Supplying High Availability with a Standard Network File System[1]

Keith Marzullo and Frank Schmuck

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

ABSTRACT

This paper describes the design of a network file service that is tolerant to fail-stop failures and can be run on top of a standard network file service. The fault-tolerance is completely transparent, so the resulting file system supports the same set of heterogeneous workstations and applications as the chosen standard. To demonstrate that our design can provide the benefit of highly available files at a reasonable cost to the user, we implemented a prototype based on the Sun NFS protocol. Our approach is not limited to being used with NFS, however. And, the methodology used should apply to any network file service built along the client-server model.

1 Introduction

There are two approaches to building fault-tolerant distributed programs. The first is to choose an available programming abstraction that reasonably fits the problem at hand (*e.g.* transactions (*e.g.* [7]), replicated procedure calls [4] or reliable objects [2]) and implement the program using the abstraction. The second is to follow a *methodology* in designing the fault-tolerant

[1]Partial support for this work was provided by the National Science Foundation under grant CCR-8702294 and by the Defense Advanced Research Projects Agency (DoD) under ARPA Contract N00140-87-C-8904. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy or decision.

program, and implement the program using available programming abstractions.

The RNFS project is an example of the second approach. We designed and implemented a network file service that is tolerant to fail-stop failures [9] and can be run on top of NFS [12], a standard network file service. The fault tolerance is completely transparent, so the resulting file system can support the same set of heterogeneous workstations and applications as NFS supports. The fault tolerance does come with a performance cost, but this cost can be limited to only those files that need to be highly available.

NFS was chosen primarily because it was available in our department, although its simple and stateless protocol did make our task easier. Our approach explained here is not limited to being used only with NFS, however. The model we used could be used with any network file service built along the client-server model.

The purpose of this paper is two-fold. First, we wish to show the flexibility of the design methodology we chose: the *state machine* approach [10]. Second, we wish to demonstrate the use of the *ISIS* broadcast protocols and toolkit routines [3] in building a fault-tolerant program. However, it turns out that the resulting program is an interesting file system of its own right.

The rest of the paper proceeds as follows. In Section 2, a general fault-tolerant design methodology is reviewed. This methodology is applied to the NFS system in Section 3, giving a network file service that is resilient to failure of servers in which clients access files through an *agent*. The methodology is applied again in Section 4 in order to make the file service resilient to agent failure. In Section 5 we look at the performance cost incurred in the prototype implementation of the system. We summarize results and discuss future plans with the system in Section 6.

2 Design Methodology

The design of RNFS follows the state machine approach to building fault-tolerant distributed systems. In this section, we briefly outline of the part of the design methodology most relevant to RNFS.

A fault-tolerant network service is developed in three steps:

1. Design the server as a state machine that responds to a set of messages. At this point, only the correctness of the server need be considered; messages are assumed to arrive correctly and in FIFO order from clients of the server.

2. Replicate the server as a set of identical state machines. Clients communicate with servers using an atomic broadcast protocol. Clients receive results from the set of servers by filtering the responses. The filter depends on the failure model. For example, under a fail-stop model the client need only wait for one response; under a Byzantine model the client needs to receive a majority of identical responses.

3. Use knowledge about the semantics of the server to reduce the complexity of the broadcast protocol. For example, if all the operations of the state machine are commutative, the atomic broadcast can be replaced by a broadcast that does not preserve order. This could be the case, for example, if the server were a vote counter that responded to each vote with an indication of whether a majority has been reached. The order the individual votes reach different counters is unimportant.

As will be discussed later, we applied this methodology twice.

There are several systems that clearly resemble the state machine model; for example [1] and [13]. In fact, they, as well as the RNFS, can be cleanly described using this model.

3 Replicating Servers

In order to build a highly available file service on top of a simple network file service, we replicate a server. To do this, we treat a network file server as the state machine referred to in step 1 of Section 2.

3.1 Sun NFS Protocol

The NFS protocol provides RPC access to a Unix-like file system. The main points of importance to us about this protocol are:

- It is a "stateless" protocol. A client does not open a session with the server nor any file on the server. For example, there is no way to open a file in "append-mode". Instead the client must provide an absolute offset into the file for every READ and WRITE operation.
- The NFS protocol is idempotent: operations do not have an incremental effect. Because of this and the statelessness of NFS, all a client has to do in order to recover from a crashed server is simply continue trying the failed RPC until the server reappears.
- A client accesses a file using a handle (called *fHandle*) obtained from the server as a result of a LOOKUP operation. Different implementations of NFS interpret the semantics of *fHandles* differently: some treat them as true object names for a file, while others may issue different *fHandle* to different clients. In our implementation, we assume that a *fHandle* can serve as a true object name. This is consistent with the current implementation of NFS supplied by Sun.
- Updates are synchronous with respect to failure. When a WRITE RPC completes, the client is assured that the data has been written. Again, this is a property not met by all NFS implementations, although the current implementation supplied by Sun does have this property.

3.2 Agents

We achieve fault-tolerance by replicating the server and using an atomic broadcast protocol for communication. We would, however, like to leave the client as unchanged as possible. Were the client to directly communicate with the servers, it would need to use a version of the RPC built on an atomic broadcast transport. Additionally, an NFS server is not strictly deterministic. For example, when a new file is created, two different servers might issue different *fHandles*.

Therefore, we achieve replication of servers transparently by directing all client requests to an intermediary called an *agent* that in turn broadcasts to all of the servers. To the client, the agent appears to be a file server that has an exceptionally reliable secondary storage. In reality, the agent uses a set of servers to store replicas of every file. The agent uses virtual *fHandles* (called *rHandles*) to identify each file or directory it manages. Associated with each *rHandle* is a set of *fHandles*, one for each replica of the file. The agent maintains a *replicated file-list* for mapping *rHandles* to *fHandles* and vice versa. In a LOOKUP operation, a client presents a file name and asks for a handle on that file. The agent forwards the request to one of the servers, finds the resulting *fHandle* in the replicated file-list and returns the associated *rHandle*.

We chose a read-one/write-all file replication scheme since it is optimized for READ operations and we are assuming only fail-stop failures. The agent performs a WRITE by forwarding the request to each server with the *rHandle* replaced by the appropriate *fHandle*. The agent replies to the client when all of the writes are completed. This ensures that the NFS write-through semantics are preserved.

There are some other complications due to non-deterministic behavior of NFS servers. For example, the various time stamps kept with a file will be different for each replica and the order in which files are kept in a directory might be different on different servers. As with *fHandles*, the agent must create virtual values. For example, the agent stores the maximum value of all replica time stamps in the replicated file-list, and a directory read from a server is sorted before it is passed on to the client.

3.3 Server Failure

When a server fails, all the replicas on that server become inaccessible. When this happens, the agent notes all replicas on the failing server as *unavailable*. When a write is done to a file with a replica on a failed server, the agent marks the unavailable replica as *invalid* and updates all the available replicas. Thus, each replica in the replicated file-list has an associated *state* of type $\{up, down\} \times \{valid, invalid\}$. The first part of the state caches the status of the server holding the replica while the second half indicates whether the replica represents a suitable latest version of the file.

When a failed server is again available, the agent changes the state of all replicas on the server from (*down, x*) to (*up, x*). Each replica with state (*up, invalid*) is then transferred to an (*up, valid*) state. File recovery is done concurrently with file access; write access to a file is blocked only during the short time an (*up, invalid*) copy of that file is recovered.

Rather than logging the operations missed by a *down* replica, we recover a file by replacing the replica with a copy of a *valid* replica. The implementation of recovery for non-directory files is straightforward and fits well with the typical size and usage of Unix files. Recovery of directories by copying is more complex. It is necessary to compare the two directories and use file creation, deletion and renaming to effectively copy the valid directory to the invalid one.

3.4 Agent Failure

Much of the information in the replicated file-list is irreplaceable. In order to protect against agent failure, the agent maintains a stable version of the replicated file-list called the *stable file-list*. At a minimum, the stable file-list must contain the mapping from rHandle to *fHandles* and whether each replica is *valid* or *invalid*. This file is itself replicated on all the servers and is expensive to update, but changes to the stable file-list are only necessary when a file is created or deleted (or changes state as described below).

When the agent fails, the state of the service remains in the latest version of the stable file-list. The recovering agent must locate a replica with the latest version. We use a version of the protocol described in [11] to locate a server with such a replica.

The only way that two valid replicas of a file can be different is if the agent crashed while writing the file, since it is possible that only some of the writes were successful. We could eliminate this possibility by running a two-phase protocol, but this would be expensive and require either changing the server or frequently writing to the stable file-list. Instead, we rely on the fact that a client does not expect a write to be stable until it receives a response from the agent. Doing this, however, requires the recovering agent to ensure each valid replica of a file be identical. A file cannot be accessed until the recovering agent ensures this property.

In order to avoid some unnecessary recovery, we also keep in the stable file-list an indication of whether or not all valid replicas of a file are identical. All valid replicas are identical if no write is currently in progress. The agent could flag a file in the stable file-list every time it started a write and remove the flag whenever the write was completed, but this would add a large overhead to writes. Instead, we leave the file flagged until a period of time elapses with no activity to the file. We call a flagged file ACTIVE and an unflagged file PASSIVE. A file is made PASSIVE by writing the latest replica write time to the stable file-list. Any subsequent write will automatically make the file ACTIVE.

4 Replicating Agents

The most obvious problem with the service described in Section 3 is that the agent is a critical process; while it is down, all files are inaccessible. Thus, the service is slower and no more robust than an NFS server. Following our methodology again, we replicate the agent. Unlike servers, we can guarantee that all copies have

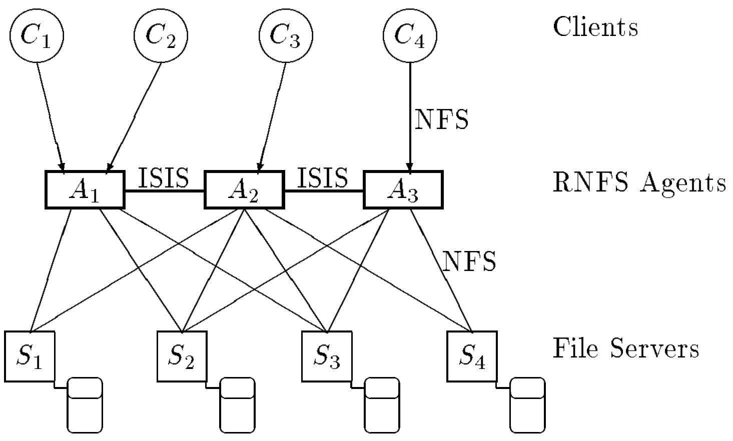

Figure 1: Client, agent and server interaction in RNFS

identical state, so we need no intermediate.

We would still like to avoid changing the clients to use atomic broadcast RPCs for communication with the agents. Also, many operations can be processed by a single agent making it unnecessary to broadcast the request to all agents. Therefore we chose to let the client select one agent to send its requests to. If necessary, this agent will communicate with the other agents using an atomic broadcast (such as the one provided by ISIS). Figure 1 illustrates this structure.

Thus we only need to modify the client to have it react to an agent failure. The special semantics of NFS — statelessness and idempotency — simplify the problem of making agent failure transparent. The client simply retries the operation by redirecting its requests to a different agent.

4.1 ISIS

The ISIS system developed at Cornell provides a set of communication protocols and tools for developing fault-tolerant distributed applications. Below we describe how features of ISIS are used in the design of agent replication for our file service. A more comprehensive description of ISIS and its underlying principles may be found in [2] and [3]. The key features for our purposes are:

- *Fault-Tolerant Process Groups.* Agents are structured as a fault-tolerant process group under ISIS. ISIS provides location transparent communication with a process group and among its members. When an agent crashes, the ISIS failure detection mechanism first completes all pending broadcasts and then informs the other group members of the failure.

- *Group Broadcasts.* Each agent maintains locally cached information that must be kept consistent with the state of the other agents: for example, the replicated file-list and the current status of all servers. ISIS supports the management of replicated data by providing a set of broadcast primitives for propagating updates to all members of a group. Depending on consistency requirements, the programmer may choose between atomic broadcast primitives that guarantee a global order on all updates and less strongly ordered, but more efficient, broadcast protocols.

- *State Transfer.* When a new agent is added to the group or when an agent is restarted after a crash, it needs to initialize its local data structures. The ISIS state transfer tool allows an agent to be integrated into the group while the system is running. This is accomplished by transferring all necessary data from an existing group member to the new agent. It ensures that the transfer is atomic with respect to updates.

- *Coordinator-Cohort.* Some actions need to be performed by a single agent, while it is still necessary to guarantee that the action is completed even if the agent fails. The recovery of a server is an example for such an action. ISIS provides a tool for structuring a computation like this. One member of the group (the *coordinator*) is chosen to execute the action, while the other members (the *cohort*) monitor its progress and are prepared to take over for the coordinator should it crash.

4.2 Tokens

Because clients don't broadcast their requests to all agents, some other mechanism is needed to properly synchronize concurrent updates to a replicated file. Therefore, we constrain the agents to exclude each other when writing to a file: associated with each file is a *write-token* that an agent must acquire before updating any replica of the file. We describe the token passing mechanism in some detail, as it provides an example for a replicated data structure that is maintained using ISIS broadcasts.

The implementation must ensure that no tokens are lost when an agent crashes. Therefore, information about the current token holder is replicated at every agent. An agent asks for the token by broadcasting a *request* message. The holder of the token, after completing the current operation, broadcasts a message "*pass-to-*X" in order to transfer the token to the other agent ("X"). Every agent listens to these broadcasts and keeps track of the current token holder as well as any pending token requests. Should the holder of the token crash, another agent is chosen arbitrarily to "inherit" the token.

Notice that it is not necessary for all agents to agree on the same order for pending requests, because the current token holder decides which agent will get the token next and this decision is part of the *pass* message. This allows us to choose an efficient broadcast primitive (the ISIS CBCAST protocol) for implementing token operations. The ISIS failure detection mechanism still guarantees that token operations are atomic with respect to failures. This ensures that all agents have the same view about which tokens an agent was holding when it failed.

The token mechanism also serves as a tool for synchronizing updates to the stable file-list. Furthermore, it allows us to detect possible inconsistencies between the replicas of a file. Such inconsistencies can be caused by the failure of an agent during an update: every ACTIVE file on which the failed agent held a write-

	indirection	replication	synchronization
READ	1 RPC	-	-
WRITE by 1			
first	1 RPC	$(n-1)$ NFS	1 ISIS-bcast
following	1 RPC	$(n-1)$ NFS	-
WRITE by > 1			
every write	1 RPC	$(n-1)$ NFS	1 ISIS-bcast
CREATE	1 RPC	$(2n-1)$ NFS	2 ISIS-bcast

Table 1: RNFS overhead for various NFS operations on n replicas

token may have inconsistent replicas. The agent that inherits the token resolves these inconsistencies before allowing further updates to the file. This approach has the advantage of deferring the cost of detecting and recovering partially completed updates to the time when a failure occurs.

5 Performance

In this section we show that our design can provide the benefit of highly available files at a reasonable cost to the user. Table 1 summarizes the overhead of RNFS operations compared to NFS. All RNFS operations incur some overhead due to the fact that client requests have to pass through an agent. The cost for this extra level of indirection is that of a standard RPC. Because RNFS uses a read-one/write-all algorithm, there is no additional overhead involved in the case of READ operations.

WRITE operations are more expensive because all replicas of a file have to be updated. If n is the degree of replication of a file, then RNFS needs to make n NFS calls to the servers in order to process a single WRITE request. In other words, the overhead for RNFS WRITE's due to replication is $(n-1)$ NFS operations. This results in higher network traffic and increased overall load on the NFS servers. However, if an agent uses broadcast style RPCs to perform updates on all replicas in parallel, this overhead does not directly affect the total time it takes to process a request. The delay observed by the client would only depend on how long it takes the slowest of the NFS severs to update its replica.

An additional source of delay for RNFS updates is the cost of synchronizing concurrent WRITE's between agents. Before an agent can start writing to the replicas of a file, it needs to send out an ISIS broadcast in order to acquire the write-token for that file. In the case where no more than one client writes to the file, this cost is paid only for the first write operation. Once the agent is holding the token, it can perform all following updates without interacting with other agents through ISIS. If, on the other hand, two or more agents receive a sequence of requests to update the same file, the token needs to be passed back and forth between these agents. In the worst case, the token passing overhead adds a delay to every single WRITE operation. However, [5] shows that, at least in a Unix environment, concurrent updates to the same file are rare, and that, when a file is updated, typically the entire file is written. Our synchronization mechanism is designed to perform well in this setting, because the cost of acquiring the token is amortized over a large number of WRITE operations.

Finally, we consider the cost of creating a new replicated file. As an update on a replicated directory, a CREATE operation involves the same kind of overhead as writing to a replicated file: the agent needs to acquire the write-token for the directory and forward the CREATE request to all servers. In addition to this, the new file needs to be inserted into the replicated file-list. The cost for this is one extra ISIS broadcast for sending the replicated file-list entry to the other agents and n additional NFS operations to update the stable file-list on all servers.

A first prototype of RNFS, which supports the full NFS protocol, was completed in November 1987. Table 2 compares the total time of READ, WRITE, and CREATE operations under standard NFS and RNFS. READs and WRITEs were measured on 2k blocks of data over a 10 Mbit Ethernet with SUN 3/50 workstations and servers. The measurements were performed without any caching on the client side.

In the prototype, the broadcast style RPC for updating replicas is

	NFS	RNFS		
		$n = 1$	$n = 2$	$n = 3$
READ	16 ms	24 ms	24 ms	24 ms
WRITE, 1 writer	80 ms	100 ms	180 ms	230 ms
WRITE, n writers	80 ms	240 ms	320 ms	370 ms
CREATE	80 ms	210 ms	350 ms	480 ms

Table 2: NFS versus RNFS; n is the degree of replication.

not implemented. Consequently, the figures in table 2 show that the delay for WRITE and CREATE operations increases roughly linear with the degree of replication. Adding broadcast RPCs should bring down the cost of updating multiple replicas to about the the cost of updating a single replica.

With some improvements, we expect RNFS to be no more than 1.5 to 2 times slower than NFS. We belief that in a system with local caching on the client side, the performance of RNFS would be perfectly acceptable for the typical user.

6 Conclusions

We have found that supporting fault tolerance on top of a standard file system is neither overly hard nor expensive. The total time needed from design to a demonstratable prototype was under six months. We expect to have a version of the file service that will be robust and fast enough for widespread department use within five more months.

In designing RNFS, we needed to replicate a service twice: once to supply multiple servers and once to supply multiple agents. This allowed us to assess the power of the ISIS primitives, since we could not use them for the first replication step. Much of the difficulty discussed in Section 3 arises from having to deal with the lack of support for replication at this level.

The extra replication step was necessary because an NFS server is not deterministic. So, we could not exactly implement rule 2 of Section 2. This extra replication step could also be used to accommodate a collection of disparate file system protocols.

Our preliminary experiences with RNFS have been very encouraging. When used with workstations that supply client-side buffering, the extra cost of replication for the most part is not noticeable. We expect that the users will want to have mainly text files and system files (such as the password file) replicated. The typical kind of access to these files leads us to believe the observed performance cost for high availability will be small.

One problem we need to address in the future is scaling to larger networks. This is clearly a problem for any file system that will be used in a reasonable workstation environment ([6] and [8]). Our current architecture will most likely have to be modified before RNFS can be put into such an environment. It will be interesting to see if we will be able to incorporate any of these architectural changes into the state machine methodology.

7 Acknowledgements

The basic architecture of the RNFS was first designed by the authors and Susie Armstrong. Susie Armstrong and Eliot Morrison implemented much of the system, and Ken Birman and Tommy Joseph helped in the design.

References

[1] A. J. Bernstein. A loosely coupled system for reliably storing data. *IEEE Transactions on Software Engineering*, SE-11(5):446–454, May 1985.

[2] Ken Birman. Replication and availability in the ISIS system. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 79–86. ACM SIGOPS, 1985.

[3] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh*

Symposium on Operating System Principles, pages 123–138. ACM SIGOPS, 1987.

[4] Eric Cooper. Replicated distributed programs. In *Proceedings of the Symposium on Operating Systems Principles*, pages 63–78. ACM SIGOPS, December 1985.

[5] J. K. Ousterhout et. al. A trace-driven analysis fo the 4.2 BSD UNIX file system. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 15–24. ACM SIGOPS, 1985.

[6] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributes file system. *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[7] Barbara Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[8] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[9] Fred B. Schneider. Byzntine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[10] Fred B. Schneider. The state machine approach: A tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Upson Hall, Ithaca, NY 14853, December 1986.

[11] Dale Skeen. Determining the last process to fail. *ACM Transactions on Computer Systems*, 3(1):15–30, February 1985.

[12] Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043. *Networking on the Sun Workstation*, revision B edition, February 1986.

[13] Premkumar Uppaluru, W. Kevin Wilkinson, and Hikyu Lee. Reliable servers in the JASMIN distributed system. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 105–112. IEEE Computer Society, 1987.