A 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

June 17, 1994

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 severely impaired by the naive synchronous multi-server write operation.

*Financed by JNICT grant BM / 2646 / 92-IA
†Financed by JNICT grant BM / 3556 / 92-IA
‡Supported in part by JNICT PMCT 163/90

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 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. In addition to enhancing portability, it also means fewer administrator and end-user surprises, such as unfamiliar behavior or error messages. Finally, simplicity will hopefully lead to small overheads.

2 Previous Approaches

Some systems achieve high availability by using a tailored file system, such as in **Coda** [**?**, **?**], **Locus** [**?**] and **Echo** [**?**, **?**]. This approach requires a considerable commitment to a specialized system. Others use the traditional NFS client and provide new server daemons that enforce data replication policies, namely the **RNFS** system [**?**] and its follow-up **Deceit** [**?**]. While only introducing special NFS server daemons and showing a small interference with the underlying Unix system, the server code must still suit the specific Unix implementations.

A smaller operating system dependency can be achieved by providing the necessary replication capabilities in a special layer over the normal NFS server daemons. This approach is used in the **repNFS** system.

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, thereby avoiding kernel changes.

On the client side, we use AMD[**?**], 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 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 we used the source code of the publicly available **Linux** NFS daemons; 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. This information is stored in a single text file distributed to all the machines. Upon a client mount request, the selected server is responsible for satisfying all the read requests and replicating all the update requests. It also manages the necessary 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. Fortunately, NFS statistics collected through a 5-month period in our department's main Unix server revealed that only roughly 10% of all NFS operations are updates. The rest are read requests or can be satisfied from the local cache.

In the event 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 upon a client mount request. 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 the clients will commute to the next alternative server in the group. 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.

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. 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 **??** 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.

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 will be necessary in the context of a recovery procedure. Incoming NFS request are validated and the forwarded requests will have their file handles properly translated.

If, in the course of a request, a server unavailability is detected (by the primary server), this fact is registered in a black list on persistent storage on the remaining servers of the group. These marks will be cleared once the machine state is recovered by a primary server. The marks also have an associated timestamp that enables the determination of the most recent marks in case of global failure.

When a server is rebooted, the black list on persistent storage can be recovered. However, the server will try first to obtain an up-to-date black list from an available server. When this is not possible, again due to a global

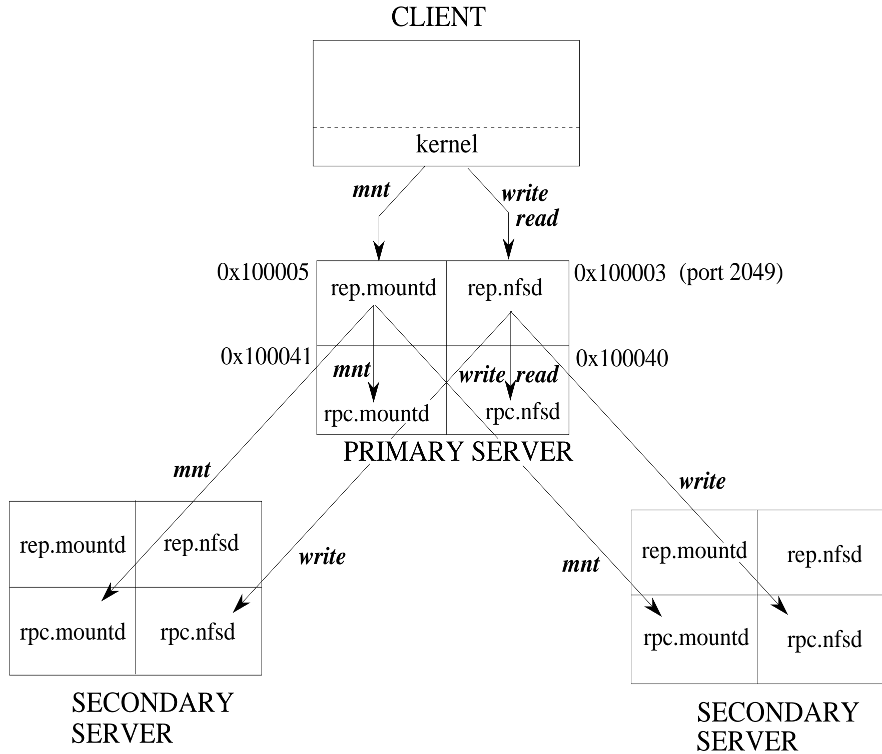

Figure 1: Client servers interaction, in the presence of the **repNFS** layer.

failure, the timestamps are compared and the correct marks determined and disseminated to all the available servers.

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 the new server. The recovery process then compares the file systems of the primary server and the one being recovered. Based on the files timestamps, it issues appropriate operations 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. After this synchronization and some updates of the file handles tuples in the list, the recovery process issues the

update requests that were logged, and finally the recovered server resumes normal mode of operation.

5 Conclusions

The first version of the **repNFS** system is operational. It was tested with the Andrew benchmark [**?**, **?**], also used elsewhere to measure **Coda** and **Deceit** 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 common 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. This, and the fact that we solely rely on the widely accepted NFS protocol, 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] J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[4] 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.

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

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

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

[8] 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.