Replication in the Harp File System

Barbara Liskov
Sanjay Ghemawat
Robert Gruber
Paul Johnson
Liuba Shrira
Michael Williams

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper describes the design and implementation of the Harp file system. Harp is a replicated Unix file system accessible via the VFS interface. It provides highly available and reliable storage for files and guarantees that file operations are executed atomically in spite of concurrency and failures. It uses a novel variation of the primary copy replication technique that provides good performance because it allows us to trade disk accesses for network communication. Harp is intended to be used within a file service in a distributed network; in our current implementation, it is accessed via NFS. Preliminary performance results indicate that Harp provides equal or better response time and system capacity than an unreplicated implementation of NFS that uses Unix files directly.

1. Introduction

This paper describes the replication technique used in the Harp file system. (Harp is a **H**ighly **A**vailable, **R**eliable, **P**ersistent file system.) Harp provides highly available and reliable storage for files: With very high probability, information in files will not be lost or corrupted, and will be accessible when needed, in spite of failures such as node and media crashes and network partitions. All modifications to a file are reliably recorded at several server nodes; the number of nodes depends on how many failures the file is intended to survive. We take advantage of replication to provide a strong semantics for file operations: each operation is performed atomically in spite of concurrency and failures.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, and in part by the National Science Foundation under grant CCR-8822158. Sanjay Ghemawat and Robert Gruber were supported in part by National Science Foundation Graduate Fellowships. The Digital Equipment Corporation provided support under an external research grant.

Harp uses the primary copy replication technique [1, 26, 27]. In this method, client calls are directed to a single *primary* server, which communicates with other *backup* servers and waits for them to respond before replying to the client. The system masks failures by performing a failover algorithm in which an inaccessible server is removed from service. When a primary performs an operation, it must inform enough backups to guarantee that the effects of that operation will survive all subsequent failovers.

Harp is one of the first implementations of a primary copy scheme that runs on conventional hardware. It has some novel features that allow it to perform well. The key performance issues are how to provide quick response for user operations and how to provide good system capacity (roughly, the number of operations the system can handle in some time period while still providing good response time). Harp achieves good performance by recording the effects of modification operations in a log that resides in volatile memory; operations in the log are applied to the file system in the background. Essentially, it removes disk accesses from the critical path, replacing them with communication (from the primary to the backups), which is substantially faster if the servers are reasonably close together.

In using the log to record recent modifications, Harp is relying on a write-behind strategy, but the strategy is safe because log entries are not lost in failures. We equip each server with a small uninterruptible power supply (UPS) that allows it to run for a short while (e.g., a few minutes) after a power failure; the server uses this time to copy information in the log to disk. The combination of the volatile log and the UPS is one of the novel features of Harp.

Harp provides reliable storage for information. Information survives individual node failures because it exists in volatile memory at several nodes. It survives a power failure because of the UPS's. Also, Harp attempts to preserve information in the face of simultaneous software failures by techniques explained later in the paper.

Harp supports the virtual file system (VFS) [19] interface. It guarantees that operations have really happened when they return, i.e., their effects will not be lost in subsequent

failovers. In fact, all operations in Harp are implemented atomically: an operation either completes entirely, or has no effect, in spite of concurrency and failures.

Harp is intended to be used within a file service in a distributed network, such as NFS [31, 35] or AFS [17]. The idea is that users continue to use the file service just as they always did. However, the server code of the file service calls Harp (via the VFS interface) and achieves higher reliability and availability as a result. Harp makes calls to low-level Unix file system operations. Thus, the Harp code is just a small layer in the overall system, as illustrated in Figure 1-1.

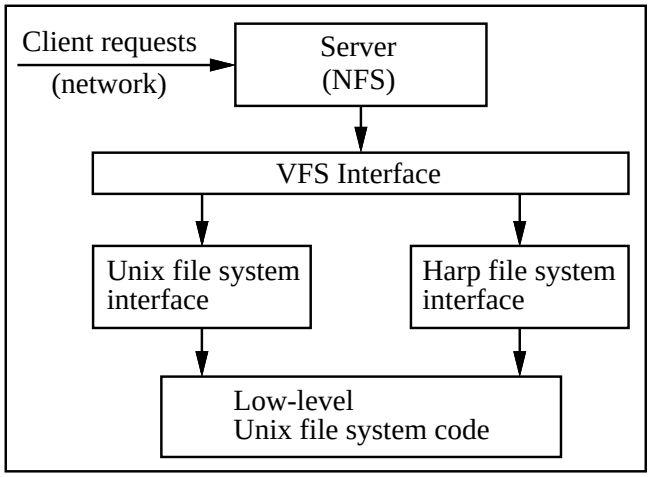

Figure 1-1: Harp System Structure

In the current implementation, users use Harp via NFS. We guarantee that the combination of the NFS code and Harp appears to the user to behave like an unreplicated NFS system; as discussed in Section 4.5, this requires a little more work than just implementing the VFS calls correctly. Harp can be used with any VFS-based NFS server implementation, and should be portable to most Unix systems. We believe it can also be used by other network file systems that use VFS, or similar systems, such as the ULTRIX generic file system [29], but we have not yet investigated such a use.

This paper describes how replication works in Harp and provides some preliminary information on system performance. The portion of Harp that handles processing of user operations has been implemented; we are working now on the failover code. The performance data indicate that Harp will perform well: in the experiments, Harp performs as well or better than an unreplicated implementation of NFS that uses Unix files directly both in terms of response time to users and in overall system capacity. The results show that high availability can be achieved without degradation of performance by using a small amount of additional hardware (extra disks to hold the extra file copies, and UPS's).

The remainder of the paper is organized as follows. We begin by discussing related work, and then describe our assumptions about the environment. Section 4 describes the replication technique — how the system works in both the normal case and in the presence of failures. Section 5 describes the current status of the implementation, gives performance measurements for various benchmarks, and compares the performance of Harp with the standard implementation of NFS in our environment. We conclude with a discussion of what we have accomplished.

2. Related Work

As mentioned, we make use of work on replication techniques [4], especially the primary copy technique [1, 11, 26, 27], view change algorithms [10, 9, 13], and schemes that use witnesses [21, 28]. One of the first implementations of primary copy replication appears in Tandem's NonStop System [2, 3]. The NonStop System significantly differs from Harp in its use of special-purpose hardware (e.g., dual interprocessor bus, dual port disk controllers, and mirrored disks). The idea of using a UPS to avoid the delay of writing the log to disk appears in [7], where it is used in conjunction with a voting algorithm [12].

The related file system work can be divided into three categories: projects that provide a new file system, projects that support NFS or VFS, and projects that use logs but not replication.

File systems like Locus [37], Coda [32], and Echo [16] differ from our work because they are complete network file systems with their own (non-Unix) semantics. These systems involve both client and server code, and use client caching to improve performance. Harp runs only at the server, and is independent of any caching that may occur at the client (in fact, most NFS implementations do client caching).

For the purposes of comparison what is interesting about these systems is their replication techniques. The Echo system, which is currently under development, uses a primary-copy scheme with a write-ahead log that is written directly to disk Echo can be configured to have a primary and backups with separate disks, but it allows other configurations (e.g., just a primary, or a primary and backup that share a disk).

Coda and Locus use replication to improve both availability and response time. Files have many replicas; clients can send requests to any replica, and operations are performed without synchronization. The lack of synchronization means that file system state may become inconsistent if there are concurrent modifications on the same file. The systems use a multipart timestamp scheme to detect consistency problems, which must then be resolved by users. (Coda also allows "disconnected operations", in which a client node can make changes to a file while it is disconnected from the network; again inconsistencies are detected using the multipart timestamps.) In contrast, Harp uses replication only to provide availability and reliability and it ensures consistency of the file system

state. It provides performance by spreading the load so that different servers act as primaries for different file systems.

There are several projects that provide high availability for NFS or VFS. The HA-NFS system [5] uses a variant of the primary copy scheme with a write-ahead log to provide high-availability and atomicity, but the log is written directly to disk (or disks), which are shared by the primary and backup, and the system as a whole does not handle partitions (it runs on a local area network). Since the log is written to disk, the performance of HA-NFS for writes is worse than unreplicated NFS (since the modifications are also written to the file copies on disk later) and we expect it will be worse than Harp's. In addition, HA-NFS's reliance on hardware to determine which node is the primary makes it less flexible (e.g., in providing differing numbers of backups). The Deceit system [22] also provides high availability within NFS but uses a different replication method, namely ISIS [6]; we expect to achieve better performance. (For example, Deceit has higher overhead than HA-NFS [5].) The Ficus system [14] supports high availability for VFS. It uses the same replication scheme as Locus, and therefore inconsistencies in file system state are possible.

Finally, some file systems [15, 18, 30] have used the write-ahead log technique within a server. These systems provide atomic file operations and they speed up recovery by avoiding the need for scavenging after a crash (e.g., by running the Unix fsck utility). We also do not need to scavenge, but fast recovery of a single node is not a major issue for us since Harp continues to provide service while the node is recovering. Some of these systems [15, 18] write the log synchronously to disk, which reduces response time and system capacity. Avoiding this write, as is done in [30], means that effects of committed operations can be lost when there is a crash. We avoid this loss by the combination of replication (for single failures) and the UPS's and other techniques (for simultaneous failures).

3. Environment

Harp is intended to run in an environment consisting of a distributed collection of *nodes* that communicate by means of a *network*. The network might be a local area net or a geographically distributed net such as the Internet. Nodes fail by crashing. Crashes may occur because of power failures, software errors, or hardware failures. The network may lose or duplicate messages, or deliver them late or out of order; in addition it may partition so that some nodes are temporarily unable to send messages to some other nodes. As is usual in distributed systems, we assume the nodes are fail-stop processors [33] and the network delivers only uncorrupted messages. We assume that nodes have clocks that are loosely synchronized with some skew ε. Synchronized clocks are maintained by a clock synchronization protocol [23] that provides a skew of less than a hundred milliseconds.

Some nodes are *servers* and others are *clients*. Clients send messages to servers to request some service; servers accept such messages, carry out the request, and if needed return a response in a message to the client. Harp runs at server nodes. Ideally the servers would run only the file system (including perhaps some additional processes that monitor system behavior). The restriction is not required by our algorithm but would allow us to provide better response to clients. Also, the servers will be more robust if they do not run user code.

Each server node is equipped with a UPS, which allows it to continue to run for a brief period after a power failure. An alternative is to use a battery backed-up memory; the node stops running but the memory contents survive the failure. We chose to use UPS's because they are cheap, available, processor independent, and provide some protection against power surges. In addition, UPS's allow us to provide uninterrupted service across short power failures, which are quite common in our experience.

4. Replication Method

In this section we describe the replication technique used in Harp and the most important and interesting implementation details. Our aim is to convey how the approach meets our goals of providing both good behavior (atomicity, high availability and reliability) and good performance.

We begin by giving an overview of our approach. Section 4.2 describes how the system behaves in the absence of failures Section 4.3 describes the failover mechanism; it also states the system requirements and discusses how we satisfy them. Section 4.4 provides additional detail on how we meet our atomicity requirement. The last section discusses some higher-level issues.

4.1. Overview

We use a primary copy method [1, 11, 26, 27] as our replication technique. This technique was developed for use in general transaction systems. We have adapted it to match the needs of this application. We take advantage of the fact that even though each individual file system operation must run atomically, support for transactions containing several operations is not needed. In addition, we have incorporated the use of a volatile log into the technique.

In any replication method, any particular file is managed by a *group* of servers that store copies of the file and respond to user requests. In a primary copy method, one of these servers acts as the *primary*, and client requests are sent just to it. The primary decides what to do and communicates with the other servers in the group as needed; these other servers are the *backups*.

Modification operations require a two-phase protocol. In phase 1, the primary informs the backups about the operation. When the backups acknowledge receipt of this information, the operation can commit. At this point the primary returns any results; the second phase, in which the backups are informed about the commit, happens in the background. In traditional primary copy schemes, opera-

tions that do not involve modifications to the file system also require communication with the backups, but we use a novel technique (described in Section 4.2) that allows these operations to be done entirely at the primary.

When a failure or a recovery from a failure occurs, the group runs a failover protocol called a *view change* [9, 10, 13]. The result of a view change is a reorganization within the group, in which a failed node is removed from service, or a recovered node is put back into service. The result of such a reorganization is called a *view*; one of the nodes in a view is the primary of that view and the others are backups. The primary of the new view may be a different node than the primary of the old view. When the primary changes, client requests need to be directed to the new primary. We discuss how this happens in Section 4.5.

As with any replication scheme that tolerates network partitions, we require 2n + 1 servers in order to continue to provide service to clients in the presence of n server failures [4]. For example, to continue to provide access to files in the case of any single failure, there must be three servers. Traditional replication methods keep file copies at all 2n + 1 servers. However, we store only n + 1 copies, since this is enough to allow information to survive n failures. The other n servers need to participate in view changes concerning that file to ensure that only one new view is selected, even when there is a network partition, but they do not store copies of the file. We call these additional servers *witnesses* following the terminology in [28], where the idea was first proposed.

In Harp, each replica group manages the files of one or more Unix file systems, i.e., for a particular file system, all files will have copies at the same group. One of the servers is the *designated primary* (this means it will act as the primary whenever it can). Another n servers are *designated backups*. The remaining n servers are the *designated witnesses*. Only the designated primary and backups store copies (on disk) of the files in the file systems managed by that group.

A good way to organize the system is to arrange the groups so that each node acts as the designated primary of one group, the designated backup of another, and the designated witness of a third. In this way the workload can be distributed among the servers; if the load becomes unbalanced, the system can be reconfigured, e.g., to move a file system from one group to another.

In the current implementation, groups have three members. Such a group has a designated primary, one designated backup, and one designated witness, and in any view there will be a primary and a single backup. Harp must have at least three servers, but there can be more, e.g., four servers supporting four different three-server groups. Although our algorithms will work for groups of other sizes, to simplify the discussion we consider only three-member groups in the rest of this paper.

4.2. Normal Case Processing

The primary maintains a *log* in volatile memory in which it records information about modification operations. The log contains a sequence of *event records*; each record describes an operation, and records at higher indices describe more recent operations. Some records are for operations in phase 1, while others are for operations that have committed. The primary distinguishes between these by maintaining a *commit point*, or CP; this is the index of the latest committed operation. Operations commit in the order of their records in the log.

To carry out a modification operation, the primary creates an event record that describes the modification, appends it to the log, and sends the logged information to the backup. The backup also maintains a log. As new log entries arrive in messages from the primary, the backup appends them to its log and sends an acknowledgment message to the primary. A backup only accepts the records in log order, so that an ack for entry n indicates that all records up to and including entry n have been received.

When the acknowledgment arrives from the backup, the primary commits the operation by advancing its CP, and returns any results. The primary sends its CP in each message to the backup. The backup also maintains a CP in which it stores the largest CP it has received in messages from the primary.

Both the primary and the backup maintain copies of the file systems managed by the group on disk. However, the modifications described in an event record are not applied to the primary or backup's file system until after they commit, i.e., until their entry number is less than or equal to the CP, and therefore we are using a write-ahead log [4]. Event records for committed operations are applied to the file system in the background. This work is carried out by a separate *apply process*.[1] The apply process performs the file system reads and writes needed to carry out the event's operation by using asynchronous Unix file system operations. It processes the event records in log order and maintains a counter called the *application point*, or AP, that records its progress.

The apply process does not wait for the writes associated with one event record to happen before moving on to the next. Instead the actual writing to disk is carried out by the Unix file system in the background. Another Harp process keeps tracks of which writes have completed. It maintains a *lower bound* pointer called the LB; all records with indices less than or equal to the LB have had their effects recorded in the file system on disk.

Records are removed from the log when they have been recorded on the disk copies of the file system at both the primary and backup. Primaries and backups send their LB's in messages to one another. Each maintains a *global* LB, or GLB, which is the lower bound on what it knows of

[1]The Harp implementation runs at kernel level and consists of several processes.

the current LB's for itself and its partner, and discards log entries with indices less than or equal to its GLB. The volatile state at the servers is summarized in Figure 4-1.

Harp runs with a bound on the size of the log. When the log gets close to full, Harp forces the completion of writes for some portion of the records beyond the current LB. This allows the GLB to advance so that earlier portions of the log can be discarded. We delay forcing since this makes it more likely that new operations can be performed using information in primary memory. Therefore, we expect the number of event records in the log to be large.

Log	holds the event records.
CP	most recently committed event record index.
AP	most recently applied event record index.
LB	largest index such that events with indices $\leq$ the LB are on local disk.
GLB	largest index such that events with indices $\leq$ the GLB are known to be on disk at both the primary and the backup.

Invariant: GLB $\leq$ LB $\leq$ AP $\leq$ CP $\leq$ top of log

Figure 4-1: Volatile State at a Server

When a server recovers from a failure, the log will be used to bring it up to date. Since only committed operations are applied to the file system (at the primary and the backup), the log is a *redo* log [4]: the records contain only sufficient information to redo the operation after a failure. Using a redo strategy simplifies the implementation since we avoid logging the additional information that would be needed to undo operations after a failure.

Operations that do not modify file system state (e.g., get file attributes) could be handled similarly to modification operations by making entries in the log and waiting for the acknowledgment from the backup, but this seems unnecessary because such operations do not change anything. Instead, Harp performs non-modification operations entirely at the primary. Such operations are performed as soon as they arrive; their results reflect all committed writes and no uncommitted writes. (I.e., they are serialized at the CP.)

Doing non-modification operations just at the primary can lead to a problem if the network partitions. For example, suppose a network partition separates the primary from the backup, and the backup forms a new view with the witness. If the old primary processes a non-modification operation at this point, the result returned may not reflect a write operation that has already committed in the new view. Such a situation does not compromise the state of the file system, but it can lead to a loss of external consistency [13]. (A violation of external consistency occurs when the ordering of operations inside a system does not agree with the order a user expects. It requires communication outside of the file system, e.g., one user telling another about a change in a file.)

We make a violation of external consistency unlikely by using loosely synchronized clocks [23]. Each message from the backup to the primary contains a time equal to the backup's clock's time + δ; here δ is a few hundred milliseconds. This time represents a promise by the backup not to start a new view until that time has passed; we expect that starting a new view will not be delayed, however, because δ seconds will have elapsed by the time the new view is ready to run. The primary needs to communicate with the backup about a read operation only if the time of its local clock is greater than the promised time.[2] When a new view starts, its new primary cannot process any modification operations until the time of its clock is greater than the promised time of the backup in the previous view. In this way we guarantee that writes in the new view happen after all reads in older views unless clocks get out of synch, which is highly unlikely.

Almost all Unix file system operations modify the file system state. In particular, file reads are modification operations because they update the file's "time last accessed" field. By treating a file read as a modifier, we can ensure that this time is always consistent with what users expect, but at a high cost: we must communicate with the backup as well as read the file. The communication time can be masked if the information to be read is not in primary memory, but otherwise reads will be slower than in an unreplicated system, since the primary will need to wait for the acknowledgment from the backup before returning the results.

Unix applications rarely make use of the "time last accessed" field. Therefore, we allow Harp to be configured in two different ways. File reads can be treated as modification operations if desired. Alternatively, we provide a weaker implementation in which an event record is added to the log for the file read, but the result is returned before the event record commits. The "time last accessed" in such a record is chosen to be greater than that of any committed conflicting write, but earlier than that of any uncommitted conflicting write. Since the read returns before its event record is sent to the backup, there is a small chance that the event record will be lost if there is a failure, causing the "time last accessed" for the file to be visibly inconsistent across a view change.

4.3. View Changes

The requirements for the Harp implementation, including both normal case processing and view changes, are the following:

- *Correctness*. Operations must appear to be executed atomically in commit order.

[2] Actually, the primary needs to account for the clock skew also. It needs to communicate with the backup if its current time is greater than or equal to the promised time - ε, where ε is the clock skew.

- *Reliability*. The effects of committed operations must survive all single failures and likely simultaneous failures.
- *Availability*. We must continue to provide service when any two group members are up and can communicate, provided we can do so while preserving the file system state correctly.
- *Timeliness*. Failovers must be done in a timely manner, and furthermore, failovers in common cases (a single failure or recovery from a single failure) must be fast, in the sense that the time during which users cannot use the system is very short. Ideally, the failover should be invisible, so that the user does not notice any loss of service.

This section describes our view change algorithm and how Harp satisfies the preceding requirements. We assume the reader is familiar with view change algorithms [9, 10, 13] and focus on what is original in our approach. We note that view change algorithms are robust in the face of problems such as failures in the middle of a view change, or several nodes trying to cause a view change simultaneously. Since the implementation of the view change algorithm is not complete, the discussion is somewhat speculative.

As mentioned, group members always run within a particular *view*. A view has a unique *view number*, and later views have larger numbers. Each group member keeps its current view number on disk. When a view is formed (as discussed below), it contains at least two group members, one of which acts as primary and the other as backup. If the designated primary is a member of the view, it will act as primary in that view; otherwise the designated backup will be the primary. The designated witness will act as the backup in any view that is missing one of the two others; in this case, we say the witness has been *promoted*. A promoted witness will be *demoted* in a later view change leading to a view that contains the designated primary and backup.

A witness takes part in normal processing of file operations while it is promoted. A promoted witness appears just like a backup as far as the primary of its view can tell, but it differs from a backup in two important ways:

1. Since it has no copy of the file system, it cannot apply its committed operations to the file system.
2. It never discards entries from its log.

When a witness is promoted it receives all log records that are not guaranteed to have reached the disks at both the designated primary and backup. It appends new entries to this log as the view progresses, retaining the entire log until it is demoted. Older parts of its log are stored on a nonvolatile device; we are planning to use tape drives for witness log storage in our initial system. The witness' LB is the index of the highest log record that it has written to tape; only log entries above the LB are kept in volatile memory. The witness sends its LB to the primary and advances its GLB in the usual manner, but it does not discard log entries below the GLB.

There are several points to notice about this approach. First, because of the way the witness' log is initialized when it is promoted, we can bring a node that has not suffered a media failure up to date by just restoring its log from the witness' log.

Second, every committed operation is recorded at two servers even in a view with a promoted witness. Thus we are providing stable storage [20] for committed operations. (We differ from Echo [16] here; Echo provides stable storage only in views that do not include a promoted witness.) Stable storage is most important in a view that lasts a long time, since the probability of a second failure increases with time.

Third, if a view with a promoted witness lasts a very long time, the log may become so large that keeping information in this form is no longer practical. A log can be processed to remove unneeded entries, e.g., if a file is deleted, we can remove all entries concerning that file (except the deletion) from the log, but even such processing may not be sufficient to keep the log size practical. In such a case, the best solution may be to reconfigure the system, changing the group membership, and bringing a different node into the group to take over the role of the missing member. This new member would have its file system state initialized by the view change algorithm.

Now we briefly discuss view changes. A view change selects the members of the new view and makes sure that the state of the new view reflects all committed operations from previous views. Although the situation leading to a view change may cause a view change in more than one group (e.g., the one where the failed node is the designated primary and the one where it is the designated backup), each group does a view change separately.

The designated primary and backup monitor other group members to detect changes in communication ability; a change indicates that a view change is needed. For example, a designated primary or backup might notice that it cannot communicate with its partner in the current view or that it can communicate after not being able to. The witness does not monitor the other group members, and it never starts view changes. To avoid the case of simultaneous view changes (e.g., after recovery from a power failure), we delay the designated backup so that the designated primary will be highly likely to accomplish the view change before the backup can interfere. (It's desirable to avoid simultaneous view changes since they slow down the failover.)

User operations are not serviced during a view change, so we want view changes to be fast. A view change will be slow if some group member needs to receive lots of information in order to get up to date. To avoid this potential delay, nodes try to get up to date before a view change. A designated primary or backup that has just recovered from a crash, or has just noticed that it is able to communicate with other group members after not being able to, brings

itself up to date by communicating with another group member. It communicates with the witness if its disk information is intact. Otherwise it communicates with the designated primary or backup, which responds by sending its file system state, and then gets (a portion of) the log from the witness. Once such a group member is up to date, it initiates a view change; it is only at this point that processing in the current view is halted.

A view change is a two-phase process; the node that starts it acts as the *coordinator*. In phase one the coordinator communicates with the other group members. If a member agrees to form the new view (it will always agree unless another view change is in progress), it stops processing operations and sends the coordinator whatever state the coordinator does not already have. Usually, just a few records from the top of the log will be sent (because the coordinator is already up to date).

The coordinator waits for messages from other group members indicating that they agree to form the new view, and then attempts to form the initial state for the new view. This attempt will succeed provided the state at the end of the previous view is intact; the initial state of the new view will be the final state of the previous view. Since the previous view started from the final state of the view before it, and so on, we can be sure that all effects of operations that committed in earlier views will be known in the new view.

If the coordinator succeeds in forming the new state, it enters phase 2 of the view change protocol. It writes the new view number to disk and informs the nodes that responded in phase 1 about any parts of the initial state of the new view that they do not already know. If both other nodes responded, the witness will be demoted; a demoted witness discards its log and its volatile information. If only the witness responded, it will be promoted. The other nodes write the new view number to disk when they receive the phase 2 message.

Now let's look at how Harp satisfies its requirements. Within a single view, correctness is guaranteed because operations are applied in commit order. As we have already shown, a new view is formed only if its state will reflect all committed operations from previous views. To complete the correctness argument, we must show that Harp preserves correctness in the following two cases. First, application of an event record must have the same effect at both the primary and the backup (since the backup may become the primary in a later view). Second, some event records may be applied after a view change even though they were already applied (and their effects were already reflected on disk) in an earlier view. Such reapplication must be consistent with applying the sequence of committed operations exactly once. Section 4.4 explains the information in events records guarantees correctness for these two cases.

We satisfy the availability requirement because Harp will form a new view whenever it is legal to do so. We satisfy the reliability requirement because Harp will not lose the effects of committed operations in the face of any single failure, nor when there is a power failure (because of the UPS's). Harp also survives some simultaneous double failures. For example, both the designated primary and backup can lose their volatile memory provided the witness is promoted and its state is intact. Simultaneous failures are unlikely, except possibly for a software bug that causes both the primary and the backup to crash. We guard against the effects of a "killer packet" that causes all nodes that receive it to crash by modifying Unix so that a portion of volatile memory survives a soft crash. We keep our system state in this part of volatile memory, and use it to restart without loss of information after such a crash. We also use the following "conservative" approach to make other simultaneous software errors unlikely. A backup only applies an event record after it has already been applied successfully at the primary. To accomplish this, the primary sends its AP in messages to the backup, and we preserve the following invariant:

$$AP_{backup} \leq AP_{primary}$$

Delaying application at the backup means that an error in our code that causes the primary to crash is unlikely to show up at the backup until after a view change. At this point, we at least have the log at the witness, so event records for committed operations will not be lost; in addition, the log should be useful to the programmer who analyzes and corrects the error.

Now let's look at the timeliness requirement in the two cases of special interest: a single failure, and recovery from a single failure. If the designated primary or backup fails, the remaining one becomes the coordinator and the witness is promoted. The information sent to the witness in phase 2 is the log beyond the GLB. Thus we require two disk writes (to record the new view number) and two message round trips. The phase 1 messages are small. The size of the phase 2 message depends on the size of the part of the log beyond the GLB; as mentioned, this is likely to be large. We can avoid sending a big log to the witness in phase 2 by keeping the witness as a "warm" standby. The primary would send log records to the witness as well as the backup, but the witness would not acknowledgment these messages and would simply discard records from the bottom of its log when its log exceeded the maximum size. A "standby" witness does not write anything to tape.

Now suppose the designated primary or backup recovers from a failure. First it will bring itself up to date by communicating with another group member. If there has just been one failure, this group member will be accessible and a member of an active view. If the recovering node has not had a media failure, it will communicate with the witness; this is likely to have little, if any, impact on the speed of processing operations. The witness would keep new log records on a different tape, or on disk, while it is reading the old records from tape. If the recovering node has had a media failure, it must communicate with the primary of the current view, and then with the witness. The primary will be able to continue processing operations while it sends the file system state. Thus users continue to receive service, but probably with a somewhat degraded response time.

After the recovering node is up to date, it does the view change, but the view change should be short, since we have small messages. The protocol requires two round trips if the coordinator is the designated primary, and otherwise it requires only one and one half round trips.

Thus we argue that there is fast failover in these cases because there is little work to be done during the view change. In the case of a view change to mask a failure, failover time also includes the time needed for a node to notice that it has lost contact with its partner. This time can be decreased by more frequent monitoring of communication ability; we conjecture that the time required to notice the need for the view change will be the dominant factor in determining the speed of the failover in this case. Whether the failover can be done before the user notices depends on how quickly the client code times out a call; a failover must occur within this interval or it will be visible to the user. Thus we can see that our timeout interval is related to that of a higher layer of the system. NFS allows its users to control this higher-level timeout; our plan is to set it to mask failovers, provided we can get them to be reasonably fast (a few seconds).

There are a number of (unlikely) failure situations that can lead to loss of information (e.g., media failures at both the designated primary and backup). In this case the view change protocol will fail and human intervention will be required. The system administrator will need to determine how to set the system state. Possible choices are: the disk state of whichever of the designated primary or backup was active most recently, the most recent file system dump, or, if the entire log is kept on tape, it could be replayed. These choices all have their problems: at the very least, effects of recently committed operations will be lost. If recovery is done from the disk state of a designated primary or backup and that node had crashed, its disk must be scavenged, and the result may be missing the effects of some committed operations even though it includes effects of operations that committed later; also some operations (e.g., a rename) may be only partially reflected. (As mentioned earlier, we do not normally scavenge the disk after a crash but instead replay the log to restore the disk to a consistent state, as is done in log-based file systems [15, 18, 30].)

If there has been a partition, some users may be unable to access their files even though they can access the designated primary or backup; the server cannot respond to operations because it cannot communicate with other group members. The system could be configured to allow such a disconnected server to carry out non-modification operations (including file reads -- the "time last accessed" would not be updated) although the information read in this way might not reflect the effects of recent modifications. The disconnected server should not carry out modification operations, however, since this may cause the file system state to diverge.

4.4. Event Records

The apply process uses the information in an event record to perform the described operation. Events are applied at both the primary and the backup, and in the presence of failures, events can be applied more than once. Nevertheless, it must not be possible to detect multiple applications of event records when the file system state is accessed via calls to Harp.

As an example of the kind of problem that arises, consider the following. Suppose user A attempts to write a file to which he or she has no write access. Later, user B grants A write access to the file. If we log an event record for the write and a later one for the grant, there is no problem if there is no crash: when the write event is applied, it will fail. However, if there is a crash things may not work properly. The problem is that the grant event may have been applied and its effects recorded on disk before the crash. However, when the crashed node recovers (and goes through a view change), the write event may still be in the log. Therefore, when the log is applied after the view change, the write would happen since it appears to be permitted.

We use a simple strategy to avoid situations like this: we add an event to the log only when we know the "outcome", and the entry in the log completely describes this outcome. For example, we do not add an entry for a write operation to the log until we know that the operation can be performed. (If the operation cannot be performed, we do not need to add any record to the log.)

A second example concerns directory operations. Suppose we start with an empty directory D and perform the following operations:

create B in directory D
create A in directory D
remove A from directory D
create A in directory D
remove B from directory D

This sequence will leave the directory entry for A in the second directory "slot." In Unix, the location of directory entries is visible to users, since directory pages can be read. Now suppose all five operations were applied to the file system and their effects reached disk before a crash. Furthermore assume that after the crashed member recovers, the resulting restored log has just the last three event records. If we treat event records naively, we might end up with A's entry in the first directory slot after recovery, which would be incorrect. Again we avoid the problem by pre-computing the outcome before writing the event record to the log.

The pre-computation of an operation's outcome requires access to the system state that affects that operation. If the operation concerns a part of the system that has not been modified recently, we can obtain the needed information by making calls on the Unix file system; otherwise we need to consult the log. Any entry with index higher than the AP is a "recent" entry (since its effects are not yet reflected in the Unix file system).

The system state that must be consulted is the inode for a file operation and the inode and directory pages for a directory operation. Our approach is to maintain "shadow copies" of this information. The shadows are stored in event records; a particular event record stores the shadows for the inodes and directory pages that will be modified by the record's operation and we have an efficient mechanism for finding the most recent shadow for an inode or a directory page. An inode shadow contains information such as "time last modified", "time last accessed", and protection information. For example, for a file write, the inode shadow in the event record will contain the "time last modified" for the write. A directory page shadow is simply a copy of that directory page as it will appear after executing the operation. Since some operations affect several different shadowed objects (e.g., directory operations), some event records contain several shadows.

Although we tried to make no changes to the low-level Unix file system operations, some were needed to support pre-computing of outcomes. For example, we need the ability to choose the "time last modified" for a file write rather than having Unix choose it for us.

4.5. Higher Level Issues

To insert Harp seamlessly into a system, we need to consider some additional issues. A system implemented using Harp must appear to users to be identical to a non-replicated system except that failures are much less frequent. To accomplish this goal, most view changes should be completely invisible to users. Making view changes invisible requires fast automatic switching of a client to send calls to a new primary after a view change. In addition, when calls are redirected invisibly, we need to ensure that they have exactly the same effect as they would in a single server system when the server doesn't crash.

One possible mechanism for switching to a new primary is the following. Our environment supports the IP multicast mechanism [8], which allows a message to be sent to several nodes, and furthermore does this efficiently (e.g., so that a node that is not currently interested in a message does not actually receive it). If we use this mechanism, we would assign one multicast address to each replica group. Both the designated primary and the designated backup of the group can receive messages sent to that address, but usually only one does; messages at the other group member are discarded by the hardware. When there is a view change, the new primary arranges to start receiving messages on that address. This mechanism does not require any changes to client code. It works well in our environment, but it has three problems: it may not scale (there may not be enough multicast addresses available for use by our system), it is not supported in some environments, and it doesn't support reconfiguration very well (e.g., a reconfiguration in which a file system is moved from one replica group to another).

An alternative mechanism that does not have these problems is to insert some code inside the NFS client code. This code caches the identity of the current primary for a group and switches to the new primary when there is a view change. This approach has the problem that each implementation of the NFS client code would have to be changed.

When a view change happens invisibly, we need to ensure that redirected operations are executed properly. A problem arises when execution of the operation depends on server state, since such state must then survive the view change. To solve this problem, we need to access server state. Then we can replicate this state (by including more information in event records) and use it after a view change to initialize the server.

In NFS, this problem shows up only for duplicate messages. The NFS client code assigns each user operation a unique id (this is the Sun RPC id [36]). The server keeps track of the ids for recent non-idempotent operations (e.g., file creates) in a table. Each id is stored with the response for the call; if a duplicate message arrives, NFS sends back the stored response. We need to read any new information in this table for each VFS operation call and store it in the event record for the operation. At the beginning of a new view, the primary uses the information to initialize the server state to contain the proper table.[3]

We will not be able to suppress duplicates if our failover takes longer than the timeout period used at the client, since in this case the user may retry the call, and it will have a different unique id. Having such duplicates does not compromise the correctness of the system, since in this case the user knows the system has failed and recovered, and duplicates across recovery from a failure are possible in an unreplicated system. However, we do not want to increase the duplicate problem by having failovers be too slow.

5. Status and Performance

This section describes the current state of the implementation of Harp and gives some initial performance data. It also describes our future plans.

At present servers and clients are MicroVax 3500's; servers have RA70 disks. The nodes run Unix 4.3BSD (with University of Wisconsin and Sun Microsystems modifications to support VFS and NFS); we have made some small changes to Unix. The nodes are connected by a ten megabit ethernet; a kernel to kernel roundtrip for a small (one packet) message takes around five milliseconds.

The Harp implementation is being done in C. Only the normal case code has been implemented so far; we are working on view changes now. The implementation runs at kernel level and makes use of several processes. At present we are using four processes within a primary or backup. One of these is the apply process; others handle communication and garbage collection of the log.

To understand the performance of Harp in the absence of

[3]This approach violates the abstraction barrier between our system and the NFS server code. A cleaner approach, e.g., call backs, would be desirable.

failures, we ran a number of experiments that compare the performance of Harp with that of an unreplicated NFS server that uses Unix files directly. The data from these experiments are given below. All Harp experiments were run in a system containing two servers. The unreplicated NFS experiments were run on a single server identical to the one used in the Harp experiments. Experiments were run at a time when network traffic was light. File reads were run using the early reply option, i.e., read events are logged but the result is returned before the event is sent to the backup. The bound on the log size was set at two megabytes; we forced Unix to write its buffers when the log was 65% full, and stopped forcing when the log was 55% full. The maximum size of the log observed in our experiments was around one and a half megabytes.

Figure 5-1 shows the results of running the Andrew benchmark [17]. The Andrew benchmark is a widely used synthetic benchmark. It performs a fixed set of operations intended to be a representative sample of the kind of actions an average user of the Andrew file system might perform, and measures the total elapsed time, which gives an indication of the file system response time. The figure compares the performance of an unreplicated NFS server on this benchmark with the performance of Harp. Two figures are given for Harp, one for the case where there is just one replica group, and the second for the case where there are two groups. In the two-group case, each server acted as the primary for one group and the backup for the other; a second client was also running the benchmark using the second group.

Time (minutes:seconds)	
Harp (one group)	6:29
Harp (two groups)	6:49
Unix (unreplicated)	6:54

Figure 5-1: The Andrew Benchmark

As can be seen, Harp runs a little more slowly in the two-group case than in the one-group case because of interference at a server between the primary from one group and the backup from the other. Harp performs slightly better than the unreplicated server; this is because we replace a disk write with a message roundtrip, which is faster in our environment. It is interesting to note that even though caching at the NFS client largely masks the actual server response time in this experiment, we are still able to improve upon the unreplicated system results.

To get an indication of server performance independent of the effects of client caching, we ran the Nhfsstone benchmark [25, 34]. Nhfsstone is a standard synthetic benchmark designed to measure the performance of NFS file servers. It simulates a multi-client NFS file server workload in terms of the mix of NFS operations and the rate at which the NFS clients make requests of the server. The benchmark generates the specified operation load and operation mix and then measures the average operation response time and the load experienced by the server. We looked at two mixes: a read-only mix, and the so-called software development mix [25]. These mixes represent environmental extremes, since the software development mix assumes 50% diskless workstations, and therefore contains a large fraction (15%) of writes. (In both mixes, file reads are (also) modification operations since they modify the "time last accessed".) We gradually increased the server operation processing rate until the server saturated, i.e., its time to respond to operations got dramatically worse. This gives us an indication of the operation processing capacity of the server which in turn is indicative of the number of users the system can support.

Figure 5-2 show results obtained by running Nhfsstone; in this experiment Harp was running a single replica group. In the figure, the horizontal axis shows the average server operation processing rate in operations per second and the vertical axis shows the average response time for operations at that processing rate and operation mix. It is clear from the figure that the performance of the two systems is comparable on the read-only mix, but Harp outperforms the unreplicated NFS on the software development mix because of the large number of writes in that mix. Harp's response time is lower and, in addition, it saturates at a higher load and therefore has a larger operation processing capacity than the unreplicated system. Of course, when configured like this Harp requires considerably more hardware than the unreplicated system. Such a system might nevertheless be of interest because it provides high reliability and availability at moderate cost. (The witness in such a system can be at a third node that is small and cheap.)

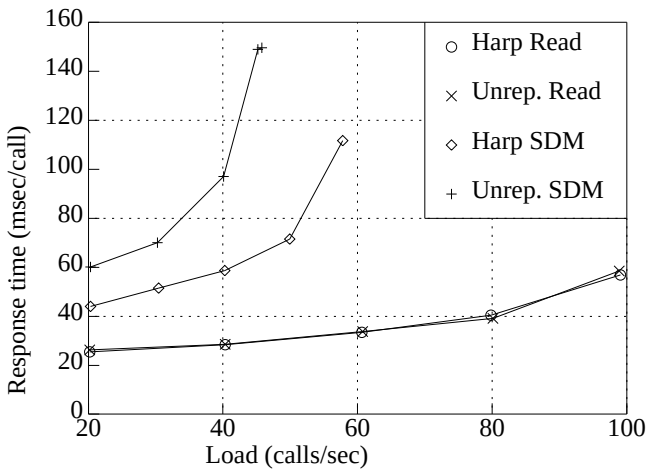

Figure 5-2: Nhfsstone Benchmark with One Group. SDM is the Software Development Mix.

However, we expect many users will want to get more processing out of the servers. Our final experiment approximates this case. In this experiment we run two simul-

taneous Nhfsstone benchmarks. For Harp, each benchmark used a different group; again, each server acted as the primary for one group and the backup for the other group. For the unreplicated system, each benchmark used a different server. The results of running these Nhfsstone experiments are shown in Figure 5-3. The two systems are again comparable on the read-only mix. On the software development mix, Harp still outperforms the unreplicated NFS, although it does not perform as well as it did in the single-group experiment. The figure shows the performance measured by one of the Nhfsstone benchmarks (the other one had the same results); since there are two benchmarks running, the two-group system is actually handling twice the number of operations shown.

The Nhfsstone benchmark generates a constant load and therefore does not model the bursty nature of the actual load on a file service. We have not yet measured Harp under such a load, but we expect that Harp's log, which acts as an asynchronous buffer between the client and the server, will smooth out the bursts in the actual client load, as is the case in other systems [24]. This will give Harp a further advantage over the unreplicated server.

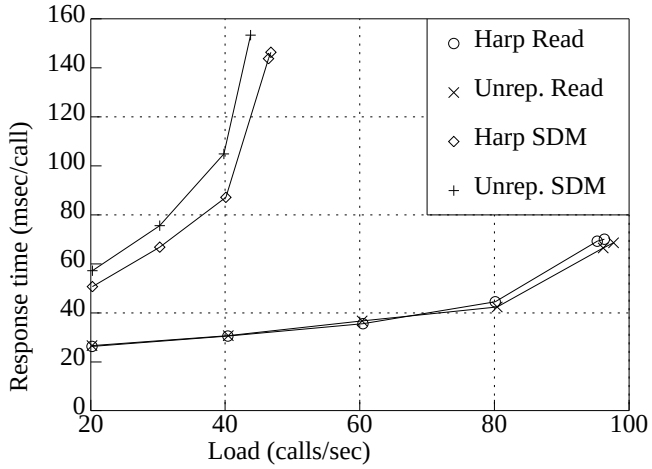

Figure 5-3: Nhfsstone Benchmark with Two Groups. SDM is the Software Development Mix.

As we add the view change code to the system, it will be interesting to see how fast we can do failovers, and what effect, if any, the added code has on normal case performance. The biggest problem will be maintaining warm witnesses; we expect this to cause little degradation in performance because the primary can use a multicast to send its messages to both the witness and the backup, and there will be no acknowledgments from the witness.

Once the implementation is complete, we plan to undertake a careful study of its performance, which depends on a complex interaction of a number of parameters (e.g., disk speed, number of disks, message speed, log size). Ultimately, we plan to move Harp onto DECstation 5000's connected by a 100 megabit FDDI network and put it into active service.

6. Conclusion

This paper has described the design and implementation of the Harp file system, a replicated Unix file system that is intended to be used within a network file service. Harp is one of the first implementations of the primary copy replication technique that runs on conventional hardware. The current implementation guarantees that all completed user operations are performed atomically and survive single failures and likely simultaneous failures. Furthermore, user files continue to be accessible when a single node fails, and failover is fast in common cases.

Harp's replication algorithm incorporates several novel ideas that allow it to provide good performance. Most notable is its use of a volatile write-ahead log backed up by UPS's, which allows it to trade disk accesses for message passing. Note that we can expect our performance to improve relative to an unreplicated system because networks and processors are getting faster more rapidly than disks.

Although Harp is used by NFS in our current implementation, it is relatively independent of the particular file service that uses it. It implements the VFS interface; any network file service that uses this interface should be able to use Harp to provide better reliability and availability. However, Harp is not totally free of dependencies of the service that uses it. As mentioned in Section 4.5, Harp needs access to request ids so that it can ensure duplicate suppression across a failover that happens invisibly to system users. Also, the timeout used in Harp to detect the need for a view change is related to the timeout for client calls; our goal is to accomplish failovers within the client call timeout period (setting the client timeout appropriately).

As distributed systems become more and more prevalent, there is a growing dependence on file servers. However existing file servers are unsatisfactory because they fail too often. Harp corrects this problem. A file server can be switched to use Harp without file system users noticing the difference, except in a positive way. Once Harp is in place, the file system will fail much less often than it did before, and even when it fails, it will usually do so more gracefully than an unreplicated system because operations are performed atomically (in the absence of a catastrophe). Furthermore, Harp does all this while providing good performance. Our performance results so far indicate that we can perform as well or better than an unreplicated NFS server that uses the Unix file system directly. If these results hold up in the complete implementation, as we expect them to do, it means that high availability and good semantics can be achieved with just a relatively small amount of additional hardware (the UPS's and the extra disks to hold extra file copies).

7. Acknowledgments

The authors gratefully acknowledge the helpful comments of Dorothy Curtis, David Gifford, John Guttag, Karen Sollins, John Wroclawski, and the referees.

8. References

1. Alsberg, P., and Day, J. A Principle for Resilient Sharing of Distributed Resources. Proc. of the 2nd International Conference on Software Engineering, October, 1976, pp. 627-644. Also available in unpublished form as CAC Document number 202 Center for Advanced Computation University of Illinois, Urbana-Champaign, Illinois 61801 by Alsberg, Benford, Day, and Grapa..

2. Bartlett, J. A 'NonStop' Operating System. Proc. of the Eleventh Hawaii International Conference on System Sciences, January, 1978, pp. 103-117.

3. Bartlett, J. A NonStop Kernel. Proc. of the 8th ACM Symposium on Operating System Principles, December, 1981, pp. 22-29.

4. Bernstein, P. A., Hadzilacos, V., and Goodman, N.. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

5. Bhide, A., Elnozahy, E., and Morgan, S. Implicit Replication in a Network File Server. Proc. of the Workshop on Management of Replicated Data, IEEE, Houston, TX, November, 1990.

6. Birman, K., and Joseph, T. Exploiting Virtual Synchrony in Distributed Systems. Proc. of the Eleventh ACM Symposium on Operating Systems Principles, November, 1987, pp. 123-138.

7. Daniels, D. S., Spector, A. Z., and Thompson, D. S. Distributed Logging for Transaction Processing. ACM Special Interest Group on Management of Data 1987 Annual Conference, ACM SIGMOD, San Francisco, CA, May, 1987, pp. 82-96.

8. Deering S. E., and Cheriton D. R. "Multicast Routing in Datagram Internetworks and Extended LANs". *ACM Trans. on Computer Systems 8*, 2 (May 1990).

9. El-Abbadi, A., and Toueg, S. Maintaining Availability in Partitioned Replicated Databases. Proc. of the Fifth Symposium on Principles of Database Systems, ACM, 1986, pp. 240-251.

10. El-Abbadi, A., Skeen, D., and Cristian, F. An Efficient Fault-tolerant Protocol for Replicated Data Management. Proc. of the Fourth Symposium on Principles of Database Systems, ACM, 1985, pp. 215-229.

11. Ghemawat, S. Automatic Replication for Highly Available Services. Technical Report MIT/LCS/TR-473, MIT Laboratory for Computer Science, Cambridge, MA, 1990.

12. Gifford, D. K. Weighted Voting for Replicated Data. Proc. of the Seventh Symposium on Operating Systems Principles, ACM SIGOPS, Pacific Grove, CA, December, 1979, pp. 150-162.

13. Gifford, D.K. Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox Corporation, March, 1983.

14. Guy, R., Heidemann, J., Mak, W., Page, T., Jr., Popek, G., and Rothneier, D. Implementation of the Ficus Replicated File System. USENIX Conference Proceedings, June, 1990, pp. 63-71.

15. Hagmann, R. Reimplementing the Cedar File System Using Logging and Group Commit. Proc. of the Eleventh Symposium on Operating Systems Principles, November, 1987, pp. 155-162.

16. Hisgen, A., Birrell, A., Jerian, C., Mann, T., Schroeder, M., and Swart, G. Granularity and Semantic Level of Replication in the Echo Distributed File System. Proc. of the Workshop on Management of Replicated Data, IEEE, Houston, TX, November, 1990.

17. Howard, J., Kazar, M., Menees , S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. "Scale and Performance in a Distributed File System". *ACM Trans. on Computer Systems 6*, 1 (February 1988).

18. Kazar, M., Leverett, B., Anderson, O., Apostolides, V., Bottos, B., Chutani, S., Everhart, C., Mason, W., Tu, S., and Zayas, E. Decorum File System Architectural Overview. USENIX Summer '90 Conference Proceedings, 1990, pp. 151-163.

19. Kleiman, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. USENIX Summer '86 Conference Proceedings, 1986, pp. 238-247.

20. Lampson, B. W., and Sturgis, H. E. Crash Recovery in a Distributed Data Storage System. Xerox Research Center, Palo Alto, CA, 1979.

21. Mann, T., Hisgen, A., and Swart, G. An Algorithm for Data Replication. Report 46, DEC Systems Research Center, Palo Alto, CA, June, 1989.

22. Marzullo, K., and Schmuck, F. Supplying High Availability with a Standard Network File System. Proc. of the 8th International Conference on Distributed Computing Systems, IEEE, June, 1988, pp. 447-453.

23. Mills, D.L. Network Time Protocol (Version 1) Specification and Implementation. DARPA-Internet Report RFC 1059. July, 1988.

24. Nelson, M., Welch, B., and Ousterhout, J. "Caching in the Sprite Network File System". *ACM Trans. on Computer Systems 6*, 1 (February 1988), 134-154.

25. Legato Systems Inc. NFSSTONE NFS Load Generating Program. Palo Alto, CA.

26. Oki, B. M., and Liskov, B. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. Proc. of the 7th ACM Symposium on Principles of Distributed Computing, ACM, August, 1988.

27. Oki, B. M. Viewstamped Replication for Highly Available Distributed Systems. Technical Report MIT/LCS/TR-423, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

28. Paris, J-F. Voting With Witnesses: A Consistency Scheme for Replicated Files. Proc. of the 6th International Conference on Distributed Computer Systems, IEEE, 1986, pp. 606-612.

29. Rodriguez, R., Koehler, M., and Hyde, R. The Generic File System. USENIX Summer '86 Conference Proceedings, 1986, pp. 260-269.

30. Rosenblum, M., and Ousterhout, J. The Design and Implementation of a Log-Strucured File System. To be published in the Proc. of the Thirteenth ACM Symposium on Operating Systems Principles, October 1991.

31. Sandberg, R., et al. Design and Implementation of the Sun Network Filesystem. Proc. of the Summer 1985 USENIX Conference, June, 1985, pp. 119-130.

32. Satyanarayanan, M., et al. Coda: A Highly Available File System for a Distributed Workstation Environment. Tech. Rept. CMU-CS-89-165, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, July, 1989.

33. Schneider, F. B. Fail-Stop Processors. Digest of Papers from Spring CompCon '83 26th IEEE Computer Society International Conference, IEEE, March, 1983, pp. 66-70.

34. Shein, B., et al. NFSSTONE- A Network File Server Performance Benchmark. USENIX Summer '89 Conference Proceedings, 1989, pp. 269-274.

35. Sun Microsystems, Inc. NFS: Network File System Protocol Specification. Tech. Rept. RFC 1094, Network Information Center, SRI International, March, 1989.

36. Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification, Version 2. DARPA-Internet RFC 1057. June, 1988.

37. Walker, B., Popek, G., English, R., Kline, C., Thiel, G. The LOCUS Distributed Operating System. Proc. of the 9th Symposium on Operating Systems Principles, ACM, Bretton Woods, NH, October, 1983.