FT-NFS : an Efficient Fault Tolerant NFS Server Designed for Off-the-shelf Workstations

Nadine Peyrouze, Gilles Muller

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

FT-NFS: an Efficient Fault Tolerant NFS Server Designed for Off-the-shelf Workstations

Nadine Peyrouze, Gilles Muller

N° 2897

Mai 1996

THÈME 1

ISSN 0249-6399

FT-NFS: an Efficient Fault Tolerant NFS Server Designed for Off-the-shelf Workstations

Nadine Peyrouze, Gilles Muller

Thème 1 — Réseaux et systèmes
Projet SOLIDOR

Rapport de recherche n ° 2897 — Mai 1996 — 19 pages

Abstract: In most modern local area network environments, NFS is used to provide remote file storage on a particular server machine. A consequence of this distributed architecture is that the failure of the server results in paralysis or a loss of work for users. This paper presents the design of a low-cost fault tolerant NFS server which can be installed on most Unix networking environments. FT-NFS runs as a user process and does not necessitate any underlying specific operating system functionality. The originality of our approach relies on the use of a stable cache which provides data availability and resiliency to a single failure. The main benefits of the stable cache are first to allow disk write operations to be safely performed in the background and second to permit the gathering of small files in large containers. The latter technique permits disk I/Os to be improved by reducing their number and increasing their length. Under the nhfsstone benchmark, FT-NFS outperforms the in-kernel Sun NFS implementation both in terms of latency and throughput.

Key-words: NFS, fault-tolerance, stable cache, deferred disk write, file gathering

(Résumé : tsvp)

To appear in the procedings of FTCS-26

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

FT-NFS : un serveur de fichiers NFS tolérant les fautes pour des environnements de stations de travail standard

Résumé : Dans la plupart des environnements de réseaux locaux modernes, NFS est utilisé en tant que système de stockage des fichiers sur une machine serveur dédiée. Une conséquence de cette architecture distribuée est que la défaillance du serveur engendre une paralysie ou une perte de données pour les utilisateurs. Ce document présente la conception de FT-NFS, un serveur NFS tolérant les fautes de faible coût pouvant être mis en œuvre dans la plupart des environnements Unix. FT-NFS s'exécute en tant que processus utilisateur et ne repose sur aucune spécificité du système d'exploitation. L'originalité de notre approche réside dans l'utilisation d'un cache stable qui garantit la disponibilité et la cohérence des données en cas de faute simple. Les bénéfices principaux du cache stable sont (i) un report sûr des écritures disque en tâche de fond et (ii) le regroupement des petits fichiers en de vastes conteneurs. Cette dernière technique offre l'avantage d'améliorer les entrées-sorties disque en réduisant leur nombre et en augmentant leur taille. Pour le jeu de test *nhfsstone*, FT-NFS surclasse, en terme de latence et de bande passante, l'implémentation NFS noyau de SUN.

Mots-clé : NFS, tolérance aux fautes, cache stable, écritures disque retardées, regroupement de fichiers

1 Introduction

In modern local area network environments, file storage is often devoted to particular machines, i.e., *file servers*. Access to files by client workstations is then transparently performed by a standard client-server protocol. Among all existing proposals, the Network File System (NFS) protocol [1] is probably the most commonly used protocol. A consequence of this distributed architecture is that the failure of the server machine results in paralysis or a loss of work for users connected to this particular server. While continuous service should be mandatory, since many modern office tasks depend on computers, very few fault-tolerant NFS servers are used in today's ordinary LAN environments. Several reasons can be distinguished: (i) fault tolerance relies on replication of machines which increases the cost of the file server [2], (ii) the recent reliability increase of standard workstations has made hardware failures rare events, (iii) efficient fault tolerant servers have been implemented at the kernel level that limits their spreading.

In this paper, we describe the design and implementation of FT-NFS [3], a low cost fault tolerant NFS server designed for commercial off-the-self workstations. Our main design goal was to provide a solution which could be easily and transparently integrated within standard networking environments in order to ensure the generalization of the use of fault tolerant servers. This led us to make the following choices: (i) user process implementation to avoid kernel modification and improve the server portability onto different operating systems, (ii) passive primary-backup replication mode and the ability to tolerate a single failure (disk or workstation) in order to minimize the cost of hardware redundancy. Our secondary goal was to design an efficient NFS server which could take advantage of the necessary replication to improve performance. This latter goal was achieved by using the main memory of backup and primary servers as a stable cache to provide data availability and resiliency to a single failure.

From the performance (e.g., user) standpoint, the first benefit of the stable cache is that disks can be removed from the critical path between the client and the server, since write operations can be performed after replying to the client. The second benefit is that disk accesses can be reorganized so as to reduce their number and increase their length. For large files, this is done by deferring write operations until there is sufficient data to be written in a large disk write. For small files, we gather them in special files called containers, allowing to replace n small disk write operations by a large one. This led us to introduce a two level filesystem implemented on top of Unix. Despite the a priori drawback of the user process choice on server performance, our measurements show that our server outperforms the in-kernel Sun NFS implementation both in terms of latency and throughput on a Sun-IPX under OS 4.1.3. Also, we have measured a CPU activity on the backup three times lower than on the primary one, allowing the use of the backup for other tasks in normal operating mode.

From the service designer standpoint, the benefit from the stable cache is that it is possible to manage atomically file data and meta-data in the cache without having to take care of their physical disk representation. Also, there are no determinism problems to avoid. Therefore, the service design and its software architecture is made simpler.

The rest of the paper is organized as follows. Section 2 is devoted to related work on NFS improvement and existing fault tolerant NFS servers. Section 3 presents the design overview of FT-NFS. Section 4 details the implementation. Section 5 analyses the performance of FT-NFS and shows the incidence of the proposed mechanisms. Section 6 concludes by presenting future improvements.

2 Background

A fault tolerant NFS server has to ensure the properties of data consistency and continuous service despite crashes. Although files and meta-data consistency is partially provided as part of NFS-v2 protocol specifications [1], its implementation is often made at the expense of performance. Therefore, there have been multiple proposals for enhancing performance of standard NFS servers. We describe them in Section 2.1. Implementation of continuous service requires additional mechanisms based on replication of server machines. We present existing solutions in Section 2.2.

2.1 Improving Performance in Standard Non-Replicated NFS Servers

Most recent proposals for standard NFS servers focus on how to improve performance and how to reduce recovery time in order to increase availability. Performance limitations of NFS has been recently analyzed in [4]. Two main sources have been identified: the stateless property of the protocol and the bounded size of (write) accesses. The stateless property means that neither the server nor clients are required to maintain state information about each other. This property simplifies client and server protocols which do not have to deal with complex situations resulting from crashes: a server crash is viewed by a client as a network delay. The drawback is that effects of write operations on data and meta-data must be forced to stable storage before replying, as the client has no way to detect a subsequent server failure. Furthermore, as NFS requests have a limited size of 8 Kbytes, the result is that large file writes through NFS generate a lot of small synchronous disk I/Os.

Improving disk accesses requires the ratio of data transfer to head positioning duration to be increased. This is generally done by delaying multiple disk accesses and merging them into a single one, either at data or meta-data level, thus permitting an increased throughput. However, from a single client call standpoint, the response time is unchanged. Existing techniques are *write-gathering* [5] and *log batching* [4], *clustering* [6, 7]. The *write-gathering* technique permits the number of meta-data updates to be reduced by processing in a single operation simultaneous write requests issued on the same file by a client. With *log batching*, it is assumed that clients generate sufficient traffic so that meta-data updates (and replies) related to independent requests can be synchronized and gathered in the log by a single disk write without generating an unacceptable delay to the client replies.

Clustering permits head movements to be reduced by allocating several blocks in a large contiguous disk space called a cluster. In local file systems, writes are deferred and a cluster

can be written in a single I/O operation. However, in the context of an NFS file server, this technique does not apply since write operations must be forced to the disk. A solution to this drawback is then to use a non volatile RAM [8, 9]. Nevertheless, this approach still has limitations: (i) the NV-RAM technology is not as reliable as a disk-based stable storage, (ii) if the server machine has to be repaired, the NV-RAM chips cannot be moved easily to another machine, (iii) NV-RAM is not replicated and its failure leads to a inconsistent file system, (iv) NV-RAM is still expensive, thereby limiting the cache size, (v) this technology is not available on standard machines and is currently restricted to high performance dedicated machines.

If a crash occurs while the server is updating the file system, there is no provision in NFS to prevent meta-data inconsistency. This is left to the underlying file system responsibility. Therefore, recent availability improvements in NFS servers mainly come from the use of logging file systems [10, 11, 12, 4]. On large disk partitions, this permits crash restart time to be reduced since replaying a log is often faster than the checking and patching performed by *fsck* [13]. Moreover, it should also be noticed that system reliability is increased by the replication of meta-data structures [4].

2.2 Existing Fault Tolerant NFS Servers

While replication of server machines is mandatory to implement continuous service, two main strategies exist, passive and active replication, which induce different overhead and availability levels. In active replication mode, all server replica are tightly synchronized together. They receive requests and perform processing in the same order. The result delivered to the client is the first server reply in a *fail-silent model*, or is determined by a majority vote if processes behave according to the Byzantine model. The advantage of this approach is to fully mask server failures from the client. However, it leads to a high cost of replication as all replica execute requests. In passive replication, only the primary executes requests and then sends the results to backups. Therefore, the backups load is minimal in normal operating mode and they can be used for other tasks. Nevertheless, a service discontinuity is noticeable by users during take over. In the case of NFS, a break of a few seconds can be easily tolerated since client calls are automatically reissued by the RPC layer.

Existing fault tolerant servers such as RNFS [14], Harp [15] and HA-NFS [16] use a combination of these techniques. RNFS mainly relies on active replication. It uses dedicated agent nodes for managing a list of file replica stored in a group of standard NFS servers. This list is used to filter and replicate client requests using a read-one/write-all replication algorithm. Agents are themselves replicated using the ISIS toolkit [17]. The advantage of RNFS is that it is implemented at user level without modification of the kernel. As described in [14], NFS clients have to be modified to be able to reissue a request to another agent when

the primary one has failed[1]. Due to the active replication and synchronization scheme, the measured performance is worse than the standard NFS.

HA-NFS relies on passive single primary-backup replication. Like FT-NFS, it adheres to the semantics of NFS and a failure is fully transparent to the client. HA-NFS is implemented on top of AIXv3 logged file system which is used to enforce meta-data consistency in case of crash. However, normal write operations are forced to the disk in the same manner as the standard NFS server. Disk availability is provided by storing files on dual-ported mirrored disks which are shared by the primary and backup servers. Transparent redirection of client requests is performed by dynamically assigning the IP/Ethernet addresses of the primary machine to a spare Ethernet interface. From the performance standpoint, the authors report a speedup between 30% to 70% on specific RPC operations due to the use of meta-data logging.

Harp relies on a semi-active replication scheme with a primary machine and one or several backups. Dedicated witness nodes are also used to tolerate network partitions. Harp possesses similarities with FT-NFS. When a request is received by the primary, it is multicast to the backups and a reply is only sent to the client when the primary has received backup acknowledgments. File system modifications are performed in the background. The difference with our approach is that Harp's internal organization relies on a log which is not used to improve disk access operation. Also, file system operations are executed by all nodes and may lead to inconsistency of timestamps. For this reason Harp has to be implemented at VFS kernel level to ensure that file systems are modified with the same timestamps. This approach leads to a very efficient solution but prohibits its use in standard environments where the kernel cannot be modified.

3 FT-NFS Design

This section presents the main choices we made for the FT-NFS design. We first discuss on how to minimize the cost of fault tolerance. Afterwards, we introduce the two main techniques we use to improve performance of the NFS server: stable cache and deferred file gathering. Finally, we show how the server architecture can be configured for either minimizing disk redundancy or tolerating a catastrophic failure.

3.1 Minimizing Cost of Fault Tolerance

One important aspect of the fault tolerance generalization in standard LAN environments, is that it should be obtained with very little additional cost to the end user. This requirement dictates several implementation choices at both the architectural and operating system level. The hardware cost of a fault tolerant system depends firstly on the degree of replication of its components and secondly on the cost of the components themselves. This led us to avoid the use of any specific or uncommon sub-system. Moreover, we chose to tolerate only a single

[1] One reviewer pointed out that transparent redirection of requests has been later implemented using a virtual subnet-based scheme.

sub-system failure (e.g., machine, disk) at a time. This is justified by the fact that reliability of off-the-self computers has increased and hardware failures are becoming increasingly rare. The probability that two machines fail at the same time is thus acceptably low in a LAN environment. Consequently, we chose to build the FT-NFS server from two machines which do not have a common mode of failure.

At the operating system level, we oriented the server design towards passive replication using a primary and a backup server machine. The advantage of this approach is that in normal operating mode, the load of the backup server is sufficiently low so that this machine can be re-used for another job. In the next section, we describe how the primary-backup replication scheme can be optimized through the *stable cache* paradigm.

3.2 Conciliating Fault tolerance and Performance with a Stable Cache

Our basic idea is to use the underlying replication of main memory in the primary and backup servers to build a fast stable memory [18, 19] and to manage it as a cache of files. The stable memory presents properties of resiliency and consistency of data against a single machine failure. These properties are implemented by atomically modifying data structures in stable memory with a two-phase commit protocol.

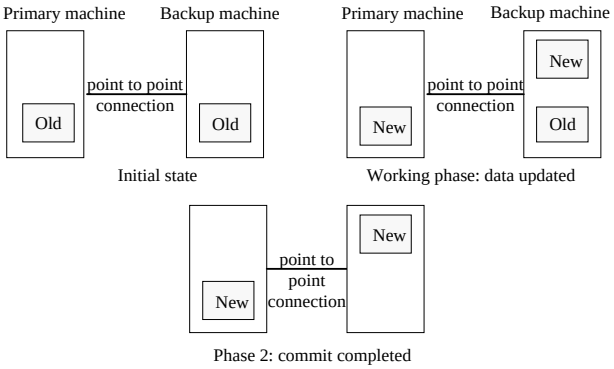

Figure 1: Data management within the stable memory

The commit protocol assumes that the two server machines are connected using a reliable connection such as TCP/IP (see Figure 1):

- Initial state: the primary and backup machines possess the old data version.
- Working phase: after being modified by the primary processor, data is sent to the backup machine. When receiving data, the backup allocates a new copy in memory

to maintain both old and new data versions. This phase is repeated for each modified data structure.

- Phase 1: when the NFS operation is complete, the primary sends a commit message to the backup and waits for an acknowledgment. After receiving the acknowledgment from the backup, the primary knows that the backup possesses all new data structures. It thus resumes servicing.
- Phase 2: after sending the acknowledgment to the primary, the backup discards the old version of data structures.

Assuming that there is only one machine failure at a time, the following situations may arise:

- *failure of the primary machine in working phase or before the backup receives the commit message*; the backup machine discards the new version of modified data and takes over using the old (initial) version of data.
- *failure of the primary machine in phase 2*; the failure does not affect the backup machine which completes the phase and then takes over.
- *failure of the backup machine*; at the hardware level, a backup failure does not affect the primary machine which is able to proceed servicing.

Using a stable cache presents several advantages. First, from the performance standpoint all update operations are performed within the cache and can be safely deferred instead of being immediately forced to the disk. Therefore, the cache overhead of most file modifications is equivalent to a round-trip (n bytes, *ack*) message. Second, the result of operations which deal with several files is always consistent in the event of a crash since several stable structures can be atomically modified in a single operation. Third, unlike NV-RAM based solution, the cache size is only limited by the size of the main memory and can be easily increased to fit the user's needs. This allows the building of a FT-NFS server from heterogenous machines at both the architecture or cache size level.

3.3 Improving Write Operations with Deferred File Gathering

Breaking the disk bottleneck is the challenge of most studies in (distributed) file systems. Improving disk accesses should be done by increasing the ratio of data transfer to I/O initialization latency. This latter duration includes the driver and controller latencies and the head settling time. Most strategies described in Section 2.1 take advantage of a high traffic rate for merging related or unrelated file accesses. In our approach, modifications stored in the stable cache are resilient to failure. For large files, it is then possible to defer disk writes until there is sufficient contiguous data to be flushed in a single large operation. The benefit of this strategy is enhanced when the underlying Unix filesystem implements clustering [7] since it makes possible an increase of the transfer size.

Concerning small files, the only possible solution for write improvements is to write n files in one I/O operation. This technique has been first introduced for log-based filesystems. Because we did not want to modify the file system itself, we have retained a combination of logging and clustering strategies. Small files are gathered in large fixed-size files called containers which are written in a single operation. Containers are allocated at flush time. After being modified, a small file is not flushed back in the same container but in a new one. Therefore, the number of files gathered in a container logically decreases as files are modified or destroyed. When this number reaches zero, the container is reclaimed.

3.4 Disk Storage Organization

While workstation replication provides a solution to server availability, disk subsystem failures have also to be tolerated. Recent years have seen the introduction of commercial RAID subsystems [20]. They provide an adequate solution for masking disk failures while permitting secondary storage scalability. Therefore, a first solution for disk availability, is to share a dual SCSI bus based RAID between the primary and the backup servers (see Figure 2.a). At the disk level, this approach is similar to HA-NFS and has the advantage that in normal operating mode the backup does not have to perform disk operations. However without logging, inconsistency of meta-data is still possible if a failure occurs during an update. To solve this problem, the file system can be modified to take advantage of the stable cache to atomically manage meta-data. As we did not want to modify the operating system itself, we finally retained another solution based on two independent disks (see Figure 2.b).

In this second approach, a disk is tightly associated with a server machine: a failure of the disk is equivalent to the failure of the machine and vice-versa. Therefore, we do not care of a meta-data inconsistency on the disk, as it will be repaired by *fsck* in the reintegration process during which the failed server disk is made consistent with the other (see Section 4.4). However, this approach requires both servers to write their cache to the disk. Cache flushing is performed independently on the primary and the backup and does not require synchronization, but it induces more load on the backup than in a pure passive replication scheme.

While this solution is more expensive in terms of disk redundancy, it is in fact cheaper for a small storage configuration as the default workstation disks can be directly used. Finally, the only constraint on server location depends on the way the connection between servers is physically implemented. This makes possible the tolerance of catastrophic failures (e.g., fire, flood) for instance by placing servers in different buildings and linking them with a fiber optic medium such as ATM or FDDI.

4 Implementation Issues

In this section, we discuss issues related to the implementation of FT-NFS. Our main requirement was to develop a software that could be used on top of most Unix-like operating systems. At the client side, it is imperative to allow the use of FT-NFS by any NFS client

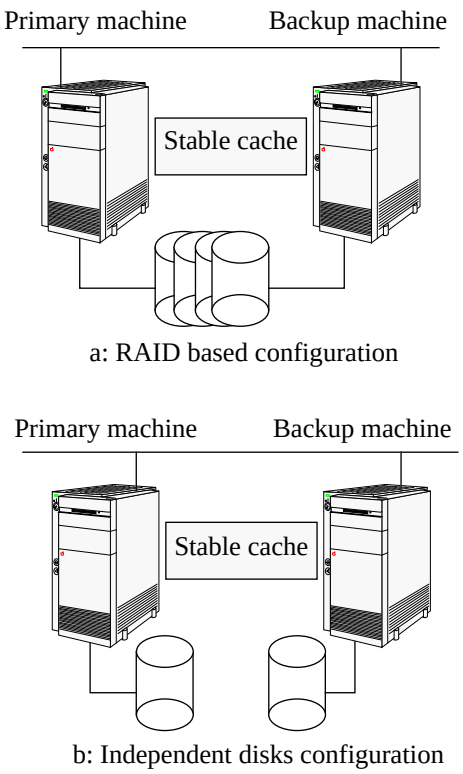

Figure 2: FT-NFS server architecture

without requiring special software configuration. For instance, the main problem which has to be solved is related to the transparent redirection of client requests to the backup, when the primary has failed. In Section 4.5, we describe two solutions to this problem.

On the server side, our portability requirement implies that the kernel cannot be modified. Therefore, we chose to implement a user mode server despite the negative consequences on performance that this choice induces since it increases buffer copies and context switches. We now detail the server's internal structures, the implementation of file gathering and the reintegration of a server after repair.

4.1 FT-NFS Internal Structures

An implementation of file gathering requires small files to be stored on the disk in terms of an associated container and an offset within the container. It is therefore impossible to use directly the Unix names of the files. This led us to design a second file system at the user

level and to separate file management from disk management [21], the latter being provided by Unix. Instead of being internally denoted by an inode, a file in FT-NFS is named with an ID. A first consequence of this choice is that file attributes are kept contiguously with the file data to allow the implementation of hardware links. This has the important advantage that the primary and backup servers possess exactly the same timestamp attributes. A second consequence is that directories have been re-implemented using FT-NFS files. Regular files and directory files are managed and swapped by FT-NFS in the same way. The benefit is that consistency between files and directories is simply implemented by modifying them atomically in a single cache transaction.

The file ID is an index into a file descriptor table describing all files in the system. A file descriptor contains the following information:

- an ID which names the container[2] in which the file is stored on secondary storage. For large files, the file and the container ID are the same,
- the offset of the file within the container,
- the size of the file,
- a list of cache objects describing the file contents cached in stable memory.

A cache object is a table of pointers to 4 Kbyte sized pages which are the allocation grain. In the current implementation, a cache object describes up to 64 Kbytes of data. The total number of pages and cache objects is specified using configuration parameters. Cache objects and pages are allocated during initialisation and stored in a free list from which they are allocated on demand whenever a file is read or written.

Used cache objects are kept in an LRU list for swapping purposes. The swapping policy has been designed to keep files in the cache as long as possible in order to prevent a ping-pong effect between the disk and the cache. It relies on two cache thresholds (85%, 98%): one to start flushing modified cache objects, the other to reclaim memory by freeing already flushed cache and page objects. FT-NFS is built from two processes, the server and the disk driver, which share the cache memory. The server process executes NFS requests and manages the connection with the other server machine. From the server process standpoint, the disk driver allows the implementation of portable asynchronous disk operations on multiple cache pages without introducing physical memory copies.

4.2 Deferred File Gathering

A file is considered small when it fits into a single cache object. Gathering of small files is performed at flush time when the cumulative size of small files is sufficient to fully fill a container cache object (i.e., 64 Kbytes). This is done by allocating a cache object for the container and by moving page pointers from the original file caches to the container one, thus avoiding physically copying pages. The container is then directly written to the disk

[2] An ID is used as a Unix file name by the disk layer.

in one 64 Kbyte transfer. For each small file, the cache object is reclaimed and container offset/ID are updated in the file descriptor. When later accessing a small file, the container is first brought in memory. Then, pages are moved from the container cache to the file one.

4.3 Failure Detection

A backup failure is detected by the primary using the time-out of the primary-backup TCP/IP connection. This approach cannot be used directly by the backup, since there could be long periods of absence of traffic if no modification call is issued by clients. As it is requested that the backup takes over within a few seconds, it regularly sends an NFS null operation to the primary to check that the primary is still alive and really servicing. This approach allows the detection of software bugs in which the server loops forever and operating system problems in which the server is never scheduled.

In a basic configuration, FT-NFS does not support a network failure or partition between servers. However, when a private link is used to optimize primary-backup communications, it is then possible to reach a server by two different paths and to discriminate a link failure from a server failure.

4.4 Fast Server Reintegration after Failure

When reintegrating a server after a failure, the failed disk contents has to be made consistent with the valid one. If the source of the previous failure was the disk itself, a new disk has to be initialized by copying all files from the active server. However, if the failure was due to the workstation or was a scheduled maintenance, most of the files on the failed disk are still valid when restarting[3]. In fact, the invalid files are those which were modified in cache memory at failure time and those which have been modified since. Therefore, duration of the reintegration phase can be minimized by retransmitting only invalid files.

Associated to the file descriptor table, each server keeps in memory two modification vectors $M_i[f]$ which tell for the file f and the server i ($i \in [local, other]$) if the disk contents is consistent with the cache. When a file is modified, the corresponding bit is set in the two vectors. When a file is flushed on the disk, only $M_{local}[f]$ is reset. Periodically, the file table descriptor is flushed to the disk and, only then, the local modification vector is sent to the other server. At any time, a server thus knows a subset of the files which are valid on the other server disk and a superset of possible invalid files. Reintegration is performed asynchronously by the active server while servicing client requests by taking advantage of free periods when no call is received. Exchange of modification vectors is restarted only when the failed server is consistent so that reintegration can be done again in the event of a successive crash.

[3] Fsck is first run when the failed server restarts to rebuild meta-data.

4.5 Re-routing Clients Requests

The major issue in masking server failure is related to how to redirect client calls transparently to the backup server. In Harp, a solution based on IP multicast has been proposed. As stated in [15], this solution has limitations: it may not scale, it does not support reconfiguration very well and it induces a CPU overhead on machines likely to receive calls. HA-NFS has introduced a solution based on a spare Ethernet interface which is given the IP/Ethernet addresses of the primary machine when taking over.

In FT-NFS, we introduce two new solutions. The first one relies on the use of a mobile IP address which is associated with the FT-NFS service. In normal operating mode, the IP address is resolved by the primary server which replies to *arp* requests. When the backup takes over, the IP address migrates from the primary to backup server: the backup broadcasts an ARPOP_REQUEST message [22] containing the association (backup Ethernet address, FT-NFS IP address). When receiving this message all hosts connected to the ethernet segment update their *arp* cache. This allows subsequent requests to the FT-NFS server to be automatically directed to the backup server. This solution is close to the HA-NFS one but does not necessitate an additional Ethernet interface. Moreover, the IP address is associated to the service rather than to the machine. This offers more flexibility in system configuration and allows multiple fault tolerant services to run on the same machine. While being simple and efficient, the drawback of this approach is that the server operating system has to support multiple IP addresses associated to one physical network interface. Today, this is not true for all operating systems. For instance, to add this functionality to SunOS, we had to dynamically load a specific driver initially developed for mobile networking protocols [13].

A second solution to handle request redirection is to use a filter which runs on a third machine. The filter is activated by the backup only when it takes over. It then transforms Ethernet frames containing NFS requests by replacing the primary server Ethernet and IP addresses with the backup ones. This transformation is transparent to the backup which replies directly to the clients. The main advantage of this approach is that it can be implemented for any operating system. However, it consumes some CPU resources on a third workstation since the filter is activated for all received frames. Also, it slows downs response time and increases the network load.

5 Performance Evaluation

FT-NFS has been developed and integrated within the IRISA network which is mainly made up of Sun workstations. More precisely, our testing environment (e.g., client and server machines) is made of several Sun 4/50 IPX workstations running SunOS/4.1.3 with 32 Mbytes of main memory. The primary and backup servers are connected to the same Ethernet segment using a TCP/IP connection; no specific link is used for interconnecting servers. In order to evaluate our proposal, we have made the following experiments:

- comparison between FT-NFS and the standard Sun NFS server,

- evaluation of the overhead due to the stable cache,
- evaluation of file gathering on the write-back policy.

The tests are done using the public domain nhfsstone benchmark [23]. The configuration consists of 10 processes, each accessing 40 files of up to 2 Mbytes. The amount of disk space used by our tests is about 60 Mbytes. To get reproducible results, each nhfsstone execution performed 10000 calls and the results are averaged over at least five nhfsstone executions.

5.1 Comparison Between FT-FNS and the Sun Server

Since our development platform is made of Sun workstations, we have chosen the in-kernel Sun NFS server as a reference for FT-NFS performance evaluation. Figure 3 compares the Sun server with FT-NFS configured with a 3 Mbyte stable cache. A curve ends when the server saturates, i.e., it cannot handle a higher throughput. FT-NFS is about two times faster than the Sun server. From the throughput standpoint, FT-NFS can handle 1.2 times the load of the Sun server before saturation. Since the Sun NFS server also uses read caching, these results clearly show the benefit of our deferred file gathering policy. We also demonstrate that it is possible to obtain nice performance despite the use of a two level file system and choosing a user process level implementation that leads to increased buffer copies.

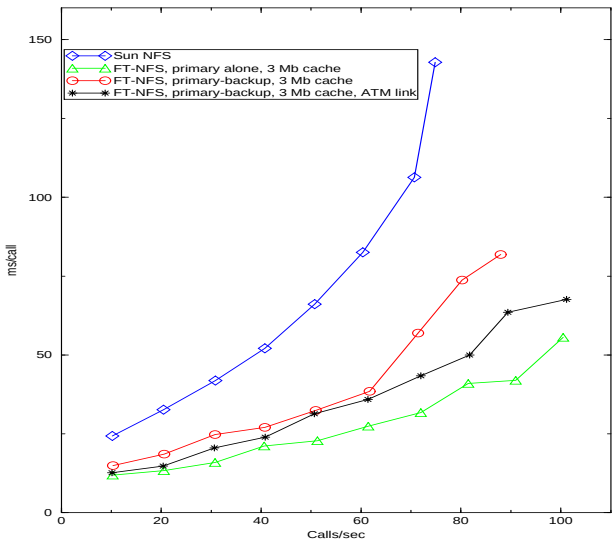

Figure 3: FT-NFS and Sun NFS server performance evaluation for the nhfsstone benchmark

5.2 File Gathering Impact on the Write-back Policy

In order to measure the gain of our file gathering policy, we have made a non gathering version of FT-NFS in which a Unix file is associated with each file. A comparison between the non gathering version and the gathering one is presented in Figure 4. Both versions implement the same deferred write-back policy. Surprisingly, the non gathering version performs even worse than the Sun NFS server. This clearly shows the importance of disk accesses on the overall performance of the server and the gain we have from gathering files to reduce the number of disk accesses.

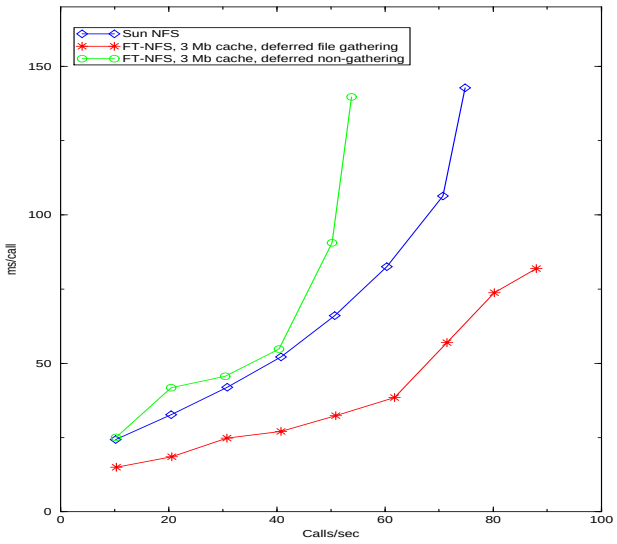

Figure 4: Incidence of file gathering on the write-back policy

5.3 Source of Inefficiencies in the Primary-backup Replication Mode

Analyzing the stable cache based primary-backup replication mode is important to understand sources of inefficiency of our proposal. This is done by comparing the normal primary-backup configuration with a stand-alone server for which there is no stable cache overhead (see Figure 3). Both versions implement the deferred file gathering policy.

The NFS operation which leads to the worse stable cache overhead is a 8192 byte write in one file. Since we have optimized the single file write operation by merging the commit message with the modifications themselves, the cost of writing on stable storage is equivalent to a (header+data:8252 bytes, ack:6 bytes) round-trip. Using Ethernet and a TCP/IP connection this round-trip execution time is about 9.8 ms. When comparing the performance

of the two versions, we observe, for a low throughput, a difference in response time which is almost equivalent to the pure round-trip time. However, when the throughput is over 60 calls/s the difference in response time also increases. Furthermore, the replicated server saturates for a slightly lower load (e.g., 87 calls/s) than the stand-alone one. By monitoring the Ethernet segment with a LAN analyzer, we have observed a high number of collisions and a use of the Ethernet bandwidth above 50-70%. The explanation of this behavior is that the Ethernet segment becomes saturated due to the fact that the primary-backup traffic nearly doubles the network load during burst file write periods.

To achieve higher NFS throughput, we have connected together the servers using a private ATM 100Mbit point-to-point link. The benefit of the ATM link is first to relieve the traffic on the Ethernet network. That permits reaching the same throughput as for the stand-alone server. Second, due to the 100Mbit ATM bandwidth, the stable cache latency is reduced (for instance, the 8252/6 round-trip time falls to 3.8 ms). Therefore, the server latency is also reduced.

6 Conclusion and Future Work

This paper has described the design and implementation of FT-NFS, a fault tolerant NFS server aimed at off-the-self workstations. The main originality of FT-NFS is that it is easy to be ported and it can be installed on most Unix networking environments by the system administrator. To achieve this, FT-NFS runs as a user process and does not rely on specific operating system functionality such as logging. FT-NFS currently runs on Sun 4 under SunOS/4.1.3; a port is undertaken on NetBSD for PC architectures.

FT-NFS relies on a primary-backup replication scheme and the stable cache paradigm which provide data availability and resiliency to a single failure. The main benefits from the stable cache are first to allow disk write operations to be safely performed in the background and second to permit the gathering of small files in large containers. The latter technique permits disk I/Os to be improved by reducing their number and increasing their length. FT-NFS adheres to the semantics of NFS and a (primary or backup) server failure is fully transparent to the client. Redirection of client requests is provided by means of a mobile IP address which does not require an additional network interface. A minor drawback induced by file gathering is that a Unix filesystem cannot be directly mounted by FT-NFS. Files need to be converted either by a copy through FT-NFS or by a special tool we developed.

Despite the a priori negative impact of the user process choice on performance, our measurements show that FT-NFS performs about two times better than the standard Sun NFS server in response time and increases the throughput by 25%. By using a private ATM link between the two servers, we also demonstrate that is possible to reach the same throughput between our stand-alone server and a primary-backup one.

After running a set of nhfsstone benchmarks, we have measured the CPU resource consumption on the primary and backup. It shows that the backup activity is three times lower than the primary one. Therefore, the backup can be easily reused for another task. One solution could be to let the backup serve another partition and use either the primary

or a third node as a backup. A second solution, which we are working one, is to enable the backup and the primary to serve the same partition. This implies extending the stable cache with recoverable DSM-like protocols [24] so that both servers could access the same set of files.

Acknowledgments

The authors are grateful to the anonymous referees for their usefull suggestions in making the presentation of this paper clearer. We also wish to thank C. Morin and S. Billiart for having read earlier versions of this paper. P. Gachet and J.P. Routeau helped us implement FT-NFS and improve system performance; we are grateful to them for this.

References

[1] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. of Usenix 1985 Summer Conference*, pages 119–130, Portland, June 1985.

[2] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, second revised edition, 1990.

[3] N. Peyrouze. *Conception et réalisation d'un système de gestion de fichier NFS efficace et sûr de fonctionnement*. PhD thesis, université de Rennes I, September 1995.

[4] U. Vahalia, C. G. Gray, and D. Ting. Metadata logging in an NFS server. In *Proc. of 1995 Usenix Technical Conference*, pages 265–276, New Orleans (LA), January 1995.

[5] C. Juszczak. Improving the write performance of an NFS server. In *Proc. of Winter 1994 Usenix Technical Conference*, pages 247–259, January 1994.

[6] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[7] L.W. McVoy and S.R. Kleiman. Extent-like performance from a UNIX file system. In *Proc. of Winter 1991 Usenix Technical Conference*, pages 1–11, January 1991.

[8] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon. Breaking through the NFS performance barrier. In *Proc. of Spring 1990 European UNIX Users Group Conference*, pages 199–206, April 1990.

[9] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of 1994 Winter Usenix Technical Conference*, pages 247–259, January 1994.

[10] R.S Finlayson and D.R. Cheriton. Log files: An extended file service exploiting write-once storage. In *Proc. of 11th ACM Symposium on Operating Systems Principles*, pages 139–148, November 1987.

[11] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware. Technical Report 11, Digital, Western Research Laboratory, Palo Alto, October 1989.

[12] M. Seltzer, K. Bostic, M.K. McKusick, and C. Staelin. An implementation of a log-structure file system for UNIX. In *Proc. of Winter 1993 Usenix Technical Conference*, pages 307–326, January 1993.

[13] T. Kowalski. *FSCK: The UNIX System Check Program*. Bell Laboratory, Murray Hill, March 1978.

[14] K. Marzullo and F. Schmuck. Supplying high availability with a standard network file system. In *Proc. of 8th International Conference on Distributed Computing Systems*, pages 447–453, San Jose, June 1988.

[15] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 226–238, 1991.

[16] A. Bhide, E.N. Elnozahy, and S.P. Morgan. A highly available network file server. In *Proc. of Winter 1991 Usenix Technical Conference*, pages 199–205, Dallas (TX), 1991.

[17] K. Birman and R.V. Renesse. *Reliable distributed computing with ISIS toolkit*. IEEE Computer Society Press, 1994.

[18] B. Lampson. Atomic transactions. In *Distributed Systems and Architecture and Implementation : an Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1981.

[19] G. Muller, M. Banâtre, N. Peyrouze, and B. Rochat. Lessons from FTM: an experiment in the design & implementation of a low cost fault tolerant system. *to appear in IEEE Transaction on Reliability*, June 1996.

[20] P.M. Chen, E.K. Lee, A. Gibson, R.H. Katz, and D.A. Patterson. Raid: High-performance reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[21] W. de Jonge, M.F. Kaashoek, and W.C. Hsieh. The logical disk: A new approach to improving file systems. In *Proc. of 14th ACM Symposium on Operating Systems Principles*, December 1993.

[22] D.C. Plummer. An Ethernet address resolution protocol. RFC 826, 1982.

[23] B. Shein. Nfsstone - a network file server performance benchmark. In *Proc. of Usenix 1989 Summer Conference*, pages 119–130, June 1989.

[24] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. Technical Report 975, IRISA, December 1995.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399