an NFS operation [1].

The major advantage of this statelessness is that NFS crash recovery is very easy. Neither client nor server must detect the other's crashes. Since a server has no state information to maintain, there is nothing for it to throw away after a client crashes. Likewise, there is no state information to re-build when the server returns after a crash. An NFS client simply continues to send (retransmit) a request until it gets a response from the server. (Optionally, the client can give up after a number of retries specified at mount time.) This client retry model solves other service problems such as network disruptions, data loss at the server's network interface, and overflow of the server's input queue.

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

[2] UNIX is a registered trademark of AT&T.

However, the NFS client retry model also has disadvantages. To an NFS client, a server that is down simply looks like one that is slow to respond. Problems arise because a server that is slow to respond simply looks like one that is down.

Due to retransmitted requests, a slow or busy server can receive multiple copies of the same request. In fact, even the quickest of servers can receive duplicate requests from an impatient client. The processing of these duplicate requests is a waste of server effort that is better spent servicing other requests, perhaps from other clients. Worse than this inefficiency, duplicate request processing can result in incorrect results (affectionately called 'filesystem corruption' by those not in a filesystem development group). The *lost writes* seen at MIT by Project Athena [2] are the result of an NFS server processing duplicate client requests for a file truncation operation.

Included in the Ultrix[3] V2.2 diskless workstation project was the goal of improving the performance of our NFS implementation. Since all of the diskless workstation's files would be accessed via NFS, its filesystem performance would be equivalent to its NFS performance. To a large extent, a client's NFS performance is determined by its server's performance.

The write operation is the most costly of all NFS server operations (see *NFS Writes*, below). A single client can have multiple outstanding write requests, and a diskless client generates more NFS write requests, on average, than a diskful one. This follows from using NFS for paging, swapping, and */tmp*.

An investigation of diskless client performance [3] has shown that a heavy write load results in poorer client performance than loads of other types. Detailed server analysis indicated that when under heavy write load, servers expended considerable effort servicing duplicate requests.

A *work avoidance* technique is described that utilizes a cache on the server to avoid the needless processing of duplicate client requests. An implementation of this technique has resulted in a significant increase in server bandwidth, especially with low end server configurations. A beneficial side effect is that the destructive re-application of a duplicate operation is much less likely.

There is no need to re-build the previous contents of the cache after a server crash. Therefore, the use of this cache violates neither the statelessness of the NFS protocol nor its crash recovery design. No NFS client side modifications are necessary, although one change is suggested that makes the technique more effective (see *Results and Recommendations*).

2. Background

This section contains some necessary background information on various NFS topics. References to "typical" NFS clients and servers refer to implementations of NFS derived from the 4.3BSD based kernel implementation available from Sun Microsystems, Inc. Similarly, the term *reference port* will refer to this implementation. Our work was done with version 3.2 of the reference port. The following characterizations may apply to other NFS implementations as well.

2.1. NFS Clients and Servers

NFS client systems range in size from small workstations to large multi-processor timesharing machines with hundreds of users.

A typical NFS client system will retransmit a request if it has not received a response from the server for that request within an interval of time that defaults to .7 seconds. This interval is implementation dependent and can also be specified as a parameter when

[3] Ultrix is a trademark of Digital Equipment Corporation

the filesystem is mounted. The client will retransmit again (and again, ...) if no response is received. The time interval is increased using a backoff algorithm (next interval = current interval X 4) up to a ceiling value (60 seconds) after each successive timeout. This level of impatience that a client has for a server is determined solely by the client and is not dynamically adjusted based on past server performance.

The number of retries within a *timeout cycle* is implementation dependent and is also a mount time parameter. The reference port sets the default number of retries in a timeout cycle to three. With a *soft* mount (a mount option), if a response is not received for a request within the first timeout cycle, then the client operation (system call) fails. With a *hard* mount, the request is retransmitted until a response is received. With a hard mount, a single NFS request may span more than one timeout cycle before it receives a reply. The backoff algorithm described above continues to increase the timeout interval across timeout cycles.

The client RPC (*Remote Procedure Call* [4]) layer assigns a transaction ID (*xid*) to each outgoing request. Duplicate requests within the same timeout cycle will have the same *xid*. Duplicate requests will have different *xids* when the number of re-transmissions exceeds the number of re-tries in a timeout cycle. Version 2 of the NFS protocol does not define an NFS transaction ID that is unique to each NFS request from a given client. This makes it impossible for a server to reliably determine whether two requests are actually duplicates. [It is not enough to know that two requests appear to be the same (e.g. a write to the same location in the same file), since they could have been generated by two separate client processes in sequence.]

The client system can have multiple outstanding read and/or write requests. A client process blocks whenever a read or write request cannot be satisfied locally and must be processed by the server. When it blocks, another process can run; that process may also generate a read or write request. A single process can have multiple outstanding read and/or write requests if the client system is running NFS block I/O (*biod(8)*) daemons. These daemons perform client read-ahead and write-behind functions asynchronously, allowing the client process to continue execution. Each outstanding request will time out individually; each can result in a retransmission. Clients discard duplicate responses (the second response received for a single request) as unsolicited input, but they are counted as a *badxid*, and available via *nfsstat(8)*.

NFS server systems range in size from small machines with a single, slow disk to large multi-processor machines with disk farms.

A typical NFS server system simply waits for work to appear on an incoming request queue. This queue is the socket buffer allocated for the NFS socket. Incoming requests are converted into a form understandable by the local filesystem routines that actually perform the work of getting data to/from a disk. The incoming request queue is of fixed size. If the queue fills (requests coming in faster than they can be processed) then some incoming requests may be lost.

The amount of work that a server can perform is called server bandwidth. It is usually not limited by CPU speed, but by network interface and/or the disk subsystem performance. Server bandwidth is sometimes measured in a general manner, e.g. NFS operations/second, and sometimes specifically, e.g. read or write speed in Kbytes/second. A typical server does not prioritize incoming requests based on type of request or originating client.

Ignoring access rights and security, an NFS server has limited control over how it is used. An administrator decides:

- Whether to serve or not. A system serves by running *nfsd(8)* daemons.
- What to serve. A server only allows operations on *exported* filesystems.

- How many *nfsd* daemons to run. This controls the number of NFS requests that the server can work on concurrently. It also controls the amount of local resources (e.g. disk bandwidth) that is available for remote use (versus use by local processes).

The server depends upon its clients to attenuate their request loads as it becomes heavily loaded (i.e. the aggregate load is coming in faster than it can be processed). However, nothing makes a particular client implementation act kindly towards a server. There is no way to enforce the client backoff scheme described above. Most implementations allow a client administrator (or workstation user) to run as many block I/O daemons as they wish, and to mount with retransmission timeout values that are very small. When faced with one of the latest generation workstations armed with a suitable workload, the performance of even the most powerful server configurations can degrade drastically.

2.2. NFS Writes

Since an NFS server is bound by a stateless protocol, it must commit any modified data to stable storage before responding to the client that the request is complete [1]. If a server is not following this rule, then it is not living up to its part of the agreement implicit in the NFS crash recovery design. An asynchronous operation carries with it the promise to fully complete that operation at some later time. Without a way to recall past unkept promises, a server cannot make them. The protocol contains no provisions for recalling past promises (which is precisely why crash recovery is so easy). Therefore, a server must complete each data modifying operation fully (synchronously) before responding .

For each remote write request, at least one, and possibly two or three synchronous disk operations must be performed by the server before a response can be sent to the client indicating that the request has been completed. At the very least, the data block in question must be written. If the write increased the size of the file, or on-disk structures have changed (e.g. adding a direct block to fill a "hole" in the file), then the block containing the inode must be written. Finally, if an indirect block was modified, then it too must be written before responding.

The reference port makes a special case for the file modify time in the inode. If modify time is the only item changed in the inode as a result of a write operation, i.e. a write to a previously allocated block, then the inode update to disk is performed asynchronously. This is one promise that the server may not keep; the risk is taken for the benefits of better performance.

2.3. Duplicate Requests

Duplicate requests (sometimes called delayed retransmissions) are part of the NFS crash recovery design. A server can receive a duplicate request while performing the original request. Multiple copies of a request can be received and placed on the input queue before any are processed. If a client is preparing to retransmit when the response it wanted is received, the server will still be sent a duplicate request.

2.4. Non-Idempotent Operations

When used in a database context, the term *idempotent* is used to describe transactions that can be applied more than once without any ill effects. Inquiry transactions are *idempotent.* Debit and credit transactions are *non-idempotent.*

When used in an NFS context, the term can be used to distinguish between request types. An *idempotent* request (e.g. read) is one that a server can perform more than once without side effect. The side effects caused by performing a duplicate request can be classified as destructive and non-destructive.

To simplify the following discussion, we will only consider the case where the file in question is being accessed by a single remote client and not being shared between multiple remote clients and/or processes local to the server. Of the sixteen request types in version 2 of the NFS protocol (the only version in production use today), nine are *non-idempotent.* These are:

Table 1. Non-Idempotent NFS Operations

Operation Name	Description
create	create a file
remove	remove a file
link	create a link to a file
symlink	create a symbolic link
mkdir	make a directory
rmdir	remove a directory
rename	rename a file
setattr	set file attributes
write	write to a file

The first seven operations in Table 1 are obviously *non-idempotent.* They cannot be reprocessed without special attention simply because they may fail if tried a second time. The create request, for example, can be used to create a file for which the owner does not have write permission. A duplicate of this request cannot succeed if the original succeeded. Similarly, you can only successfully remove a file once; if permission was granted the first time, a second try cannot succeed. This type of scenario is not destructive, but is a nuisance for the server implementation to sort out.

Another scenario, one that does have destructive side effects, involves retransmitted *non-idempotent* requests and a race condition between *nfsd* daemons on the server. In Example 1, the client runs a process that creates a file and writes one block into it. The server is running two *nfsd* daemons. Client and server activities are shown at a number of points on a time line; the points are not equally spaced.

Example 1. Destructive Non-Idempotent Scenario

Time	Client Activity	Server Activity
t0	process starts	idle
t1	transmit create request c(0)	idle
t2	wait for create response	receive c(0), schedule *nfsd1*
t3	retransmit create request c(1)	*nfsd1*: complete c(0), truncate file, send create response
t4	receive create response process resumes	receive c(1), schedule *nfsd1*
t5	transmit write request, w(0)	*nfsd1*: starts but blocks on a system resource
t6	wait for write response	receives w(0), schedules *nfsd2*
t7	wait for write response	*nfsd2*: complete w(0) send write response
t8	receive write response process completes	*nfsd1*: complete c(0), truncate file, send create response
t9	receive create response and discard it	idle

In Example 1, the server processes two create requests and one write request. The net effect is a zero length file; the write has been lost. This problem has been seen at MIT [2].

There is a variant of the scenario in Example 1 that involves no *nfsd* block on a system resource (t5). If both *nfsd1* and *nfsd2* are scheduled, one with the write and the other

with the duplicate create, then scheduler vagaries can make their order of execution unpredictable.

The setattr and write operations are not as obvious as the first seven listed in Table 1. They are destructive only if re-applied after some other intervening operation. Setattr can be used to truncate a file; it can be used instead of create in a scenario similar to Example 1. Write would appear to be *idempotent*, but there is a curious destructive case here as well. If a duplicate write request is applied in a server *nfsd* race after a file truncation, then the file size is non-zero and the file contents are binary zeroes + the block of write data.

2.5. Reference Server Transaction Cache

The 4.3BSD based kernel implementation of NFS that is available from Sun Microsystems, Inc. features a cache of recently processed transaction IDs (*xids*) on the server. This cache is implemented in the kernel RPC layer, not in the NFS layer.

The server NFS layer uses this cache to store the *xids* of recent requests that have succeeded. The scope of the cache is five types of transactions: create, remove, link, mkdir, and rmdir. The cache is accessed by hashing the *xid* into an array of lists. The cache entries are re-used in a round-robin fashion. If an operation fails, the cache is used to help determine if the failure was due to the request being a duplicate of one that previously succeeded. If it is, then a positive response is returned to the client. Used in this manner, the cache helps to sort out the non-destructive type of behavior described above, but not the destructive behavior.

Destructive behavior is not prevented for two reasons:

- The cache is not used or consulted for the setattr or write operations.
- For the operations where the cache is used, it is searched only after an operation has already been performed, and only then if it has failed.

3. Wasted Server Effort

As mentioned earlier, we chose to look at servers under write load for areas where server performance could be improved. An NFS server implementation (for UNIX) is an order of magnitude simpler than the NFS client implementation. That made it a simpler place to look for a quick improvement. Also, a server can be used by many different client implementations; improving our client implementation didn't necessarily improve conditions for our servers.

NFS servers were placed under load and their activities were logged. The logging was done by modifying the server kernel to write information of interest to the existing Ultrix error log facility. An existing kernel interface for writing to this error log was used. The *nfsstat(8)* program was used on the client to measure the total number of write requests and the number of duplicate responses (*badxids*).

An examination of the server log showed that when the server's response slowed, the client would "ask again", as designed. This duplicate request, or retransmission, was added to the already heavy load on the server. Performing the operation a second time helped guarantee that it would be slow in responding to some other request, which might then result in a duplicate of that one, and so on.

Our *reference port* based NFS server implementation didn't distinguish the second request for work that had already been done from the first request and performed the request a second (or third!) time.

When the workload is "expensive" to perform (writes are the most expensive [3]), it is relatively easy to create this wasteful server scenario. One µVAX-II[4] client system running 4 *biods* can have up to 5 concurrent write requests from a single user process. A

[4] VAX and µVAX are trademarks of Digital Equipment Corporation.

simple test program was used that wrote 1 Mbyte to a file via 128 8 Kbyte write requests. The file was always 1 Mbyte in size when the program started; the program emptied the file when it was opened and proceeded to extend it to the 1 Mbyte final size. A write that extends a file is the most costly of write requests (see *NFS Writes* above).

In the tables that follow,

- 'small server' is a dedicated μVAX-II with an RD53 (70 Mbyte Micropolis 1325D Winchester) running 4 *nfsds*.
- 'large server' is a dedicated VAX 8550 with RA81 450 Mbyte disk running 4 *nfsds*.
- 'write requests' is the number of 8 Kbyte writes generated by the client.
- 'duplicate writes' is the number of retransmitted client write requests.
- 'excess writes transmitted' is a measure of the excess data sent from the client to the server.
- 'duplicate writes processed' is the number of duplicate writes performed by the server.
- 'excess writes processed' is a measure of wasted server bandwidth.
- 'write speed' is a client measure (in Kbytes/second) of throughput; it is 1 Mbyte divided by the time needed to open, truncate, write, and close the file.

When the test program is run, a minimum of 128 write requests must be made; if more are generated, then they are duplicates. Likewise, 128 write requests must be processed by the server; processing more is a waste of server bandwidth. The initial timeout interval used was .7 seconds. Twenty iterations of this test program were run and statistics gathered. Table 2 contains the averaged results.

Table 2. Write Load Test Results

	small server	large server
write requests	190	141
duplicate writes	62	13
excess writes transmitted	48%	10%
duplicate writes processed	61	13
excess writes processed	48%	10%
write speed (Kbytes/sec.)	35	75

These results showed that significant server bandwidth was spent performing duplicate writes. This relatively simple test immediately indicated an area upon which to focus. If the server could be made to spend less bandwidth on "useless" work when under write load, then it would have more bandwidth for "useful" work.

4. Server Modifications

The *reference port* RPC transaction cache described above was used as a starting point.

The scope of the cache was increased from: create, remove, link, mkdir, and rmdir to include all request types. More information was added to each cache entry: an in-progress flag, a transaction completion code, and a time stamp. The size of the cache was kept at the reference port size of 400 entries. The entries are re-used in the same round-robin manner as in the reference port. That is, entries age equally and can be referenced right up to the time of re-use.

The major change is in how the NFS layer uses the cache. The reference port performs an operation and checks the cache afterwards only if it fails and it is one of the types within the scope of the cache. This approach was changed in the following ways.

4.1. Requests In-Progress

A check of the cache on every incoming request was added as one of the very first NFS layer operations. If the transaction is in the cache, and marked in-progress, then the request is counted and quickly discarded without any response. If the transaction is not in the cache, it is added and marked in-progress.

The overhead associated with a fast cache check pales in comparison to the overhead associated with performing even the fastest request and transmitting a (useless) response. When under load, the check is saving much more than it costs. When not under load, the cost is negligible. As for throwing the request away: it is, after all, a duplicate and already getting attention; a response will be forthcoming.

4.2. Requests Recently Completed

If a duplicate request is received within a certain time interval (called *throwaway window*) and the original request was processed and successfully completed, then the request is counted as a duplicate and quickly discarded without a response. A duplicate of a request that earlier failed is re-tried. Experiments with various intervals showed that a *throwaway window* of from 3 to 6 seconds was sufficient to drop the processing of duplicate requests to nearly zero.

If a client asks a server to repeat an operation, then either the response was lost, or more likely, the duplicate request passed the response in transit, i.e. the client has actually gotten the original response by the time the server starts to process the duplicate. In the former case, the client will simply keep asking; after the *throwaway* interval has expired, the server will perform the request. The latter case is exactly what we are looking for; we truly want to throw this request away. It isn't clear whether it is better to retry requests that have previously failed or not. Failures don't happen often and are usually processed quickly. By re-processing, the old server behavior was kept for failure cases. Therefore, nothing was broken that was not previously broken. [There is a potential, but unlikely, problem here: the client may get a failure response from the original request and the server replays a duplicate which for some reason now succeeds. The client thinks the operation failed, when in fact it has succeeded. This can cause client/server cache consistency problems.]

4.3. Non-Idempotent Operations

Special treatment is given to the *non-idempotent* operations (including setattr and write). History on these operations is kept, namely completion status and a time stamp when the operation completed. When one of these operations is completed, its cache entry is updated with completion status and the appropriate inode time (either "modified" or "changed"). When a duplicate is received for a request that earlier succeeded, and the inode indicates that the file has not changed since that time by comparing its time with the time in the cache entry, then a successful response is returned to the client. This will occur until the cache entry is re-used and the cached information is lost. After that time, a duplicate request will be processed as a new one.

The goal was to avoid re-processing these duplicates as long as possible. Writes are very expensive and the *non-idempotent* operations can be dangerous. If the earlier request has succeeded and the file has not since changed, then it seems clear that to return a response is correct. What to do for earlier failures is less clear, but (like above) by re-processing them, old server behavior was maintained.

5. Results and Experiences

When the write tests described earlier were run using a modified server, the number of duplicate writes performed by the server dropped to zero (the server was using a *throwaway window* of six seconds). Since the server was doing less wasteful work, it responded faster to the necessary work and the write throughput as seen by the client increased. Since the amount of bandwidth wasted was greatest on the small server, that is where the improvement was greatest. Table 3 compares the earlier results (from Table 2) with the averaged results obtained using a modified server.

Table 3. A Comparison of Write Load Test Results

	small server	modified small server	change	large server	modified large server	change
write requests	190	190		141	141	
duplicate writes	62	62		13	13	
excess writes transmitted	48%	48%		10%	10%	
duplicate writes processed	61	0	-100%	13	0	-100%
excess writes processed	48%	0%	-100%	10%	0%	-100%
write speed (Kbytes/sec.)	35	48	+37%	75	83	+11%

The real gains are not necessarily seen from a single client, but a group of clients that take advantage of the server bandwidth increase and better response to load conditions.

A very beneficial side effect is that there have been no reports of the destructive behavior caused by the re-processing of *non-idempotent* operations. These operations, as a whole, are expensive to perform, but their frequency (except for write) is relatively low. The major reason for handling them specially is their destructive potential.

The server modifications described here were integrated into Ultrix V2.2, which has been in production use since early 1988.

In Ultrix V3.0 the client RPC layer was modified so that the caller (NFS) could define the transaction ID (*xid*). In this way, the NFS layer can control the *xid* and make sure that it is unique for each request, even across timeout cycles. This in turn helps make the server transaction cache even more effective. More work needs to be done comparing the effectiveness of different cache sizes over varying workloads.

6. Conclusions

NFS servers were placed under heavy write load and the results (Table 2) showed that the system was unstable in response to this load. A heavy write request load resulted in retransmitted write requests that resulted in a heavier load. The server modifications using the *work avoidance* technique described earlier reduced the positive feedback effects of duplicate client requests. These results are displayed in Table 3. The modified server responds to heavy load by simply trying to throw some requests away. This notion didn't set well at first. But the problem under observation was a slow server looking like a downed one to an impatient client; it seemed ironic (and amusing) that it might help if the slow server started selectively acting like a downed one and discarding requests. Throwing a duplicate request away opens up server bandwidth for other work; not responding reduces the amount of network traffic. The NFS client retry model will take care of server responses that are lost on the network or by the client. NFS is usually used on LANs, where network losses are infrequent. Therefore, delays due to this scheme should be rare.

An implementation of this technique resulted in a significant increase in server bandwidth, especially with low end server configurations. A very beneficial side effect is that the destructive re-application of duplicate *non-idempotent* operations is made much less likely. There is no need to re-build the previous contents of the cache after a server

crash. Therefore, the use of this cache violates neither the statelessness of the NFS protocol nor its crash recovery design.

The load tests showed that the server's input queue worked very well (at least with this selective type of load). When under heavy write load, very few requests were lost (seen by comparing the number of duplicate requests generated by the client with those processed by the server). The limiting factor was not an input queue that was too small. An analogy can be drawn here between NFS server congestion and network congestion. John Nagle [5] has shown that long (even infinite) queues won't help a network gateway with an overload; the load must be shed in some way. The modified server screens its load and sheds requests that it is currently processing or has recently completed.

The mechanisms leading to congestion in datagram packet-switching networks are similar to those that lead to congestion in stateless (non flow-controlled) RPC based file servers. Van Jacobson [6] has shown that an effective answer to network congestion is for the client to detect it and back off on its request rate. Raj Jain [7] stresses the need to operate a system below the point of unstable positive feedback. An area that begs for future NFS work is the client side of the feedback loop. The NFS *reference port* client implementation does back off when retransmitting a given request more than once, but it does not use past server response characteristics to determine future initial timeout intervals.

7. Recommendations

This technique is recommended for use by all NFS server implementations. An NFS server selects neither the implementation of its client, nor its timeout characteristics. Even if NFS client timeout behavior is modified in some future implementations, current ones (using version 2 of the NFS protocol) will be used for years to come; performance gains are obtainable with them by using this technique. Likewise, the correctness improvements seen by this technique are useful for current client implementations. Version 3 of the protocol won't be defined until at least 1989; it may fix some of the *non-idempotent* operation problems, but won't be widely used until years after that.

NFS servers that use this technique can offer better performance and correctness to their clients. It is recommended that client implementations change their RPC layer to assign unique transaction IDs (*xids*) to NFS requests. This improves the effectiveness of the technique. It is further recommended that Version 3 of the NFS protocol include a transaction ID in each request.

8. Acknowledgements

The diskless performance work done by Charlie Briggs [3] was the takeoff point for this work. I further relied on Charlie's judgment, patience, and encouragement while writing this paper. John Dustin, Fred Glover, Joe Martin, Bob Rodriguez, Ursula Sinkewicz, and Mary Walker were draft reviewers; they offered many comments that greatly improved this paper's readability. Jeff Mogul made the insightful analogy between stateless file server congestion and datagram packet-switching network congestion [5][6][7].

9. References

[1] Russel Sandberg, et. al., "Design and Implementation of the Sun Network Filesystem", *USENIX Summer Conference Proceedings,* pp. 119-130, June 1985. Where it all started.

[2] Described by Dan Geer, MIT Project Athena, Nationwide File System Workshop, Carnegie Mellon University, Pittsburgh, PA, August 23-24, 1988.

[3] Charles Briggs, "NFS Diskless Workstation Performance", Digital Equipment Corporation Technical Report, February 24, 1988.

[4] Bob Lyon, "Sun Remote Procedure Call Specification", Sun Microsystems, Inc. Technical Report, 1984

[5] John Nagle, "On Packet Switches with Infinite Storage", in *IEEE Topics in Communications,* pp. 435-438, April 1987.

[6] Van Jacobson, "Congestion Avoidance and Control", in *Proceedings of SIGCOMM88,* pp. 314-329, Stanford, CA, August 1988.

[7] K. K. Ramakrishnan and Raj Jain, "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer", in *Proceedings of SIGCOMM88,* pp. 303-313, Stanford, CA, August 1988.