A Portable Lightweight Approach to NFS Replication

Raquel Menezes* Carlos Baquero† Francisco Moura‡

Universidade do Minho/INESC,
Departamento de Informática,
Campus de Gualtar,
4700 Braga,
PORTUGAL

{mesram,mescbm,fsm}@di.uminho.pt

Abstract

Under normal circumstances, NFS provides transparent access to remote file systems. Nevertheless, a failure on a single file server compromises the operation of all clients, and thus various replication schemes have been devised to increase file system availability.

The approach described in this paper is lightweight in the sense that it strives to make no changes to the NFS protocol nor to the standard NFS client and server code. Rather, a thin layer is introduced between the clients and the original server daemons, which intercepts all NFS requests and propagates the updates to the replicas. Replication is hidden under a primary-secondary update policy and an improved automounter. If the primary server fails, the automounters elect a new primary and remount the relevant file systems. Secondary server failures remain unnoticed by the clients.

A prototype version is operational and preliminary results under the Andrew benchmark are presented. The figures obtained show that while read overhead is negligible, the performance of updates is at present impaired by the naive synchronous multi-server write operation.

1 Introduction

With the introduction of personal computers, individual users achieved a large independence from centralized host systems. However, the consequent partitioning of a unique file system resulted in a major waste of valuable resources. It was usual to find identical data on unshared file systems. On the other hand, although NFS has been quite successful in supporting data sharing on local area networks, it does

*Financed by JNICT grant BM / 2646 / 92-IA
†Financed by JNICT grant BM / 3556 / 92-IA
‡Under contract JNICT PMCT 163/90

so at the expense of reintroducing dependencies, often centralized ones. Once again, the failure of a single file server can block several client machines.

This problem motivated the introduction of replication schemes, which increase the availability of a remote file service with a moderate increase in processor and file system resources. This is the case of the **repNFS** system (replicated NFS) described in this paper. It is aimed at providing NFS-compatible file services in the presence of occasional server failures, but with almost no changes to the underlying system software — both client and server.

The goal here is simplicity (hence its lightweight approach). Reducing the number of changes to the original software is likely to ease the switch from NFS to **repNFS**, especially in heterogeneous networks. It also means fewer administrator and end-user surprises, such as unfamiliar behaviour or error messages. Finally, simplicity will hopefully lead to small overheads.

2 Previous Approaches

Systems such as **Coda** [2, 4], **Locus** [7] and **Echo** [10, 1] achieve high availability using their own file system (and kernel), instead of the standard NFS. Although ensuring tight integration, this approach requires a considerable commitment to a specialized system, thus reducing portability and being of limited value for networks with existing heterogeneous systems.

Other systems try to comply with the NFS protocol, though changing the traditional NFS client and providing new server daemons that enforce data replication policies. Examples of this approach are the **RNFS** system [5], its follow-up **Deceit** [8], and **Ficus** [3]. Since the changes to the NFS client code usually involve kernel manipulation, this approach also implies a strong investment on particular Unix implementations. The NFS client must be enhanced with the capability to commute to another server upon server failures. By contrast, **repNFS** avoids kernel changes by using an improved automounter as an alternative switching mechanism.

The use of special-purpose NFS server daemons also contributes to operating system dependencies with respect to the local file system interface, as specific (Unix) flavours and versions must be accommodated. The alternative used in the **repNFS** system is to provide the necessary capabilities in a special layer over the normal NFS server daemons. Despite its potential overhead, this solution is highly portable.

Table 1 compares several systems on the basis of the client kernel code that deals with file system operations, the daemons on the replicated servers and the communication protocol.

3 repNFS System Overview

The **repNFS** system offers a highly available file service by coordinating file replication among an arbitrary number of machines and applying file coherence politics. This is achieved by a small extension to the NFS system, in the user level processes,

	Client Kernel	Server Code	Protocol
Coda	Specific	Specific	Specific with Callbacks
Locus	Specific	Specific	Specific
Echo	Specific	Specific	Specific
RNFS	NFS slightly Modified	NFS Modified	NFS and ISIS
Deceit	NFS slightly Modified	NFS Modified	NFS and ISIS
Ficus	NFS Modified	NFS Modified	NFS Modified
repNFS	Same	Intercepted	NFS

Table 1: Comparison of approaches to replication

thereby avoiding kernel changes. This use of user level processes to provide additional capabilities to the file system has been previously advocated in **Ficus** [3], with the notion of stackable layers of file system services.

On the client side, **repNFS** uses AMD[6], an improved automounter that enables run-time server switches between a group of servers. The traditional Sun automounter is able to choose a server among some alternatives, but once chosen it is committed to that server. In the case of failure it cannot select a different one. By contrast, the AMD automounter constantly monitors the known servers, and once one server is found to be unavailable any affected mounts are removed and an alternative server is chosen for its replacement. At the moment, the sequence of election among available servers is pre-defined by assigning different weights to each server.

On the servers side, the server that is elected by the AMD automounter becomes the primary server. In addition to providing normal file system service to the remote clients, it propagates relevant calls to the secondary servers. Under normal circumstances, all servers are therefore synchronized.

The basic idea in **repNFS** is to intercept the client NFS calls before they reach the original NFS server daemons. This is accomplished by changing the NFS server daemons RPC registration numbers, and registering our **repNFS** daemons instead. Although this approach requires the modification of the NFS daemons, it is very localized, as it just requires the change of two numbers (associated with `mountd` and `nfsd`). In our case, the source code of the publicly available **Linux** NFS daemons was used; it compiled cleanly under **SunOS 4.1.3** and successfully replaced **SunOS** NFS daemons.

Replicated servers for a specific file system are organized in groups. To each file system $f_i \in F$ (any exportable subtree of files), a subset Sr_{f_i} of replicated servers is associated so that $Sr_{f_i} \subseteq S$, being S the set of all servers. This set is defined as the group of servers that keep a replica of that file system. With this information one can derive, for each server $s_j \in S$, the set $Sa_{s_j} \subseteq S$ of associated servers, the servers that share some file system with the server s_j. For each $s_j \in S$, $Sa_{s_j} = \bigcup_i (Sr_{f_i} : f_i \in F \wedge s_j \in Sr_{f_i})$.

The associations Sr_{f_i} are stored in a single text file in the format defined for

AMD maps. This file is distributed to all machines (servers and clients) by actual copy or using NIS[9]. As an example, the following AMD map

```
f1   type:=nfs ; rfs:=/dir_f1 \
     rhost:=A rhost:=B rhost:=C
f2   type:=nfs ; rfs:=/dir_f2 \
     rhost:=B rhost:=C
f3   type:=nfs ; rfs:=/dir_f3 \
     rhost:=C rhost:=D
```

defines the replicated servers sets $Sr_{f_1} = \{A, B, C\}$, $Sr_{f_2} = \{B, C\}$, $Sr_{f_3} = \{C, D\}$, and the correspondent associated servers sets $Sa_A = \{A, B, C\}$, $Sa_B = \{A, B, C\}$, $Sa_C = \{A, B, C, D\}$, $Sa_D = \{C, D\}$.

The current implementation forces the replication groups to be disjoint, although permitting multiple file systems replicated within each group. This restriction covers the most useful topology of replication; a non-disjoint approach would lead to unclear interdependencies among machines without offering significant advantages. This restriction can be formalized as $\forall f_i, f_j \in F, (Sr_{f_i} \cap Sr_{f_j}) = \emptyset \vee (Sr_{f_i} \cap Sr_{f_j}) = Sr_{f_i} = Sr_{f_j}$ which implies that for each $f_i \in F, \forall s_j \in Sr_{f_i}$ it holds $Sa_{s_j} = Sr_{f_i}$.

As a result, the group of replicated servers can be determined for each machine $s_j \in S$, regardless of the file system that the client call addresses. This group Sr_{s_j} is identified by any Sr_{f_i} to which the local machine belongs, i.e. $Sr_{s_j} = (Sr_{f_i} : s_j \in Sr_{f_i})$. Additionaly we can observe that $Sr_{s_j} = Sa_{s_j}$.

repNFS uses a primary-secondary server update policy. This was prompted by statistics collected throughout a 5-month period in our department's main Unix server. These showed that only roughly 10% of all NFS operations are updates. The other are read requests or can be satisfied from the local cache. In this asymmetric approach, upon a client mount request, the selected server s_j is responsible for satisfying all read requests and replicating all update requests among Sr_{s_j}. It also manages the translation of file handles among the replicated servers. The replicated update commands originated in the primary server are delivered to its own NFS daemons and to those in the other servers, using the changed RPC registration number.

In the case of failure of one server, a subsequent respawn of the **repNFS** and NFS daemons will put them in a recovery mode that prevents assuming server functions should a client issue a mount request. In recovery mode the server ignores AMD queries, therefore appearing unavailable. Normal mode of operation is resumed once they are updated by another server in the same group. If the failing server was the primary server — the one that receives the clients mount requests — the AMD automount daemon on each clients commutes to the next alternative server in the group. As this implies the removal of any client mounts on the previous server, any file handle to file name associations cached on the clients are automatically destroyed. This prevents any inconsistent use of file handles with the newly selected primary server.

The primary server is also responsible for detecting among his group of servers those requiring recovery. Should it be necessary, a separate recovery process is

launched, the servers are updated and then returned to the normal mode of operation. This defines three possible states to a replicated server, as shown in figure 1.

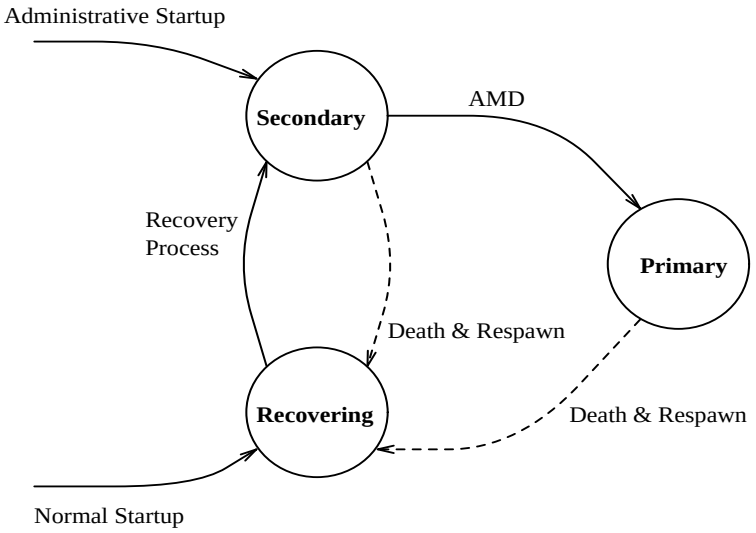

Figure 1: State transitions in **repnfs** servers.

If all servers in a group fail at the same time, as in a local power down, the detection of the server with the most recent changes is made by querying all servers in that group, as every server keeps track of other servers' status (every s_j should know the status of all the servers in Sa_{s_j}). Only when that server is up and available, or by external administrative procedures, can the system be synchronized to the most recent state, and other servers switched to the normal operation mode.

4 Architecture

Figure 2 shows how the **repNFS** (rep.*) daemons couple with the NFS daemons (rpc.*). The standard NFS services RPC registration numbers are 0x100003 (2049) and 0x100005, the numbers 0x100040 and 0x100041 are the new registration numbers for the NFS daemons. We can also see how the operations are redirected to the appropriate daemons.

All NFS operations use file handles to refer to files or directories. Each NFS server generates its own file handles and returns them to the client for later use. With multiple servers, the same logical replica is referred to by different file handles. Since each client only knows one of the multiple file handles, an association between file handles that denote the same logical replica must be kept.

The **repNFS** layer maintains a list of tuples that associate file handles denoting the same file/directory across different servers; the primary key to this list is the file handle of the primary server. The complete file name is also stored, as it

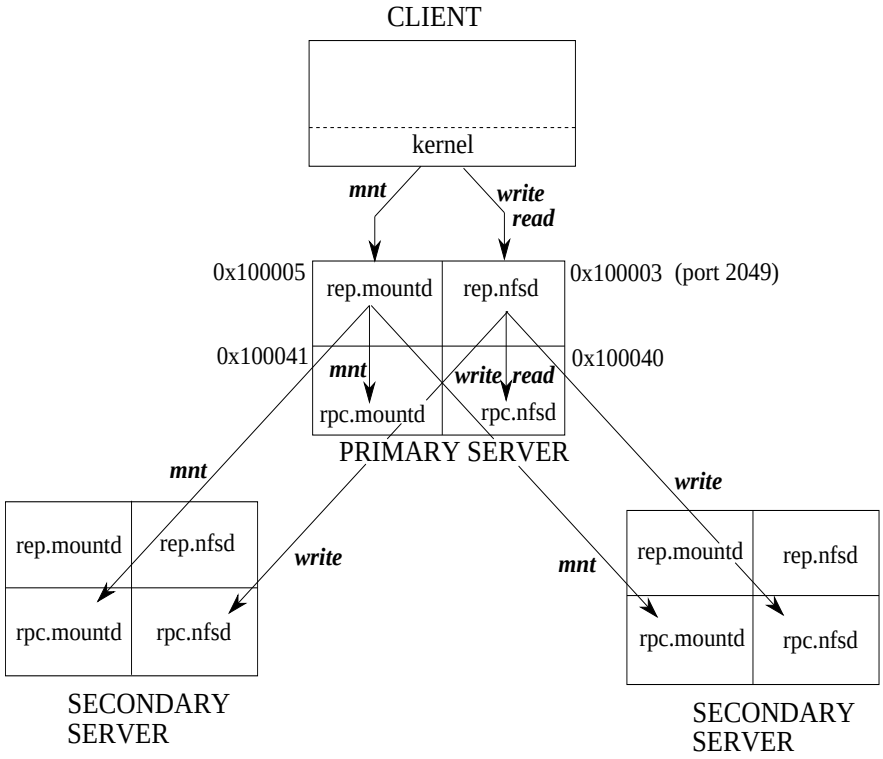

Figure 2: Client servers interaction, in the presence of the **repNFS** layer.

will be necessary in the context of the recovery procedure. Incoming NFS request are validated and then forwarded with their file handles properly translated. The NFS request received in the primary server include the credentials of the user that generated those requests. These same credentials must be used on the forwarded requests, otherwise a user *anonymous* would be implied.

An unfortunate consequence of file handle translation is the fact that the update packets addressed to the replicas have (slightly) different contents. This precludes the use of a broadcast approach to packet delivery, and leads to the use of multiple RPC calls (one for each server in Sr_{s_j}, where s_j is the primary server), with the corresponding performance penalty.

If, in the course of an update request, a server unavailability is detected by the primary server, this is registered in a black list on persistent storage on the remaining servers of the group, marking the state of the file systems as "stale". The black list in each server s_j keeps information on the state of all servers in Sa_{s_j}. For each associated server, the list holds an entry with its name, a flag denoting if the files systems on that server are up to date, a flag indicating if the entry is valid or not, and the time of the last update to the entry. When a server moves into *recovering* mode (see figure 1) all the entries in its black list are marked as

invalid. In this mode the server tries to recover the elements on the list by querying other servers in Sa_{s_j}. When queried about an element of its black list, a server only answers if that element is marked valid. In all modes, a server is available to change the information on the list accordingly to other servers requests. This set of rules ensures that if at least one of the servers in a group remains operational, the black list is valid and can be used to update the other machines.

In case of global failure within one group, each machine, in a subsequent reboot, will first try to update its black list by querying the other machines. Since those machines cannot give an accurate answer as the entries are invalid on all machines, a second protocol is initiated. One fixed machine within each group will request all the entries with the associated time stamps, from the other machines in that group. The information with the most recent time stamp is chosen and disseminated as valid to the other machines. It is assumed that all machines will be rebooted, in order to determine the most recent state, and so the use of a fixed machine to apply these queries is not an additional constraint. Naturally, administrative procedures can circumvent this behaviour.

The recovery process is initiated on a primary server that detects the availability of a server marked down. When this process, by constantly monitoring the unavailable server, learns that it was just rebooted, it asks the **repNFS** daemon to start logging all the updates directed to that server. Since these file handles are different from server to server, the logged operations must refer them by "free" variables. These free variables are represented by the complete pathnames associated to the file handles. When the log is processed these variables are replaced by the corresponding file handles generated in the server being brought up to date.

After requesting the generation of the log, the recovery process compares the file systems of the primary server s_p and the one being recovered s_r. This is accomplished by mounting the s_r file system in the primary server s_p and transversing both trees in parallel. Based on timestamps (and giving some margin for clock skews), the appropriate operations are issued to get the two servers almost synchronized, i.e. except for the operations that were requested in the meantime, which are in log. Whole files are transferred.

As the synchronization of the two file systems is done concurrently with potential changes to the s_p file system, the version of the s_r file system that will be obtained in the secondary server may already incorporate some of the logged operations. If we tag as σ_{t_0} the file system state of s_p when the log is started, and σ_{t_1} its state when the transversal of the file system trees ends, the state σ_{t_x} of the obtained s_r file system will be $\sigma_{t_0} \leq \sigma_{t_x} \leq \sigma_{t_1}$. However, the information registered in the log suffices to derive σ_{t_1} from σ_{t_0}. Since NFS operations are idempotent, we can obtain σ_{t_1} in server s_r by applying the log operations to its state σ_{t_x}.

Before executing the logged operations on server s_r, one must query for the file handles of the newly created files in order to update the list maintained in the primary server s_p. The free variables (pathnames) on the log will be instantiated while executing log operations on s_r. When the log is empty, the **repNFS** layer stops logging client update requests and all servers in the group Sa_{s_r} update their

black lists. The server s_r is now up to date and so it can can now resume normal mode of operation, in a *secondary* state (see figure 1). Note that although triggered by a client update request, the recovery process does not entail substantial delays other than those associated to log maintenance.

Figure 1 also shows that the system may be started in two different modes. Prior to the first execution, the system administrator must ensure that all the replicas are synchronized and then launch the daemons in a special mode indicating that all the machines are coherent and that the black list can be constructed and placed in persistent storage. The other mode of launching the daemons will be typically selected in a startup file, and assumes that the other servers must be queried to obtain the current state.

5 Conclusions

The first version of the **repNFS** system is operational. It was tested with the Andrew benchmark [2, 8], also used elsewhere to measure **Coda**, **Deceit** and **Ficus** performance. This test showed the relatively small overhead introduced by our approach to replication.

The benchmark involved operations on a subtree of 125 files totaling 670kbytes in size. Five distinct phases named `MakeDir`, `Copy`, `ScanDir`, `ReadAll` and `Make` were timed. As noted above, the large majority of the traffic corresponds to phases that do not require update operations. With 1 to 3 replicated servers, the overhead imposed by **repNFS** ranges from 2.5% to 2.8% over the time taken by the native Sun NFS system. This shows that the overhead introduced by the additional level of indirection and book-keeping operations is minimal.

The average of all phases (including updates) already shows overheads with respect to SUN NFS of 18%, 86% and 110%, for 1 to 3 replicated servers, respectively. These figures clearly indicate that the **repNFS** performance is dominated by the update policy of the alternative servers: we simply use a sequence of synchronous RPC calls. Since server updates can proceed in parallel (e.g. using a multi-threaded layer), we estimate that **repNFS** overhead can be reduced to the 18% value, as the wait time for replication will be conditioned only by the time of the slowest update. If this is confirmed, the system will then be tested in a production environment.

The **repNFS** system is intended to be lightweight both in the overheads introduced and in the interference with the underlying operating system. Other alternatives involve kernel changes in the client, which are here avoided by the use of a special automounter. The changes to the NFS server daemons are also minimized by intercepting and redirecting the client requests in a wrapper process. This, and the fact that **repNFS** solely relies on the widely accepted NFS protocol, avoiding direct interaction with the local file system, leads to greater portability and adaptability to changes.

References

[1] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital, Systems Research Center, 130 Lytton Avenue, Palo Alto, September 1993.

[2] M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computers*, 39(4):447–459, April 1990.

[3] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[4] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[5] K. Marzullo and F. Schmuck. Supplying high availability with a standard network file system. In *Eighth Intl. Conf. on Distributed Computing Systems*, pages 447–453, May 1988.

[6] J. Pendry and N. Williams. *Amd, The 4.4 BSD Automounter*. Imperial College and University of California, March 1991.

[7] G. J. Popek and B. J. Walken. *The Locus Distributed System Architecture*. MIT Press, 1985.

[8] Alexander Siegel. Deceit architecture. June 1991.

[9] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[10] G. Swart, A. Birrel, A. Hisgen, and T. Man. Availability in the echo file system. Technical Report 112, Digital, Systems Research Center, 130 Lytton Avenue, Palo Alto, August 1993.