A Highly Available Lock Manager For HA-NFS

Anupam Bhide – IBM T.J. Watson Research Center
Spencer Shepler – IBM Austin

ABSTRACT

This paper presents the design and implementation of a highly available lock manager for highly available NFS (HA-NFS). HA-NFS provides highly available network file service to NFS clients and can be used by any NFS client without modification. This is provided by having two servers share dual-ported disks so that one server can take over the other's disks and file systems if it fails. Making the NFS service highly available is not enough since many applications that use NFS also use other services provided with NFS such as the network lock manager. We describe a scheme whereby each server transfers enough of its lock state to the other so that if it fails, the other server can go through a lock recovery protocol. Our design goal was to make the overhead of transferring the state during failure-free operation as low as possible.

1. Introduction

This paper presents the design and implementation of a highly available lock manager for a Highly Available Network File Server (HA-NFS) [3]. HA-NFS provides tolerance to file server, disk and network failures and can be used by any NFS client. Recovery from server failure is provided by having two servers share access to dual-ported disks and provide backup service for each other. These servers are therefore referred to as twins of each other. However, it is not enough to recover the file server state at a backup server in case of a crash. Most NFS implementations are accompanied by a network lock manager so that clients can obtain locks for files that are remotely mounted. NFS file locking is an extension of local file locking and was designed so that applications can use file locking without having to know whether the file is local or remote. Most NFS implementations support a *lockf()/fcntl()*, System V[1] style of advisory file and record locking over the network. A number of applications use the network lock manager to synchronize access to shared files and to prevent multiple processes from modifying the same file at the same time. Since locking is inherently stateful and NFS is supposed to be stateless, the lock manager is implemented separately from NFS.

When the primary server fails, the lock state must be recovered at the backup server. This paper will describe a design for recovering the locking state at the backup in case of server failure and for enabling a failed server which is recovering and re-integrating to regain its locking state.

We have implemented a prototype of the Highly Available lock manager for HA-NFS on the same platform on which HA-NFS was implemented: a network of workstations and two file servers from the IBM RISC System/6000 family of computing systems running the AIX Version 3 (AIXv3) operating system, and connected by either a 10 Mbit/s Ethernet network or a 4 Mbit/s or a 16 Mbit/s token ring network. We constructed dual-ported disks from off-the-shelf SCSI disks attached to a SCSI bus that is shared by the two servers. The prototype is operational and has satisfied the design goals.

In section 2, we present background information on HA-NFS. In section 3, we describe the NFS locking protocol. In section 4, we describe various design alternatives to enable lock state to survive processor failure and recovery events. Section 5 describes the design we chose and why. Section 6 describes the re-integration protocol executed when a failed server recovers. Section 7 presents an evaluation of the design from the point of view of implementation effort and performance. Section 8 describes the technique used to recover from media and network failures. Section 9 compares our approach with other approaches and section 10 proposes some items for future work.

2. The Highly Available Network File System (HA-NFS)

Traditional approaches for providing reliability in networked file systems use server replication. HA-NFS differs from traditional approaches in that it tolerates server failures by using dual-ported disks that are accessible to two servers, each acting as a backup for the other and hence are called twins of each other. The disks are divided into two sets, each served by one server during normal operation. Each server maintains on *its* disks enough information to reconstruct its current volatile state. Since NFS is an almost stateless protocol, the only volatile information is the duplicate cache information that is needed to detect duplicate transmissions. For example, a "create new file" remote procedure call (RPC) may reach a server and the file create operation may take place, but the acknowledgement to the client could be lost. The client would re-try the RPC and may receive an error because the file already exists as a result of the previous RPC unless the RPC was flagged as a re-try. To detect this, the server stores a cache of recently executed RPCs called the "duplicate cache". For further discussion of this topic see [4].

The two servers periodically exchange liveness-checking messages. If one server fails, the failed server's disks will be taken over by its twin server. The twin then reconstructs the lost volatile duplicate cache state using the information on disk. Then the twin *impersonates* the failed server by taking over its IP address and operation continues with a potential reduction in performance due to the increased load. The clients on the network are oblivious to the failure and continue to access the file system using the same address. During normal operation, the servers communicate only for periodic liveness-checking. The servers do not maintain any information about each other's volatile state or attempt to access each other's disks during normal (failure-free) mode of operation. HA-NFS adheres to the NFS protocol standard and can be used by existing NFS clients without modification.

HA-NFS is implemented on top of the AIXv3 log-based file system. The AIXv3 file system provides serializable and atomic modification of file system meta-data by using transactional locking and logging techniques. File system meta-data are composed of directories, inodes, and indirect blocks. Every AIXv3 system call that modifies the meta-data does so as a transaction, locking meta-data as they are referenced, and recording the changes in a disk log before allowing the meta-data to be written to their "home" locations on disk. In the case of system failure, the meta-data are restored to a consistent state by applying the changes contained in the log. The reliability of ordinary files is ensured by NFS semantics, which require forcing the file data to disk before sending an acknowledgement to the client. The volatile state at an NFS server consists of the duplicate cache; this information is recorded on the disk log so that it can be recovered by the backup server. Further details about the design and the implementation of HA-NFS can be found in [3].

3. The NFS Locking Protocol

In this section, we will provide and overview of the NFS/ONC locking protocol. The locking protocol is implemented outside of the NFS protocol, because the NFS locking protocol is stateful and NFS is designed to be stateless. In most implementations the file locking protocol is actually implemented in two daemons. The daemons are usually named rpc.lockd and rpc.statd and these are the names used in AIXv3. The rpc.lockd daemon at a server handles locking requests for NFS clients which are accessing files at the server. The rpc.lockd daemon acts as a surrogate at the server for client processes and keeps track of what locks are held by clients at any one point in time. At a client, the rpc.lockd keeps track of what locks are held at the NFS server by the various processes.

The second daemon, the rpc.statd daemon at a server keeps a list of clients that are to be tracked for system failure. Similarly at a client, this daemon keeps track of what remote servers are currently being accessed by file lock requests.

Getting A lock

When an application at a client makes a system call requesting a lock on a NFS-mounted file, the client kernel makes a RPC to the client's rpc.lockd. This rpc.lockd then sends the lock request to the rpc.lockd at the server which makes a lock request to the server's kernel. The server's kernel accepts the lock request and returns the appropriate response to the server's rpc.lockd. The server's rpc.lockd will then respond to the client's rpc.lockd with the result. It in turn will respond to the client's kernel which will then return the response to the application.

When the client's rpc.lockd receives the original lock request from the client kernel, it will register the server's host name with the client's rpc.statd. This is done before the lock request is sent to the server's rpc.lockd to be processed. Upon receiving the lock request from the client the rpc.lockd at the server will register the host name of the client with the rpc.statd at the server. The rpc.statds on both the client and server record the host names on disk so that it can be accessed after a failure. This registration process is done for the first lock request only. The rpc.lockd keeps internal state about which hosts it has registered with the rpc.statd so this initial registration step is skipped on subsequent lock requests.

Recovery Actions Upon Client/Server Failure

In a standard NFS (not HA) server configuration, there is a method to rebuild the locking state that is kept by the NFS server. Rebuilding of state occurs only after server failures. It is not needed after client failure and recovery since the applications that took the locks no longer exist after the client's system failure. The only actions that need to be taken when a failed client recovers is to tell relevant servers to release the locks that are held on the client's behalf.

The following explains the corresponding steps that the NFS server rpc.lockd/rpc.statd follow to recover locking state. The rpc.statd is started before the rpc.lockd during system initialization. When the rpc.statd on a server restarts, assuming system failure, it reads from disk the names of systems it was monitoring during its previous incarnation. The rpc.statd then informs each of the rpc.statds on these client systems about the server's failure. Client rpc.statds then inform their rpc.lockds about a server failure.

In the case where the rpc.lockd process on the server has failed, the rpc.lockd process during initialization will inform the local rpc.statd that it had failed and goes into a grace period in which it accepts only lock reclaim requests from clients. When a client rpc.lockd is informed of server failure, it goes through its lock table and re-requests or reclaims all locks it had held at that server. After all clients go through this protocol, the server has now regained the lock state that was held before the failure.

In the case where the rpc.lockd process on the server has failed, the rpc.lockd process during initialization will inform the local rpc.statd that it had failed and goes into a grace period in which it accepts only lock reclaim requests from clients. When a client rpc.lockd is informed of server failure, it goes through its lock table and re-requests or reclaims all locks it had held at that server. After all clients go through this protocol, the server has now regained the lock state that was held before the failure. If the client system fails, the rpc.statd at the client will notify the servers of client failure. The list of servers is built from the list of monitored servers that the rpc.statd was keeping in the file system of the client. Upon notification of client system failure, the server's rpc.lockd will release all of the file locks that were held by that client.

4. Design Alternatives

To design a highly available lock manager for HA-NFS, a way must be found to transfer the locking state held by the primary HA-NFS server to the backup or twin HA-NFS server. This needs to be done so that correctness can be maintained in the operation of the NFS server from the client's perspective.

The first approach that might be taken is to follow the same general scheme for transferring the lock state that the HA-NFS server uses to transfer file system state and duplicate cache entries to the twin HA-NFS server. Recall that duplicate cache entries are used to detect request re-transmissions that occur when acknowledgements get lost. The HA-NFS server stores the duplicate cache entry in the file system log when the duplicate entry is initially added to the duplicate cache table. This works because of the one to one mapping of the duplicate cache entry and the commit of the entry to the AIX Journaled File System (JFS) log. For this same mechanism to work for the NFS locking there needs to be a mapping between the lock/unlock operations of the client and JFS logging commit points which correspond to metadata modification points. This mapping does not exist and would not be possible without a redesign of the logging services to serve other than normal JFS activity. This would also mean that locking operations would run at disk speed.

The second approach could be to have the rpc.lockd of each HA-NFS server transfer the locking state to the twin HA-NFS server. This would need to be done with each positive response to a client's locking request. Before the primary server's rpc.lockd sends its positive/granted response to the client, it would have to call the rpc.lockd on the twin HA-NFS server. The rpc.lockd on the twin could then build the same locking state as the primary server. This approach would have a performance impact on each locking operation. This would also affect the twin's locking performance and general system performance since it would be fielding the same locking requests that the primary would be handling. Another drawback to this approach would be the added complexity of keeping locking state for the twin and differentiating that locking state from the local locking state of the twin.

The third approach is similar to the second, except that instead of all positive lock/unlock responses being passed to the twin, host names of new clients are passed to the twin on the client's first lock request. Thus, the rpc.statd would be the one passing state information to the twin's rpc.statd. Recall that the rpc.statd is contacted by the rpc.lockd on the first lock request of a client. The rpc.statd is told to monitor that client. The rpc.statd in turn creates a file in the directory /etc/sm. This file name matches the host name of the client making the request. With this procedure the rpc.statd can then recover the list of clients that were being monitored before failure. After the rpc.statd has placed the file named after the client in the /etc/sm directory, it will contact the twin's rpc.statd. The twin's rpc.statd also places an entry in the /etc/sm directory that corresponds to the primary server's entry.

With the corresponding /etc/sm entries in place on the HA-NFS twin, all it has to do during takeover is to go through the lock recovery protocol playing the role of a recovering server for both itself and its twin. The clients of the failed server and those of the twin will execute lock recovery and the twin will effectively rebuild the locking state held by the primary server.

5. Our Design

The third design alternative involves the least overhead during normal failure-free operation. Tables 1 and 2 show the details of this protocol. Table 1 shows how a lock is obtained. Table 2 explains how locking state is re-established at the backup after a server fails. This design requires that the rpc.lockd be stopped and restarted during the takeover process to force the lock recovery to occur. One disadvantage of this scheme is that locking state has to be rebuilt instead of being already available as in the second scheme or getting it from the log as in the first scheme. We considered this trade-off acceptable since we were getting better performance in the normal case for sacrificing some performance during recovery from failure.

For simplicity in the chosen design, we decided to let the rpc.lockd reclaim lock state for itself in addition to the failed server. It would be possible to modify the rpc.lockd so that it would not drop its own locking state during recovery of the twin HA-NFS server's locking state.

- An application requests a lock on a file that resides in an NFS file system.
- The NFS client's kernel makes a RPC to the client's rpc.lockd requesting the lock
- If this is the first lock request for the server, the rpc.lockd on the client registers the server's host name with the rpc.statd on the client.
- The client's rpc.lockd sends the lock request to the server's rpc.lockd.
- If the lock request is the first one received from this particular client, the rpc.lockd registers the client's host name with the rpc.statd on the server.
- The server's rpc.statd then informs its twin rpc.statd on the backup server that the client should be monitored and then sends an acknowledgement to the rpc.lockd.
- The server's rpc.lockd makes the lock request to the server's kernel.
- The server's kernel accepts the lock request and validates it. Returns response to the server's rpc.lockd.
- The server's rpc.lockd responds to the client's rpc.lockd.
- The rpc.lockd on the client responds to the NFS client's kernel with the lock response.
- The application is given the answer to its lock request.

Table 1: Getting a lock on remote file in HA-NFS

The rpc.statd on a system is informed of the name of its twin host through a RPC call by a HA-NFS daemon when the HA-NFS subsystem is started. Once this is done, the rpc.statd will contact the twin's rpc.statd every time it is called by the local rpc.lockd with a new host to be monitored. The twin's rpc.statd will create an entry in the /etc/sm directory with the host name specified by the primary's rpc.statd and respond to the monitoring request.

If a server fails, the HA-NFS daemons running on its twin will detect the failure and take over its disks. They will then replay the log and bring the file systems to a consistent state and mount them on the appropriate directory. Finally, they will take over the IP address of the failed server on a spare network interface provided for this purpose. The network interface may be either ethernet or token ring. Finally the rpc.lockd is restarted. Because of the restart, the rpc.lockd and rpc.statd will go through the lock recovery protocol sending out

- The failure of the twin server is detected by the backup HA-NFS server. The backup server takes over the disks, brings the file systems to a consistent state and rebuilds the duplicate cache of the failed server. The rpc.lockd is stopped to prevent requests from being processed until takeover is complete. The backup then takes over the IP address of the failed server and starts to provide NFS file service.
- The rpc.lockd is restarted at the backup server. When it starts, it contacts the server's rpc.statd to tell it of its failure.
- Upon receiving the failure notification from the rpc.lockd, the rpc.statd at the backup server contacts each of the clients (both its own clients as well as those of the failed twin) that were being monitored because of the current locking state. This notification lets each of the clients know that the server's rpc.lockd has failed. The rpc.statd notifies each of the clients playing the role of the appropriate server.
- After the rpc.lockd notifies the rpc.statd of its failure it goes into a grace period for lock recovery. This allows clients to reclaim locks that they held before the failure of the twin servers rpc.lockd. The default grace period is 45 seconds.
- When the client's rpc.statd receives the notification that the server has failed, it "calls back" to the rpc.lockd on the client to notify it of the failure of the server.
- When notified, the client's rpc.lockd will send reclaim requests for all locks currently being held that are from the failed server. These reclaim requests will be honored at the server. As a result the server will regain the locking state that was held prior to its failure.
- During the grace period on the server, the rpc.lockd will only honor *reclaim* requests from clients. This assures consistency for the clients that held locks prior to the failure. The locks will be held again when the server restarts normal lock service. Regular locking requests that the server's rpc.lockd receives will be returned to the requesting client with a message that the client should retry the lock request. After the grace period elapses, new lock requests start being honored.

Table 2: Lock Recovery After Server Failure in HA-NFS

requests to clients of both this server and its failed twin to perform lock re-claim actions. The clients will see a simultaneous failure of both the twin servers and send out reclaim requests to both. All these requests will be received by the operational twin (since it is responding to both IP addresses) and the correct locking state will be recovered.

Another detail of the rpc.statd modifications deals with contacting the clients upon server failure. The clients are notified of the server failure by receipt of an RPC. Within the parameters of the RPC is the host name of the server that has failed. The rpc.statd at the client uses that host name to check if it among the list that is currently being monitored. If so, the rpc.statd then sends an RPC to the rpc.lockd at the client signifying the server failure. A simple solution would be for the twin rpc.lockd upon takeover to contact each client with the name of the failed server as well as its own host name. To get around this overhead, the file with the client's name in the /etc/sm directory indicates whether it is this server that needs to monitor the client or its twin.

A call to unmonitor a host needs to be supported for the twin-registration process. This is needed so that when a primary's rpc.lockd decides it no longer needs to monitor a client, the rpc.statd on both the primary and twin systems will remove the monitoring entry from the /etc/sm directory. When the rpc.statd on the primary HA-NFS server receives an unmonitor request from the rpc.lockd it will remove its /etc/sm entry for that host. The rpc.statd will then call the twin's rpc.statd with the request to remove the host from its monitoring tables. The rpc.statd on the twin will decrement the reference count of that monitored host. If the reference count is zero, it will remove its entry from the /etc/sm directory.

6. Reintegration

When a failed server recovers, the HA-NFS daemon at that server will recover the duplicate cache state from its twin after taking over the file systems.

The failed server will also receive the rpc.statd state from the twin. The mechanism to handle this is achieved through the twin registration process at the twin. After the duplicate cache state is transferred to the recovering server, the host name of the recovering server is registered with the twin's rpc.statd. By design the rpc.statd will transfer to the registered server the full list of host names that it is currently monitoring. This allows the recovering server to obtain the current list of clients that are being monitored.

After receiving the list of clients, the recovering server then restarts the rpc.lockd. Through its normal recovery process each of the clients in the transferred monitoring list are contacted and told of the failure of the recovering server. The rpc.statd at the notified client will then follow the normal lock recovery process by contacting the client's rpc.lockd allowing it to reclaim the client's locks.

The twin at this point will restart its rpc.lockd and it will also have the locking state rebuilt when the client's reclaim their locks. The rpc.lockd on the twin was originally stopped so that the file systems of the recovering system could be unmounted. When the rpc.lockd exits gracefully, the locks that it holds are released from the file system therefore freeing the file systems of reference counts that may prevent them from being unmounted.

7. Evaluation

The effort it took to implement the design chosen was minimal. A simple RPC program was designed to handle the twin registration and the transfer of host monitoring requests between rpc.statd's. The rpc.statd code was structured in such a way that the extra logic required to implement our design was small. Most of our effort was spent on understanding the design and implementation of the rpc.lockd and rpc.statd prior to modification. After the design of these two daemons was understood it was straightforward to design and implement the method chosen. There are three areas where the performance of a HA-NFS server and its clients will be affected by our design for the highly available lock manager. This performance penalty is in comparison to a standard NFS server and the standard implementation of the network lock manager protocol.

1. The first performance penalty is taken when a given client makes the very first lock request of the HA-NFS server. Contacting the twin server for the monitoring of the client will add extra delay in responding to the client. No penalty is incurred for subsequent lock requests.
2. The second penalty will be paid when a twin fails and the twin that takes over its identity starts to process the lock requests of its own clients and the clients of the failed twin.
3. The third penalty is incurred when the reintegration of the failed twin occurs. The twin server that has taken over must transfer the monitoring state to the recovering twin. In this case the implementation has the rpc.statd forking a child that handles the transfer of the monitoring state.

Because the first two penalties are more important, they will be discussed in further detail.

Penalty For The First Lock Request

In both the standard NFS and our lock manager designs, the first time a client makes a lock request, the rpc.lockd contacts the rpc.statd, and asks that the client's name be placed in the /etc/sm directory.

This is done by creating a file name which corresponds to the host name of the client. In our design, the rpc.statd at this point of first registration will also contact the twin's rpc.statd and have it monitor the same client. The rpc.statd makes this RPC to the twin's rpc.statd and waits for a response before responding to its rpc.lockd monitoring request. This means that the rpc.lockd will not be able to continue processing the lock request of the client until the rpc.statd at the twin finishes creating its file that corresponds to the initiating twin.

This extra overhead of contacting the twin's rpc.statd will obviously delay the response to the first lock request of the client. Table 3 illustrates the impact of this delay. A configuration of Risc Systems/6000s were used to measure what the cost of this first lock request would be in a HA-NFS environment. Three systems were used for these measurements. They were running AIXv3 with the HA-NFS subsystem installed and configured. The rpc.lockd and rpc.statd had been modified to follow our design. The three systems were isolated on a 16 Mbit/s token ring. The test case that was executed reset the systems so that no previous rpc.statd state was held at any system (client or server). The test case at the client mounted one of the HA-NFS exported file systems from the HA-NFS server pair. The test case then noted the current time and issued a system call to obtain a lock on a file in the NFS mounted directory. After the lock system call returned, the current time again was noted and the elapsed time measured. This is reported on the row labelled "First lock". The test case went on to do the same sequence of lock and unlock requests a second time. This second iteration however did not reset the rpc.statd state. Therefore the overhead of obtaining a second lock was measured. This is reported on the row labelled "Second lock".

This test case was also executed with a standard NFS server for comparison. The same configuration described above was used except that it was just one standard NFS server and one client. This time the rpc.lockd and rpc.statd were running the standard algorithm. Again the rpc.statd state on both client and server was removed and the test case executed. The second lock request was also executed as in the scenario described above.

Both of these lock test scenarios were executed 50 times and an average response time for the lock request and standard deviation calculated. These are the results reported in Table 3.

The overhead of contacting the twin server to have the rpc.statd monitor the client almost doubles the response time for the very first lock request. We felt that this cost was reasonable given that it occured only for the first lock request from a particular client. The numbers for the "second lock" request with and without HA-NFS are the same within the limits of experimental error.

The same was done for the unlock request and again the elapsed time reported. Table 4 shows the measurements for the unlock requests. This data shows that HA-NFS unlock requests do not suffer from any extra overhead.

	Without HA-NFS		With HA-NFS	
	Lock Oprn	Std. Dev.	Lock Oprn	Std. Dev.
First Lock	132.12 ms	13.28 ms	245.06 ms	70.18 ms
Second Lock	15.66 ms	0.98 ms	16.08 ms	1.20 ms

Table 3: Highly Available Lock Manager Overheads for locks

	Without HA-NFS		With HA-NFS	
	Unlock Oprn	Std. Dev.	Unlock Oprn	Std. Dev.
First Unlock	14.30 ms	0.41 ms	14.42 ms	0.80 ms
Second Unlock	14.43 ms	0.47 ms	14.63 ms	0.60 ms

Table 4: Highly Available Lock Manager Overheads for unlocks

	Without HA-NFS		With HA-NFS	
	Lock Oprn	Unlock Oprn	Lock Oprn	Unlock Oprn
First Run	14.80 ms	13.89 ms	15.19 ms	13.97 ms
Second Run	14.82 ms	13.85 ms	14.91 ms	13.90 ms
Third Run	14.80 ms	13.89 ms	15.17 ms	13.98 ms

Table 5: Highly Available Lock Manager Overheads (500 operations)

We believe that the typical mode that clients use servers in most applications is that a client would tend to get many locks on a particular server in a given period. This would tend to wash out the effect of the higher HA-NFS first lock overhead as compared to the standard NFS lock manager. Table 5 shows a test case that executed 500 sequential lock operations in the same configurations specified above (unlock operations were also measured). The results reported are the per request elapsed time.

Overhead Of Handling The Failure Of A Twin

The steps that are taken when a server fails and its twin takes over operation for the failed server have been enumerated in Table 2. Remember that one of the steps is to stop the rpc.lockd while the takeover configuration is executing on the twin. Once the takeover is complete, the rpc.lockd is restarted. At this point, the rpc.lockd will notify the rpc.statd of its failure and the rpc.statd will execute its lock recovery algorithm.

With our design, the rpc.statd will have to contact each of the clients twice. One RPC will contain the twin's host name and the second RPC will contain the host name of the failed server. In this way the clients will be notified of both server's failure and they will reclaim the locks held at the server pair. With our design the rpc.statd has exactly twice the normal number of RPC's to execute. This number may also include clients that held locks at the failed server and not at the twin that has taken over. Since this notification mechanism is asynchronous to the other operations occurring at the server it should not be a significant burden for the twin.

Also, the twin will have to handle its own incoming reclaim requests and the reclaim requests for the failed server. This load is difficult to determine since it depends on the type of applications that are being executed at the clients and their locking behavior. Therefore under heavy stress the default grace period that the rpc.lockd uses for lock reclaims may not be sufficient for correct operation. This is also true of a standard NFS server but is made worse by the fact that the twin will also be handling the failed server's lock requests.

In the testing that has been done with this implementation the recovery of client locking state was achieved in a reasonable amount of time. The majority of this testing was done under light to medium locking stress. Since the rpc.lockd has a relatively short default time for its grace period, the recovery process that the clients are forced to go through may fail under a very heavy load. If this happens, the grace period can be increased by the system administrator to handle this case.

It should be mentioned that the lock response time will not be the only thing that will suffer when a server fails. The normal NFS requests that the twin receives will also be affected by the extra work load that it has taken on as part of the impersonation of the failed server.

8. Network and Media Failures

HA-NFS provides recovery from disk and network failures for the file server as described in [3]. The same methods can ensure that the lock manager can recover from these failures.

Fast recovery from disk failures is achieved in HA-NFS by mirroring files on different disks. However, all copies of the same file are on disks that are controlled by the same file server, eliminating the overhead of ensuring consistency and coherence between the two servers that would otherwise occur. Since disk failures are not frequent, mirroring is only used for applications that require continuous availability. Otherwise, archival backups could be used to recover from disk failures. The files used by the rpc.statd could be mirrored to provide high availability.

Network failures are tolerated by optional replication of the network components, including the transmission medium. However, packets are not replicated over the two networks. Instead, the network load is distributed over the networks. Clients detect network failure because of loss of heartbeat from servers and switch over to the second network by changing their routing tables. Also, a message is sent to the server to change its routing tables. This mechanism works for all messages between client and server, including the locking protocol messages.

9. Comparison with Other Systems

Tandem's NonStop architecture [2] [5] uses special-purpose hardware in the form of dual-ported disk controllers which allow each disk to be attached to two processors. If a single processor fails, the other takes over the disks and provides processes that were using these disks with continued access. However, Tandem has the concept of process-pairs. Thus each I/O process has a twin to which it continuously checkpoints its state. This ensures that the backup I/O process knows what operations are needed to bring the disk to a consistent state when it takes over. On the other hand, HA-NFS has no such checkpointing overhead during normal (failure-free) operation. The information for bringing the disks to a consistent state is stored on the disk itself by treating each NFS client-to-server RPC as a transaction and writing a log. Thus, there is a significant difference between the HA-NFS and Tandem approaches. Presumably in the Tandem approach the same processor-pair/checkpointing approach is used to transfer locking state from one process to its backup twin. In our approach, the rpc.statd communicates with its twin rpc.statd only when a new client makes its first lock request. No communication is required for subsequent lock requests.

VAXcluster [6] also has a distributed lock manager which recovers locks after a processor fails. Upon being notified of node failure, the lock manager on each node must perform recovery actions before normal cluster operation continues. First, each lock manager deallocates all locks acquired on behalf of other processors. Only local locks are retained. Next, each lock manager acquires each lock it had before the failure. The net result is to deallocate all locks owned by the failed node. However, note that this requires *all* locks to be re-acquired on any failure. In NFS and HA-NFS, clients that held locks at a failed server node need to re-acquire only those locks that were held at the failed node. The trade-off is that the first lock request is slower in HA-NFS.

10. Future Work

We used a simple design where the twin of a failed server rebuilds both its own locking state along with the locking state of the failed server. The load of this server would decrease if only the failed server's locking state were to be selectively rebuilt. This can be done by having the rpc.statd keep a more detailed record of what clients were monitored by which server. This way only the clients affected by a takeover would be notified of the server failure.

The other part of the design that might be extended has to deal with the grace period that the rpc.lockd uses. The grace period is used to allow the clients to rebuild their locking state after a server failure. This grace period in the implementation is a default of 45 seconds. As mentioned earlier, this default seems to work well with the test cases used but it may not be sufficient when the load of the server increases. There is a possibility that the grace period could be dynamically decided based on the number of clients that respond to the failure message that the rpc.statd supplies. The rpc.statd could possibly keep track of the percentage of clients that have contacted the server after being notified of the failure. Once a certain percentage has been reached the rpc.statd could then notify the local rpc.lockd so that it can make a decision to either continue the grace period or extend it. It is possible that clients do not need to contact the server after failure of the rpc.lockd so the percentage to use in determining validity of the grace period would not be simple.

These future work items could possibly decrease the work load of the server and increase the likelihood of correct and the timely reclamation of locking state.

Bibliography

[1] AT&T System V Interface Definition.

[2] Joel Bartlett, A NonStop Kernel, In *Proceedings of the Eighth Symposium on Operating Systems Principles, Vol 15 No 5, Dec 1981.*

[3] Anupam Bhide, Elmootaz Elnozahy and Stephen Morgan, A Highly Available Network File Server. In *Proceedings of the Winter 1991 USENIX Conference*

[4] C. Juszczak, Improving the Performance and Correctness of an NFS Server. In *Proceedings of the 1988 USENIX Conference.*

[5] J. Katzman, A Fault-Tolerant Computing System. In *Proceedings of the Eleventh Hawaii International Conference on System Sciences, Jan 1978.*

[6] N. Kronenberg, H. Levy and W. Strecker, VAXclusters: A Closely-Coupled System. In *ACM Transactions on Computer Systems, Vol. 4, No. 2, May 1986.*

Author Information

Anupam Bhide is a research staff member at the IBM T. J. Watson Research Center. His current research interests include database systems, operating systems, fault-tolerance and racquetball. In addition to his work on fault-tolerance in network file systems, he has worked on fault-tolerance in parallel database machine, and high performance transaction processing. He graduated with a Ph.D. from the University of California-Berkeley in 1988. Previously, he has a B. Tech from I.I.T.-Bombay and a M.S. from University of Wisconsin-Madison. He can be reached at anupam@watson.ibm.com.

Spencer Shepler is currently a software engineer for IBM in Austin Texas. His interests include operating systems and distributed systems specifically distributed file systems. Spencer graduated in 1989 from Purdue University with a Master of Science degree in Computer Science. Since that time, he has been working with and partly responsible for NFS and its implementation in the AIX V3 operating system. Reach him electronically at shepler@netmail.austin.ibm.com.