ACHIEVING STRONG CONSISTENCY IN A REPLICATED FILE SYSTEM

by

Carl Neilson

B.Sc. Simon Fraser University 1990

THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

APPROVAL

NAME:	Carl Neilson
DEGREE:	Master of Science
TITLE OF THESIS:	Achieving Strong Consistency In A Replicated File System

Examing Committee:

Chair: Dr. Warren Burton

Dr. Tiko Kameda
Senior Supervisor
Professor Of Computer Science

Dr. Peter Triantafillou
Assistant Professor Of Computer Science

Dr. Stella Atkins
Internal External Examiner
Associate Professor Of Computing Science

Date Approved: August 5, 1993

ACHIEVING STRONG CONSISTENCY IN A REPLICATED FILE SYSTEM

Abstract

Replicated file systems help to provide fault tolerance in distributed computing environments. Many existing replicated file systems sacrifice strong semantics to achieve efficiency and have costly failure handling and recovery algorithms. This thesis puts forward a replicated file system protocol that enforces strong consistency semantics while achieving more efficient failure handling and recovery. Although the protocol pays a performance cost in order to ensure the stricter semantics, this cost is reduced through the use of a non-centralized protocol in which all of the replicas are peers. This decentralization of the protocol avoids the bottleneck problem noticed in primary-copy systems, facilitates load balancing, lets clients choose physically close servers and allows for the reduction of work required during failure handling and recovery. Instead of optimizing each operation type on its own, file system activity was viewed on the level of a file session and the costs of individual operations were able to be spread over the life of a file session.

The performance of a prototype of the protocol is compared to both NFS and a non-replicated version of the protocol that also achieves strong consistency semantics. Through these comparisons the cost of replication and the cost of enforcing the strong consistency semantics are shown. Qualitative comparisons of this protocol to other replicated protocols are also provided.

Acknowledgments

I would like to thank my supervisors Peter Triantafillou and Tiko Kameda for their assistance in researching and writing my thesis. I would also like to extend my appreciation to M. Satyanarayanan and the School of Computer Science at Carnegie Mellon University for their help in acquiring the invaluable file system traces.

ACHEIVING STRONG CONSISTENCY
IN A
REPLICATED FILE SYSTEM

Table Of Contents

List of Tables

List Of Figures

1 Introduction

This thesis proposes a protocol for a replicated file system that differs significantly from protocols described in recent research. The protocol achieves UNIX semantics unlike other replicated file systems that offer weaker semantics. Also, this protocol is fully distributed in that all of the servers are equal peers.

The remainder of Chapter 1 shall cover definitions and topics relevant to the protocol. Chapter 2 will survey other relevant file systems, describing the basic protocols and the achieved semantics. A general overview of our protocol is given in Chapter 3 with a more detailed explanation of the algorithms in Chapter 4 and the failure handling being explained in Chapter 5. A qualitative comparison of the protocol to existing replicated file systems is presented in Chapter 6. Chapter 7 will discuss the prototype we implemented, the methods we used to test the protocol, and the results of the tests. Concluding remarks and issues for future research directions are found in Chapter 8.

1.1 Replicated File Systems

In a replicated file system files are stored redundantly, with their copies distributed among a number of servers. The motivation for replication is to increase the availability of files during periods of failures, and/or to increase the performance of the file system in a distributed computing environment.

Replicating files on multiple servers allows files to be accessed when one or more file servers are inaccessible. However, in using multiple file copies, consistency problems are introduced. The desired semantics of the file system must be adhered to, when concurrent accesses are being performed, and when only a subset of the servers is unavailable. For example, if the file system is to provide **UNIX semantics** which dictates that all writes be immediately visible to all **clients**[1], unavailable servers can be left out-of-date during periods of updates. If multiple conflicting updates arrive at different servers with copies of the same file, inconsistencies will result if the updates are not applied in the same order at all of the servers. In order to maintain file consistency, the file system must control when and at what servers file accesses can be performed.

As well as tolerating failures, a replicated file system can be used to improve performance over widely distributed file systems. Rather than sending requests across the network to a distant server, a physically close server can serve requests, thus reducing the cost of accessing data. Unfortunately, not all replicated file systems make full use of this feature.

1.2 Failures

There are different types of failures that can affect a replicated file system. A server's disk can fail, a server can fail, communication links can be broken, and clients can fail. Each type of failure can affect the replication protocol differently. How a site fails can also effect the file system.

Usually servers failures are assumed to be **fail-stop** in that the server does not show any degradation before the failure. The server will always perform its functions correctly and will not generate invalid messages. When a site undergoes a **Byzantine** failure, the site acts "crazy" or in a incorrect manner possibly producing illegal or incorrect messages, or destroying memory and/or disk contents.

When a server's disk fails, the other servers can be made aware that the server will be out of service and can change their behavior accordingly. The other servers can assume one less server, or can choose another site to replace the ineffective server. Detectable disk crashes are easily handled in a replicated environment (as long as some other up-to-date copy of each file exists.)

1. A client is the process or site that requests a file system operation.

Failure of a machine (server or client) cannot be distinguished from the failure of a communication link, or from slow responses due to extreme overloading. Therefore, when a site does not respond one cannot determine if the site has failed and stopped processing, or if a communication link is down and the site is still operational. One must then assume that the unavailable site is still capable of processing file requests. Requests that could conflict with possible requests at the inaccessible site should be disallowed so that the consistency guarantees of the file system are not violated.

Partitioning caused by communication link or site failures must also be considered. If strong consistency semantics are to be offered by the file system, not all partitions can be allowed to serve accesses on a given file. Some methods to determine in which partition file operations can continue include using tokens, requiring that a majority or specific sites be included in the partitions (i.e., the partition with site A is allowed to continue processing, all others are not), etc.

Client failure can also affect whether a server can process requests on a file. If a client has a file cached, but the client does not respond to a server, the server cannot know if the client is accessing the cached file or not. In order to achieve the desired consistency semantics, the server may have to either assume that the cache is being used by the client or the protocol must provide a mechanism to prohibit the client from continuing when partitioned. Which accesses the server may allow on the file by other clients will be determined by the consistency guarantees of the file system.

1.3 Definition Of Terms

1.3.1 File Session

A file session is a sequence of system calls by a process starting with the opening of a file for access, and ending with the corresponding close. All read, write, seek, and truncate operations between the open/close pair, that reference the same file belong to the file session. No read, write, seek, or truncate system calls can be performed outside of a file session

```
file session = open [read | write | seek | truncate]* close
```

Figure 1-1: Definition of a file session.

1.3.2 Write-Sharing

Write-sharing refers to concurrent file sessions that access a file in conflicting modes. Two, not necessarily concurrent, file sessions conflict if at least one of them allows writing.

1.3.3 Availability

The term **availability** refers to the accessibility of a file. If a client requests access to a file that the client has the right to access and is unable to access the file, then the file is said to be unavailable. The fraction of time that legal operations can be performed on a file by a client is the availability of the file. Failures and/or restrictions imposed by a file system, can make a file available to some clients and not available to others; in this manner a file that n clients have access to has a lower availability than a file that m clients are able to access when $m > n$.

Failures reduce the availability of a file by making it impossible for clients' file requests to be granted. In a system free of failures, a client should always be allowed access to a file unless the file is locked or the client does not have the required privileges. This paper does not consider the lack of availability due to locking performed by an application or due to a user's lack of privileges. Availability is then a measure of a file system's ability to serve requests when failures are present.

1.3.4 Serializability

A concurrent execution of a set of actions, performed on a set of objects, is **serializable** if it is equivalent to a serial execution of the set of actions performed on the set of objects. Each action is a called **a serialization unit** and is a collection of operations, of which each may be executed concurrently, and/or inter-mixed with operations from the same or other serialization units. The ordering of the operations within a serialization unit must be the same in both the concurrent execution, and the serial execution. In the case of file systems, the serialization unit is usually a single file system call.

An execution performed on a replicated set of objects is **one-copy equivalent** if the execution is equivalent to an execution performed on a non-replicated set of the same objects. An execution on a replicated set of objects is **one-copy serializable** (1SR) if it is serializable and one-copy equivalent. (For a more formal definition refer to [BERN87]).

In file systems, it is also desirable that the execution of a set of system calls performed by a single process be equivalent to a serial schedule which arranges them in the same order in which they are performed. Since the serialization unit in a file system is typically a system call, 1SR does not guarantee this. An execution of a set of serialization units that are performed by a group of processes is **global one-copy serializable** (G1SR) if the execution is 1SR and the partial ordering of the serialization units observed by each process is maintained.

So far, we have discussed desirable properties of a given execution *E*. As a separate issue, there is a question of how accurately *E* reflects the actual order the serialization units were executed in. An execution, *E*, is **real-time consistent** if for any two operations, *op1* and *op2* in *E*, *op1* precedes *op2* in *E*, iff *op1* occurred (in real time) before op2 [SIEG89].

1.4 File Semantics

A principal issue when designing a file system is its behavior in the presence of:

1) concurrent conflicting requests and

2) failures.

This issue is identified as the **file semantics** issue. Four types of commonly used file semantics are: **UNIX semantics**, session semantics, **immutable files** and **transactional semantics**.

1.4.1 UNIX Semantics

When one uses basic file operations as the units to be serialized, and these operations are constrained by real-time consistency, UNIX **semantics** result. Under UNIX semantics update operations are "immediately" visible (i.e. within d seconds, where d is small) to all read operations that follow and are always applied to a copy of the file that reflects all previous updates. The strictness of UNIX semantics can hinder the efficiency of distributed file systems due to the overhead in guaranteeing all updates on a file are reflected in all copies of the file (i.e., cached copies, and server replicas), and in guaranteeing that the updates are applied in the same order at all servers.

1.4.2 Session Semantics

Under session semantics, each file session gets a logical copy of the current version of the file. The current version of a file reflects the updates from closed file sessions, and none of the updates from ongoing file sessions. All read and write accesses for the file session are performed on this copy. When the file session ends, the logical copy becomes the current version of the file (if the session updated the file). An update file session, S1, will have its updates completely obliterated when another updating file session S2 closes, if S1 opens after S2 starts and closes before S2 closes, even if the two sessions update non-overlapping sections of the file.

Session semantics are not always defined quite so strictly. File sessions on a single machine may or may not share a copy of the file; it depends on the interpretation by the implementor. Sharing the local copy introduces two questions, one on open and one on close. When the second session starts, would the version brought from the server destroy any updates to the cached version that were performed by the first session? When the first close is received, is the file written back to the server, or are the updates kept locally until all sessions have closed? This question has not been decisively answered in the literature.

1.4.3 Immutable Files

Similar to session semantics, each file session is given a logical copy of the file being accessed and updates to the file are only noticeable to other sessions after the close. The difference is that an update session creates a new version of the file so that both the old version and the new version are present and accessible. An open system call can request an old version so that any version of the file is accessible (if all versions are kept).

This method partially overcomes the problem of session semantics where the effects of some sessions can be obliterated. A session's effect will not be obliterated, but it may not be reflected in the most current version of the file. Immutable files also require extra disk space to store the multiple copies of each file.

1.4.4 Transactional Semantics

Transactional semantics require that a set of file sessions be serializable, and that any given file session appear to be an *atomic action*. A set of file sessions executed concurrently is serializable if the outcome of the execution is the same as the outcome of some serial execution of the same set of file sessions. Atomicity of file sessions guarantees that either all of the actions of a file session will be performed, or none of them will, and that the actions appear to be instantaneous, no transient state of the file is noticeable. Each file session can be equated to a transaction in a database environment.

1.5 Basic Methods Of Replica Control

1.5.1 Primary-Copy/Primary-Site

The primary-copy approach centralizes replica control at a single site. A server is chosen to be the primary server for a file, all other servers are secondary servers. All client requests to the file are sent to the primary server. The primary server is responsible for sending updates to the secondaries in order to keep them up-to-date. Some systems have a single primary for all files (**primary-site**), while others have a primary that controls a group of files (**primary-copy**). This group of files is called a **file group**.

A primary and the set of accessible secondaries is called a view. When the failure of the primary or a secondary is detected, a server will initiate a view change. If the network is partitioned, a view change will occur for each file group, in every partition. If the protocol requires strict consistency semantics then the view change will only succeed in one partition, usually the majority partition. If the primary is not in the successful partition, then a secondary will be elected to the position of primary. While a view change is taking place, no file requests can be served.

Such primary-copy systems pay a high cost when a server that is participating in the current view becomes inaccessible, or when a server becomes accessible. No file system requests are fulfilled while the servers confer amongst themselves to elect a new primary. The view change typically includes the transferring of the current content of files between the servers to ensure that all participating servers in the new view are up-to-date. Harp ([LISK91]) reduces this cost by allowing recovering servers to become up-to-date before the view change, if possible. Echo ([MANN89]) delays bringing the secondaries up-to-date until after the view change succeeds, but the primary is brought up-to-date during the view change. Delaying the file transfers allows clients to access the files sooner but does not reduce the strain on server cpu's or network bandwidth caused by a view change.

Different computers have different failure characteristics; some servers fail more frequently than others. Allowing primaries to be chosen from those servers that survive during failures, sites that fail less frequently are more liable to become the primary for a collection of file groups. Therefore, a primary protocol that does not take measures to balance the load during view changes may degrade to the performance of a primary-site protocol.

All clients sends their file requests to the file's primary server. If the primary is not physically close, the client must send the request to the distant primary.

Primary-copy algorithms result in systems that are easier to administer and which support efficient read operations. However, such systems cannot exploit replication to improve data access times and suffer from considerable overhead during failure and recovery.

1.5.2 Tokens

Tokens can be used for replica control instead of a primary system as in Deceit ([SIEG89], [SIEG90]), or for cache consistency control in conjunction with some other replica control protocol as in Echo ([MANN89]). When tokens are used for replica control, a server is required to obtain a token before performing accesses on a file. When tokens are used for cache consistency, it is required that a client obtain a token before accessing a cached copy of a file.

If a site fails while holding a token, the token is inaccessible until the failure is resolved. A time-out can be attached to a token so that a token is only valid before the expiration of the time-out. The drawback to using a time-out, is that any updates that are only reflected at the site that holds the expired token may conflict with updates performed by any new token holders. Another method is to issue a new token when the old token cannot be found, thus allowing multiple tokens to exist in the system at once. Generating new tokens whenever the token holder is inaccessible allows multiple sites to concurrently access different copies of a file which can violate strong consistency guarantees.

1.5.3 Voting

A more distributed approach to replication is voting ([GIFF79]). Each server is assigned a number of votes. For an operation to be performed by a server S, S must first collect responses from a sufficient number of other servers so that a majority of votes is gathered. Since any two sets of servers with a majority of votes overlap, all operations on the file are guaranteed to notice the most current value of the file.

In a replicated file system there are many ways that voting can be utilized. Voting can control server accesses on files by requiring, for example, that a majority of the servers be notified of the intent to access. This will allow the server performing an access to detect concurrent conflicting accesses, and to determine which servers store up-to-date versions of the file.

The number of servers that need to be consulted when updating or querying a file need not be confined to majorities. For example, if there are N sites, the number of sites that need to be contacted for a read

$$W + R > N$$

$$W > N / 2$$

Figure 1-2: Restrictions on number of sites to contact when updating and/or querying an information item.

operation (R) plus the number of sites needed for an update operation (W) must be larger than N. This ensures that read operations find a current version of the file. Also W must be larger than $N/2$ to guarantee that any two update operations will communicate with overlapping sets of servers, allowing serialization of the update operations.

1.5.4 Witnesses

It is likely that increasing the number of replicas will increase availability. However, this is achieved at the expense of increasing the storage costs, and increasing the overhead for maintaining consistency. Witnesses do not store copies of files; they only store some state information (i.e., version #) that is useful for processing operations on replicated files ([PARI86]).

Witnesses can also replace lost servers. When the failure of a replica is noticed, a witness can be promoted to become a replica, although this can require significant bandwidth to send the files to the witness.

1.6 Client Caches

As in all distributed systems, client sites may be allowed to cache file data. This introduces another level of file consistency that must be checked. When can a client cache file data? When can a client access cached file data? When should modified data be propagated back to the file server? How are client failures handled with respect to file availability, cache consistency and the permanence of cached updates? What unit of data is cached, a file block, a file, or some other unit?

1.6.1 Cache Consistency

Client caches are used in most distributed file systems to improve performance. When a file system is providing UNIX semantics, a client's cache can be considered just another replica of the file and therefore accesses must be constrained in order to guarantee consistency. Cache consistency can be addressed through the use of tokens, by allowing a client to become a pseudo-server, by disabling client caching during periods of write-sharing, by choosing a semantics notion that trivializes caching (session semantics), or cache consistency can simply be ignored.

Tokens can be issued to clients to allow them to perform reads and/or writes on their cached copies. This method is used by Echo. One problem with tokens is loss of availability when the token holder becomes unreachable. Echo overcomes this problem with a token time-out; once a token times out, the client is no longer guaranteed that the copy in its cache is the most current, or that any updates made to the cache will be kept when contact with the server is re-established. If a client's token times out when the cache is dirty, and another client, in a different partition, updates the file, updates in the original client's cache will not be written to the server, but will be thrown out when contact with the server is reestablished.

When a file system is to provide strong consistency guarantees, cached copies must be kept up-to-date. Updates performed by one client must be seen by other clients. This can be accomplished through call-backs where the server requests all updates to be written to the server. The new version of the file will then be passed on to subsequent clients of the file. During conflicting file sessions this can cause significant overhead. In order to cut down on the cost of constantly passing updates back to the server, and then passing them to other client's caches, the server can disable client caches. This method is used in Sprite ([LEVY90]) which forces all file operations in conflicting file sessions to be passed through to the file server.

Another method to lower the cost of update propagation has clients gathering updates from other clients ([TAIT91]). Instead of writing the updates to the server, an updating client becomes a pseudo-server and serves all requests on the file. Other clients send read and write requests to the client with the cached copy, and that client performs the operations just as a server would.

NFS effectively ignores cache consistency ([LEVY90]). Updates in NFS are not written back immediately to the server, and a server ignores any possible updates that may be in client caches. This allows there to be updates in a client's cache, when the server sends file data to a different client. NFS clients check that cached file data is as current as the server's copy on open, and whenever a cache miss occurs. Other than at these points, no cache consistency check is performed; thus NFS' semantics are indeterminate. Due to this lack of strict consistency controls, any replicated file system that is designed as a replacement for an NFS server (e.g., Harp, Deceit) also ignores the cache consistency problem.

Session semantics simplify cache consistency as a client's cache need only be current on open, and replaces any current version at the servers on close. The only consistency issue that could arise would happen if the system designer chose to delay update propagation after closing. In this case, for the server to determine if a session has closed, the server would either need to poll all clients that have a cached copy and the ability to modify it, or a client must notify the server on close but not transfer the file (a better choice). On open the server would be required to perform a call-back to retrieve the modified cache contents from the current client before continuing with the open.

1.6.2 Update Propagation

When a client performs an update to a file, at some point the update will need to be reflected in the servers' copies. This can be done in different ways and at different times. Each method affects efficiency and availability differently.

1.6.2.1 Sending Updates to the Servers

When caching a file, updates can be made on the cache, or can be written through to the server. Writing every update through to the server effectively negates any benefit of having a client use a cache for update file sessions, but does provide better permanence guarantees. Although better permanence is provided, most, if not all, systems do not use write-through due to the performance penalty of sending each write request to the server ([LEVY90]). When **delayed-write** is used, cached updates are usually kept either for a specified length of time, until after the updating file session closes, or until they are flushed to make room in the cache. Some systems keep updates in the cache even after the file session has closed so that files that are deleted soon after closing will not need to be transferred to the servers.

Sending all of a session's updates after the session closes has the benefit that only one update call to the server needs to be performed. A disadvantage to delayed-write is that updates kept in a cache in volatile memory will not survive the failure of the client. Once a file session closes, updates to the file can be written to a server immediately, or the propagation of the updates can be delayed. Studies of file access patterns in [BAKE91], and [FLOY86] have shown that about 60% of recently modified files are deleted within 5 minutes of being closed. If the client does not send updates to the server before the deletion, then network bandwidth and server resources are not used. As is the case with any caching of updates, the longer the updates are only reflected in a client's cache, the lower the achievable availability and the higher the probability that updates will be lost due to client failure (if the cache is not on disk.)

If a client fails while the most current updates are still only in its cache, those updates will not be seen until the client recovers and will only survive the failure if the client's cache is stable. UNIX semantics would require that no other accesses be performed on the file until the client recovers and, if possible, sends the updates to the servers so that they become visible.

An additional drawback is the uncertainty of data written to a file during a file session. If the updates are not written back to a server on close and the client's cache is not stable, failure of the client would mean loss of the updates. The implication is that one cannot be guaranteed the permanence of changes to a file even after the file has been closed!

1.6.2.2 Immediate vs. Lazy Update

Once the updates have been sent to a server, they can either be propagated to all servers at the same time ensuring replication of the current copy of the file, to a portion of the servers, or to no servers. **Lazy update** refers to propagating updates at a later point, either when the updates are needed, when the servers are not busy, or after some time-out period. By updating all of the servers immediately, the availability of the current version of the file is enhanced. The more servers with up-to-date copies of the file, the more server failures that can be tolerated when the file system provides UNIX semantics.

Delaying propagation can be beneficial in a number of ways. Transmitting updates only when they are needed at a different server causes network usage only when required but significantly reduces availability. Waiting until server load decreases before propagating updates can balance the load on the servers and improve response times. Time-outs can be used to ensure that updates will be propagated at some point so that servers will receive the current version eventually. Delaying updates can also allow multiple updates to a file to be transferred at once, thus reducing messaging overhead.

1.6.2.3 Client Broadcast

Another method of reducing server cpu utilization is to have the client perform the distribution of updates. Instead of sending the updates to a server, and then having the server send the updates to the others, the client multi-casts the updates to all servers in a single operation.

2 Related Work

In this chapter we present an overview of some recently-developed well-known replicated file systems.

2.1 Echo

Echo [MANN89] is a primary-site protocol. Tokens are issued by the primary to control accesses to client caches. When failures are not present in the system, Echo enforces UNIX semantics.

2.1.1 Token Protocol

Echo uses tokens to ensure client cache consistency. A client must acquire a read or write token for a file before accessing cached file data. When a client is finished with a token, it will notify the primary. Tokens are issued and recalled by the primary. When a client requests a token, the primary will issue the token if it does not conflict with any tokens held by other clients. If the token being requested conflicts with currently issued tokens, the primary will recall the conflicting tokens by sending a recall message to each client that holds one. On receipt of a recall message, a client will acknowledge the recall, and send any modified cached data for the file to the primary. Once all of the conflicting tokens have been recalled, the primary will issue the requested token. Updates sent to the primary during a token recall are applied in the same manner as any other updates would be applied.

Each token has an associated time-out. To keep tokens from expiring, clients must refresh tokens periodically. This is done implicitly whenever a client communicates with the primary. If the primary cannot reach a token holder while trying to recall a token, the primary will wait until the token expires. This allows the system to continue when token holders crash. If a client becomes partitioned from the primary, and cannot refresh its token, the client will continue to access its cache, even though the token has expired. When the failure is resolved and the client can communicate with the primary again, the client will request a token refresh. If a conflicting token has not been issued, the primary will refresh the token. If a conflicting token was issued, the client will discard its cache contents and any updates in the cache are lost. The lost updates may include some performed while the client's token was valid.

In order to track token holders, each secondary keeps a list of clients that hold tokens but not which tokens they hold. When a client requests its first token, or gives up its last token, the primary sends a message to each secondary so that they may adjust their lists accordingly. These lists are used during view changes to determine which clients hold tokens. During a view change the new primary will query each client on the list to determine which tokens each client holds.

When conflicting sessions exist at different client sites, Echo will pay a high price in issuing and revoking tokens. File data will also be required to be shuttled between the clients, the primary, and the secondaries as the tokens are transferred.

2.1.2 Reads

Reads in Echo are performed on a client's cache. In order to load a client's cache, the client must acquire a token and read the file from the primary. Even if the file is already cached, the client site must hold a read token before it can read cached file data.

2.1.3 Writes

As with reads, write operations are performed on a client's cache. When a modified cached file is to be written back to the servers, it is sent to the primary. The primary sends the updates to all secondaries in the view. The secondaries apply the updates to disk before sending an acknowledgment to the primary. Once all secondaries have responded, the call is returned to the client.

2.1.4 Failure Handling

Server failures in Echo are handled through the use of a view change. When the failure or recovery of a server is noticed, a view change is initiated. A view change consists of:

1) determining all of the servers that are in the partition and whether the partition contains a majority of the servers and, if so, electing a new primary,

2) deciding the outcome of in-progress updates, and applying those that are chosen to succeed,

3) determining which clients hold which tokens, and

4) bringing all of the servers' files up-to-date.

The first step in a view change is an election. The election phase allows each server to determine the new primary, and lets the primary know what other servers are in the partition. Next all in-progress updates need to be resolved. All of the servers that participated in the previous view confer on which non-committed updates will be applied and which will not. Which in-progress updates that will be allowed to succeed is not covered in the literature. The servers from the previous view will apply those non-committed updates that are to be committed and discard those in-progress updates that are to be aborted. Any replicas that are still out-of-date will be updated once the file service has been reinstated.

Finally, the primary must determine which tokens are held by which clients. Servers keep only a list of clients that hold token. The primary queries each client on the list and each client responds with a list of tokens it holds. If a client does not respond to this request the primary will wait for the duration of the token time-out period before issuing new tokens.

Performing all of these functions requires a significant amount of time and resources. If, during a view change, a server crashes, recovers or takes too long to respond, the view change process will start all over again. During the election phase there will also be contention for the network as each server may try to become the new primary; this will also slow down the view change process. Finally, if "perceived failures" occur (caused by network overload, for example,) initiating the view change algorithm will exacerbate an already serious problem.

2.1.5 Semantics

Echo provides a close approximation to UNIX semantics, achieving UNIX semantics in the absence of failures. When the primary revokes a client's token by allowing it to time-out, and issues a conflicting token to another client, UNIX semantics may be violated. Updates performed by the partitioned client while its token was valid will not be noticed by subsequent readers. Also, if another client updates the file, updates by the original client will be discarded even if the two sets of updates do not conflict. Both of these cases can occur after the original client performed the file system operations successfully, and possibly even after the original client's file session had closed.

2.1.6 Summary

Echo provides UNIX semantics when failures do not exist through the use of a primary-site protocol. The primary-site format of Echo will cause a bottleneck at the primary. Even though it may be simple to modify Echo to provide multiple file groups, there will still be the problem of one server becoming the primary for many file groups after a few failures. The cost of transferring tokens during periods of write-sharing, and the costs incurred during view changes are a high price to pay to achieve a higher level of availability.

2.2 Harp

HARP is a primary-copy replica control protocol ([LISK91]). The protocol deals only with servers and does not provide a mechanism for client caching. In order to improve efficiency while ensuring permanence of updates, Harp uses uninterruptible power supplies (UPS), and nonvolatile RAM.

The files in the file system are organized into logical sets called **file groups**. Each file group has a preferred primary server, a set of secondaries, and a set of witnesses. A server may participate in multiple file groups, serving as a primary for one group, a secondary for another, etc. A witness server does not have a copy of the files in the file group, and only participates in view changes or when there are not enough regular servers to form a majority. Witnesses have a tape to store event records in case the witness is promoted to the position of secondary.

2.2.1 Open and Close Operations

Harp was designed to be used as a replacement for NFS servers and as such does not use the notion of a file session. In Harp, as in NFS, to provide stateless servers, clients perform read and write operations without first opening the file, and without later closing the file, so there is no support for open or close operations.

2.2.2 Reads

All file operations are sent to the primary. Reads are simply fulfilled by the primary and returned to the client.

2.2.3 Writes

For write operations, the primary determines the outcome, packages each operation in an event record, appends the event record to its event queue, and sends the event record to all secondaries. When every secondary in the current view responds after appending the event record to their event queues, the primary commits the write and returns the call to the client. The primary then informs the secondaries of the committal of the write operation. All event records are committed in order, so the primary, and each secondary, keep a commit pointer to track the latest committed event record.

The event records are used in order to guarantee permanence of the write operations. An event log is kept in memory at each server. Event records are appended to the event log instead of to disk before secondaries respond to the primary. Stability of the event logs is enhanced through the use of stable RAM and UPSs. In case of power failures the UPSs give the servers enough time to write the logs and other state information out to disk. To help overcome other server failures, some of the file system state information is kept in stable RAM. When a failure of any server is noticed, all remaining servers dump their state to disk to minimize the chance of information loss due to the crashing of multiple servers.

2.2.4 Failure Handling

Failures in Harp cause a view change to occur, as do recoveries. During a view change, a secondary may be upgraded to become a primary, or a witness may be upgraded to become a secondary. In Harp, each file group has a designated primary. If this server is available then it will be the primary for the file group. If the server is not available then a secondary will be promoted. Witnesses cannot be promoted to primary as they have no copy of the file system. Without a copy of the files, witnesses would not be able to fulfill file access requests. When a witness is promoted to secondary, all event records are stored onto tape so that they may be replayed, at a recovering site, in the event that the witness becomes the only available current site for the file group.

When a primary or secondary server recovers, or becomes connected to the majority partition, the server will update its files from a server in the current view. Once its files are close to current, (i.e., they reflect most of the recent updates), the server will initiate a view change. During the view change the recovering server will acquire the rest of the updates.

2.2.5 Semantics

Harp achieves UNIX semantics if clients do not use caches. The goal of the Harp designers was to provide a server protocol that would provide atomic file system operations. This has been achieved at the level of the servers, but as an NFS server replacement, Harp can only provide the consistency guarantees of NFS.

2.2.6 Summary

Harp provides a primary-copy server protocol that ignores the cache-consistency problem. As a server protocol, the use of a preferred primary allows file system administrators to choose which sites will be primaries, and alleviates the problem of all file groups migrating their primaries to the same server, although, if the preferred server is inaccessible, bottlenecks may occur. In order to achieve this, Harp needs to pay a high cost during failures and recovery, performing expensive view changes in both situations. The addition of UPSs and stable memory to the server hardware is a useful tool in increasing the achievable performance of a replicated file system.

2.3 Deceit

Deceit is a decentralized protocol that uses write-tokens and **stability notification** to control file replicas. The main goal of Deceit is to provide variable file semantics in a replicated NFS server. The variable file semantics provided offer a range of consistency guarantees from no consistency checking, to consistency that approaches that stipulated by UNIX semantics. As Deceit is a replacement NFS server, there is no control over client caches. Variable semantics are provided through the variable methods of: token regeneration, replica regeneration, stability notification, file migration and update propagation.

2.3.1 Writes

The replicas in Deceit are all peers so there is no central control. Instead, access to file copies is controlled through the use of tokens and stability notification. In order for a server to perform updates for a client it must acquire the token for the file. The current token holder is found through the use of forwarding addresses; the address of the server that generated the token is kept with the file handle and each time the token is passed, which server received the token is left at the previous holder. If the chain is broken, due to an inaccessible server, then a request will be broadcast to all servers. The current token holder will broadcast a token pass message in response. After acquiring the token the token holder will broadcast a request to the other server to mark their copies unstable. The new token holder can then perform updates on behalf of the client.

When a token holder receives an update request, it broadcasts the update to all other servers. The other servers apply the update to disk and then respond back to the token holder. Once the token holder has received a response from a predetermined number of servers (which may be 0), the write call is returned to the client. The number of responding servers required is set on a per file basis. The servers without the token remain unstable until a lull in update activity occurs, at which point the token holder broadcasts a stability notice to all servers.

2.3.2 Reads

Read requests can be performed by any server that has a stable copy of the file. If an unstable server receives a read request, the request is sent to the token holder. The token holder fulfills the request and sends the result back to the originating server which passes it on to the client.

2.3.3 Failure Handling

The inability to communicate with another server does not cause any special processing to occur until an operation is requested that requires the unavailable server. If the failed server is required because it has the only stable file copy to read from, then the most current, accessible, unstable server will be made stable, and the read will be performed there. All of the other accessible, unstable servers will destroy their copies of the file. If an inaccessible server is required because it holds a required write-token, then a new write-token will be generated if allowed by the token generation setting for the file. If there are not enough accessible servers to propagate updates to, the token holder will generate new servers, then continue to propagate updates.

Recovery in Deceit is simpler than in a primary configuration, but can still require a significant amount of work. When a server recovers, it attempts to determine the token holder, and version number for all of its files. The server will then destroy all out-of-date files. For any file for which the server holds a token, the server determines if any stale or inconsistent versions exist at other servers. If the server's copy of the file is out-of-date, yet is a direct ancestor of the current version, the file copy is destroyed and so are other servers' copies of that version of the file. If the file copy is out-of-date, yet not a direct ancestor, both copies are kept and the user is notified of the inconsistency so that the file copies may be reconciled manually. Stale and inconsistent copies result when tokens are generated and updates are allowed to be performed in multiple partitions.

2.3.4 Semantics

The variable file semantics are achieved on a per file basis through the use of some file settings. These settings are: token regeneration, replica regeneration, stability notification, file migration and update propagation. The settings allow each file to be adjusted for consistency, availability or efficient accesses.

File migration either allows or disallows a file to be migrated to a server that receives a request for a file operation yet does not hold a copy of the file. Replica regeneration determines the minimum number of replicas that must be kept up-to-date. If the number of replicas drops below this level the token holder will create another replica by copying the file to another server. The update propagation setting determines the number of servers that must reply to the token holder before an update call can be returned to the client.

The token generation setting determines whether tokens can be generated to replace missing tokens. If the token regeneration setting is high, a new token is created whenever the old token is not available. A medium setting allows token generation only in majority partitions, and a low setting disallows token generation completely. When tokens are allowed to be generated, file version vectors are used to track the different lineage of the files.

The stability notification setting turns stability notification on or off for the file. If stability notification is turned off, file copies are not made stable during periods of updates. This setting trades better performance for weaker consistency guarantees.

2.3.5 Summary

Deceit is unique among those file systems surveyed here, in that the semantics are variable. Although a variation of consistency guarantees are provided in order for the users to determine their requirements, Deceit fails to provide the option of UNIX semantics. In partitions, a Deceit client will always be allowed to read from accessible servers, even if the token holder is not accessible. With the token holder possibly in another partition allowing updates to the file, UNIX semantics cannot be achieved. Even if Deceit were to provide a strict form of semantics, they would be nullified by the client caching scheme used by NFS and adopted by Deceit.

2.4 Coda

Coda is the successor to the Andrew file system ([KIST91], [SATY90]). The development of Coda was directed to address specific issues including the operation of clients while not connected to the servers and higher availability through replication. Both of these are achieved through the use of an optimistic replication strategy. The semantics of the Andrew system are made looser in order to support clients that are partitioned from all servers.

In Coda, UNIX semantics are not used. Instead, an approximation to session semantics is adopted. This allows individual read and write operations to be ignored by the replication protocol as they are always performed on a client's local copy, and cannot create inconsistencies among the replicated copies. Instead, the protocol deals with providing a copy on open and with the propagation of updates on close.

2.4.1 Open

When a connected client opens a file, it is guaranteed to receive the most current copy of the file held by any accessible server. The client sends a request for the file to its **preferred server**. A client's preferred server accepts all of a client's opens and will remain the client's preferred server until the server becomes inaccessible. The client also contacts every other available server to make sure that the copy of the file retrieved is the most current. If the file was not the most current, the client notifies the servers and retrieves the most up-to-date copy. Thus all out-of-date servers are notified that they hold stale copies.

The preferred server promises to notify the client if the file becomes out-of-date. On subsequent open operations for the file, the client will use the cached copy unless the preferred server has notified the client that the cached copy is out-of-date.

2.4.2 Close

On the close of an update session, the client broadcasts the file to all accessible servers. Each server responds to the client stating its version number of the file and the client computes the new version number of the file and notifies all servers what the new version number is.

2.4.3 Reads and Writes

Individual reads and writes are always performed on the client's local copy of the file. If there are multiple file sessions on a client machine, they will all access the same copy of the file. Accesses to this local copy behave according to UNIX semantics.

2.4.4 Failure Handling and Recovery

Server failures and partitions are noticed by clients when they do not receive a response from a server during open and close operations. The client will stop sending any file system requests to that server until the server becomes accessible again.

Every τ seconds a client attempts to communicate with each inaccessible server. If the client receives a response from one, the client marks all files cached locally to be out-of-date. Subsequent opens will then cause the files to be checked for consistency and currency and they will be refetched if necessary.

On recovering from a failure, Coda servers do nothing. The server will be notified on a per file basis during open and close operations if its copy of the file is out-of-date. When a server is notified that it is out-of-date, it will copy a current version of the file from another server. If a server is found to hold an inconsistent copy, the replicated copies will have to be reconciled manually. Until the reconciliation is performed all operations on the file will fail.

2.4.5 Disconnected Operation

Disconnected operation refers to machines, operating while communication with servers is not possible. In order to facilitate disconnected operation, clients hoard files. Clients cache files that are deemed necessary for remote operation (e.g., system binaries.) Clients also keep recently accessed files cached. When disconnecting voluntarily, the client will make sure that all cached files are up-to-date.

While disconnected, file sessions are performed on the cached copies. All operations are also logged for playback later. Upon re-connection, modified files are transferred to the servers. The playback log is used if any conflicts exist to allow reconciliation and partial updates.

2.4.6 Semantics

The semantics achieved by Coda are very loose. Session semantics were approximated by Andrew, and that approximation was loosened in order to better support disconnected operation. While the network is fully connected, the semantics of Andrew are achieved. File sessions on a single client machine share the same copy of the file, and accesses to the local copy adhere to UNIX semantics. When there are disconnected machines, or machines in different partitions of the network, clients are allowed access to files as long as a copy of the file is available. This copy is only guaranteed to be the most recent available version of the file. No constraints are placed on file accesses to prohibit inconsistencies, although tools are provided for merging inconsistent file versions. The merging of files is done manually, and most merging of directories is done automatically.

2.4.7 Summary

Coda provides a method for replication that will not overly burden large scale networks, and that allows clients to operate in disconnected mode. The choice of semantics and the optimistic replica control are crucial in supporting these characteristics. By requiring that clients perform the update propagation and consistency checking, Coda removes some of the burden from the servers.

2.5 A File System for Mobile Clients

This section describes a file system protocol designed to facilitate mobile clients (e.g., portable computers) that is described in [TAIT91] and [TAIT92]. The system is a **contact**[1] based system, where each client has its own contact server to handle all of its file accesses. Mobile clients are supported by allowing a client to change its contact depending on the location of the client site.

2.5.1 Reads

This protocol offers two different read operations: a **loose_read** and a **strict_read**. The loose_read does not guarantee anything about the value returned and will return the first value for the file it finds, regardless of whether the value is stale. In contrast, the system is claimed to guarantee[2] that a strict_read is performed on the most current version of the file A loose_read first checks the client's cache, then the client's contact, and then checks each other server until a copy of the file is found. The first copy of the file found is used to fulfill the read request.

Their implementation of a strict_read queries all servers and all other clients' caches that may update the file in order to determine the current version of the file. A client that strict_reads a file and has the ability to update the file is a **potentially consistent writer** (PCW.) The ability to update a file is based on the access rights for the file and not the mode in which the file is opened. The contact checks the other servers and all PCWs. The other servers and the PCWs send the file to the contact, and the contact sends the most current version to the client.

2.5.2 Currency Tokens

In order to make strict_reads more efficient, a **currency token** (CT) can be issued to a client. A currency token is only issued if there pfare no other PCWs for the file. The currency token guarantees the client that the first copy found is the most current version when checking first the cache, then the contact and finally the other servers. This allows strict_reads to be performed on the client's cache without communicating with any servers.

1. Contact is a term that we coined. The authors use the term primary but we felt that it was confusing as the system does not use a primary-copy/primary-site protocol.

2. The system is supposed to guarantee this but does not.

Currency tokens are revoked when another client becomes a PCW by requesting a strict_reads on the file, or when a client cannot be reached by its contact for an extended period of time. The ability of the contact to unilaterally revoke the currency token violates the guarantee that the currency token is supposed to provide, and results in this file system failing to achieve the desired semantics. Thus, it appears that strict_reads can return stale data.

2.5.3 Writes

This protocol controls consistency among file copies by noting the existence of PCWs at all servers. Whenever a PCW is introduced into the system, all of the servers are notified. If any server cannot be notified the strict_read that marks the arrival of the PCW will fail. Currently, once a client has obtained PCW status, it will remain a PCW. How PCW status is to be revoked has yet to be decided by the designers of the system.

It is assumed that before a client updates a file, the file is first strictly read into the clients cache. All updates are then performed on this cached copy. The client's contact will request periodic cache flushes in order to propagate updates to the servers. The client will keep all updates cached until the contact notifies the client that the updates can be purged. The contact will notify the client that updates can be purged after N servers have responded to the contact's propagation of the updates. N-1 is the number of server failures that the system is configured to handle. Update propagation at the request of the server is called **server-based writing.**

This protocol allows conflicting updates to create inconsistent cached copies. Although the system will be aware of the different writers it will make no attempt to coordinate the updates. The literature does not discuss how inconsistent versions are dealt with.

2.5.4 Failure Handling

When a site fails or is partitioned from the majority group, elections will be held. An election is held to determine the best contact for each client of the inaccessible server. Whether one server will take all of the inaccessible server's clients, or whether each client will be dealt with separately is not clear. The elected contact will try to determine all of the currency tokens that are held by its clients. When a client is given a new contact, the new contact informs the client by performing a server-based write.

The client determines the server to communicate with as the last server that performed a server-based write. This process is not discussed in detail in the paper, but it appears that during a partition failure a client could have two contacts. If the client is disconnected from its contact, the other partition will still elect a new contact for the client. When the partition failure is repaired, there will be two servers that believe they are the client's contact.

If a server is not the contact for a client, the failure/partitioning of this server will not cause any excess processing, although the failure will result in loss of availability as no new PCWs can enter the system since all servers must note the existence of PCWs. If fewer than N servers remain reachable then update propagations will fail, and strict_reads that require server participation will also fail. Also, if the client cannot flush updates to its contact, then the client's cache may fill up, and cause write operations to fail.

2.5.5 Semantics

The semantics achieved by this system are weak. The paper claims that the system achieves **one-copy UNIX serializability** (1USR) and defines this as the semantics achieved in a centralized UNIX system. The paper also states that inconsistent files can arise from this form of semantics (and also in UNIX); therefore our definition of UNIX semantics appears to be different, although we believe that our definition of UNIX semantics holds in a centralized UNIX environment.

The system definitely does not support UNIX semantics, as we defined them, as conflicting file versions can and will result if two processes concurrently update a file. It appears that after a file has been loaded into a client's cache, no effort is made to reconcile updates on the cache with updates on other caches. This protocol also appears to fail to achieve its stated objectives for strict_reads, since it allows the contact to revoke CTs unilaterally, resulting in disconnected clients reading stale data.

2.5.6 Summary

This protocol uses a unique method of update propagation in allowing the servers to determine when updates will be transferred from a client's cache. The provision of two read operations provides an alternate method for allowing variable consistency at the choice of the programmer. This method may not necessarily be better as the variable consistency is tied to a program, not to a file, so one user can be attempting to access a file in a consistent manner while another is using the file in a 'loose' manner.

The use of PCW status needs to be refined. Currently, once the protocol issues PCW status to a client for a file, that client always has PCW status. Since this is tied to a client machine and not a user, if a user normally accesses his files through a number of machines there will be two or more PCWs for the user's files, and thus no CTs.

3 Protocol Overview

3.1 Motivations and Goals

Many of the previously built systems employ a primary-copy strategy. Primary-copy protocols ignore a potential benefit of replicated file systems: the physical locality of the file servers. In large networks, communication delays can vary dramatically depending on the relative locations of the communicating sites. If there is a file server that is physically close, why send requests to a server that is not?

Primary-copy systems also require more participation in individual operations. In order for the secondaries to be able to replace the primary in case of failure, they are kept up-to-date. All updates are immediately sent not only to the primary but also to enough secondaries so that the updates will be noticeable by a new primary after any successful view change. If this requirement is removed, the method of update propagation can be changed to send updates when they are required, when the system is not busy, or when it appears that the file will not be updated again soon.

Involving the primary server in all file system requests regarding a particular file group causes the primary server to become a bottleneck. All file sessions involving a particular file group will be competing for the primary's resources. Allowing each server to serve requests provides an opportunity for dynamic load balancing in the file system; primary-site systems ignore this opportunity. Primary-copy systems allow static load balancing through the use of multiple file groups, but view changes can force one site to be the primary for many groups.

A more distributed approach would exploit physical locality to enhance performance. When a client opens a file session, a physically close server can be chosen for that session. If that server is not available, the next best server can be chosen, and so forth. There need be no constraint on which server a client chooses to deal with for any given file session.

Decentralizing the protocol will also facilitate load-balancing. If a server determines that it is too overloaded to handle a request and knows that others are not, it could redirect the request to another server. The information required for determining server load can be sent with the messages passed between servers as file state information or updates are moved about the system. This method of load balancing will also remove the bottleneck problem found in primary-copy protocols. Even without explicit load balancing, if all client sites generate similar loads, a natural load balancing scheme is provided.

Immediate update schemes have the disadvantage of involving multiple servers in individual write operations. The **peer server** approach can remove this requirement, at the expense of availability, as long as a sufficient number of servers know where the most current copy can be found. A single server would accept updates from clients for a file, and return the call to the client without involving other servers. At some later point the updates will be sent to the other servers in order to increase the replication of the current copy, and to improve the probability that accesses can continue in the presence of failures. In the case of failure, recently updated files may become unavailable, but the majority would remain available as most files would be sufficiently replicated to be available when server failures occur.

Failures in a primary-copy system cause the invocation of a view change. During the view change, access to the file system is blocked and all clients must wait while a new primary server is elected and stale replicas are brought up-to-date. In protocols where there are multiple file groups, there may be multiple view changes, one for each group. This will cause each of the changes to compete for server cpu time, network bandwidth, and server disk bandwidth, slowing every view change process. Protocols that have expensive failure and recovery operations behave even more poorly in situations of perceived failures. Expensive recovery operations, while attempting to provide better availability by noting the repair of a failure, may degrade the performance of a system that is working just fine. Low cost repair operations will punish users less for repairing their machines.

Most of the systems that are covered here assume an academic environment. In such an environment certain situations are much more frequent than others. Our approach capitalizes on this information by focusing on the frequent situations and streamlining them. The infrequent situations, such as write-sharing, may be slower, but by keeping the frequent operations faster, the overall performance is improved.

The trend towards distributed computing has emphasized the cost of obtaining strict file system semantics. A centralized file system has the advantage of low cost for serialization of accesses. Adding client caches into the mix introduces the cost of cache consistency, and serializing accesses to files residing in multiple client caches. Upon replicating the file system, the cost of providing strict semantics is increased further still. All of the replicas, and all of the clients' caches must only allow accesses that conform to the consistency guarantees. The stricter the guarantees, the more work that is required to meet them.

A close examination of previously-implemented replicated file systems reveals this perceived cost, as none of the systems provide strict semantics under all conditions. Echo fails to provide 1SR during failures by allowing cache tokens to time-out. Harp completely ignores the problem of client cache consistency, and the mobile-user protocol does not appear to meet its stated semantic goals. Deceit can provide a variety of semantics, but the main focus is on a weak semantic notion relying, like Coda, on infrequent write-sharing to avoid inconsistencies. Deceit cannot provide UNIX semantics; their closest approximation requires a significant decrease in efficiency, and also does not consider client cache consistency. We believe that UNIX semantics are achievable in a replicated file system, and that the cost need not be over-bearing.

Of the systems that we reviewed in the previous chapter, only Coda mentioned complete file sessions. The other systems considered 'periods' of access. Given that standard interfaces to file systems include *open()*, and *close()* file system calls, protocols that supply file service should take advantage of these. An *open()* operation is effectively a letter of intent, warning the file system to be ready for a barrage of requests. The corresponding *close()* operation, likewise tells of the end of a series of accesses. With this added information, a file system can prepare for a file session during the *open()* call, and cleanup at the end, after a *close()* call. Work performed during the *open()* and *close()* operations in order to provide more efficient *read()* and *write()* operations can be amortized over the whole file session. In contrast, a file system that does not note these important operations, will need to perform more work during individual *read()* and *write()* operations, or make a best guess at how long periods of accesses will last.

Our work attempts to address all of these concerns by developing a decentralized replicated file system. Through the distribution of server responsibility we allow failures to be handled more easily and more efficiently, and strict semantics to be achieved with very little cost. Our system does not address the possibilities of load balancing, but the structure of the protocol inherently facilitates load balancing.

3.2 Assumptions

Like the other systems developed recently we designed this protocol for an academic environment with loads similar to those load measured in [BAKE91]. In that environment, read only file sessions make up about 88% of the mix, while write only file sessions make up 11% and the remaining 1% are read-write sessions.

We assume that there will be several read and/or write operations per file session. Using this assumption, we have tailored the protocol to achieve fast read and write operations. The cost of replication, and consistency control would be paid mostly during the open operation, and partially during the close operation. The benefit is that the cost is paid once for all operations, so that individual read and write operations, which are more frequent, could be made efficient. Also, since write-sharing is rare, consistency is not an issue for most file sessions, and thus update propagation is not performed until a file session is closed. Propagating updates once removes the excess overhead paid by sending the updates in multiple messages. When write-sharing does occur, we ensure consistency by disabling client caching, forcing operations to be sent to the servers.

File traces were obtained from Carnegie Mellon University ([MUMM93]) that provided more detail about file sessions than other sources we had come across. From these file traces we noticed that the average number of read or write operations per session was approximately 8. Table 3-1 shows the average and median

Table 3-1: The average number of operations per session as observed in the file traces.

	Read Session	Write Sessions	Read-Write Sessions	All Sessions
% of the mix	89.3%	9.0%	1.7%	100%
Average	4.96	33.28	29.22	7.91
Median	2	2	2	2

values for each session type. Although the overall average is 8, most of the sessions (> 50%) had only 1 or 2 operations per session.

The intended operational environment requires that client machines should not be trusted. As users have direct access to the machine hardware, we felt that trusting client machines would not be appropriate. Users could easily add extra hardware, fiddle with the internals of the system, and should be expected to shutdown the system without warning whenever it appears that the system has crashed or the response time becomes unbearable.

3.3 General Overview

The protocol we descnbe here distributes the processing so that no one server is ultimately responsible for control of the file system. The protocol achieves UNIX semantics, controlling the replicas and client caches to ensure that consistency is maintained.

Each server provides service to files as requested by clients, and is responsible for letting a majority of the servers know what file sessions it is working on, and for ensuring that new sessions that it starts will not compromise UNIX semantics. Each piece of file state information (i.e., the current version number, record of each active file session, etc.) is replicated at a majority of the servers, not at all servers. Keeping state information replicated at a majority of the servers allows a server to determine the state of the file system by querying a majority of the servers and merging the information obtained. By not requiring that all servers keep up-to-date information, and by not specifying which servers must be kept up-to-date, our protocol alleviates the need for views and view changes.

In this protocol there are four main entities: contacts, servers, **agents**, and clients. A client (process) is an application that uses the file system service. Each client machine has one agent (process) that controls caching for that machine and provides the file system interface for the client processes. An agent intercepts client requests and either fulfills them or passes them on to the contact. The agent is also responsible for fielding requests from the servers.

3.3.1 Sessions

When a client starts a file session, the agent chooses a server to handle the session and sends the open request to that server. This server is referred to as the contact. Each file session is represented by an **agreement** between a contact and an agent. All operations associated with the file session that require a server will be fielded by the contact. The contact guarantees that all successful operations are performed in accordance with UNIX semantics and the agent agrees to send all remote requests associated with this agreement to the contact.[1]

As part of its duties, the contact is required to guarantee that, if an agent has cached file data, the data is current. During an open, the contact will send file data to the agent if there are no conflicting file sessions, and if the agent does not already have the current version of the file cached. The contact will notify the agent to stop serving requests locally, using the cached file, if a conflicting file session starts during the life of the an agreement. On close, the contact is responsible for accepting any updates from the agent and for distributing them to the other servers to increase availability.

UNIX semantics require that each file operation be performed on the most current version of the file. This increases the cost of ensuring cache consistency as updates written during a write-shared file session must be reflected in any conflicting sessions. Our protocol handles this problem by disabling client caching for files that are being accessed in a write-shared mode. File operations during write-shared file sessions will be sent to the contact for processing. If the conflicting file sessions have different contacts, then a two phase commit protocol is used to ensure that updates are applied at all of the contacts. By keeping all contacts up-to-date, read operations need only involve the contact for the agreement. Since we expect that there will be many more read operations than write operations, the above approach improves the efficiency of the common cases at the expense of the uncommon cases.

3.3.2 Failure Handling and Session Progress

The server failure handling operation is called **change-of-contact** and may be performed for each server that fails. When a server fails, client sites that have agreements with the server will be left without a contact. In order to allow processing to continue, the change-of-contact operation is introduced to replace the contact in all agreements held by a specific inaccessible server with an accessible server. Due to the distributed nature of the protocol, some agreements cannot have their contacts changed during a failure. Those agreements for which the contact cannot be changed will not be able to proceed with any requests that require participation of the contact until the contact becomes accessible again. Situations where the change-of-contact operation will fail are described in more detail later.

1. Except in the case of failures.

3.3.3 Semantics

The semantics of a file system are an important part of the service it provides. The semantics define what can be expected from the different system calls provided for the interface to the file system, in the presence of concurrent accesses and failures.

UNIX semantics give a clear, concise definition of what can be expected when one or more clients are performing operations on a file. The outcome of a system call is simple: all operations are performed on the most recent copy of the file. Other semantics notions, like the semantics offered by Coda, are non-deterministic given the set of previous operations. Outside influences must be factored in before the outcome of an operation can be known. For example, in Coda, a read operation on a file can give two different results solely dependent on whether a conflicting access is being performed on the local site, or at a remote site. Such semantics may be acceptable on single user machines, but many networks support cpu-servers and thus multi-user machines. A user of the file system must then be prepared for different results depending on which computer is being used.

For reasons like this, we have chosen to adopt a strict semantic notion, UNIX semantics, which has been used extensively and one which provides logical results to file system operations. UNIX semantics have the benefit of being able to provide most other types of semantics, if one wishes to obtain them, without too much extra processing. For example, session semantics are simply achieved by performing whole file reads into a buffer, and then accessing the buffer until close. If the buffer is modified it is written out when the file is closed.

The semantics of a file system can also be affected by how caching is implemented. Although UNIX semantics provides strong guarantees to concurrent operations, they fail to provide strong guarantees in the presence of failures, since they do not guarantee permanence of update operations. In some UNIX systems, updates are cached in the kernel and not written to disk immediately. If the host fails before the updates are applied to disk then the updates will be lost, even though the system call has returned. In fact, the file session can end, or a different file session can read the updates, and the permanence is still not guaranteed as the updates may still only remain in the kernel buffer cache. The effect is that a client of the file system is not guaranteed that its updates will be reflected after the file session has ended.

The protocol we outline in this chapter makes an additional guarantee to clients of the file system. The effects of a write system call are guaranteed to be stable if:

1) the file session ends successfully, or

2) any other file session reads the updates, or

3) the client explicitly requests a cache flush and it succeeds.

If these conditions are not met there is no guarantee that the updates will be stable.

This choice of file semantics has the effect that the results of client site failures are non-determinable. If a client machine crashes during an update session, one cannot be guaranteed that the successful updates will kept, or that they will be discarded. Only updates from file sessions that have successfully closed, have been flushed or have occurred during a concurrent write-sharing session can be guaranteed to survive failures.

3.4 File System State

Failures of servers must not cause loss of state information. Due to the distributed nature of the protocol, loss of state information could result in breaches of the semantic constraints. In order to keep state information stable, the file system can use uninterruptible power supplies (UPSs) and/or stable RAM to store the state information. Using such extra hardware will allow processing at servers to be performed with fewer disk accesses and result in better performance. UPSs help by allowing servers to write state information out to disk when a power failure occurs. Stable RAM (battery backed up) helps to overcome server crashes by saving important state information in memory that will not be lost by unexpected server shutdown. It is assumed that during failures a server will not overwrite these portions of its memory. Both of these hardware methods are used by Harp to enable state information to be kept stable without paying the cost of writing it to disk. In lieu of such additional hardware, state information can be kept stable by using the servers' disks.

Each server keeps state information about every file in the file system. This state information is used to:

• determine which servers have a current copy of the file,

• what agreements exist for the file, and

• the status of in-progress updates.

When a file is opened, the state information for the file is acquired from a majority of the servers, and is merged to determine the current state for the file. No server can determine the accuracy of its state information for a file without first receiving the state information from a majority of the servers. The only guarantee that a server has is that, if the server does have an agreement, it will know about it.

File Version Record =	Server Id	
	File Version	
Agreement Number Record =	Client Machine Id	
	Client Process Id	
	Contact Server Id	
	Number	
Change Record =	New Agreement Number Record	
	Old Agreement Number Record	
	Change Result	(none/success/failure)
Agreement Record =	Agreement Number Record	
	Mode	(read/write/read-write)
	Open Result	(none/success/failure)
	Close Result	(none/success)
	Change Record^{Cn}	
File State =	[	
	File Id	
	File Version Record^{Sn}	
	Agreement Record^{An}	
	]^{Fn}	
Change Log =	[	
	Change Id	
	Change Record	
	]^{Cn}	
Server Information =	[	
	Server Id	
	Lowest Open Agreement Number Record	
	]^{Sn}	
	Next Agreement Number To Issue	
Fn = number of files	An = number of agreements	
Sn = number of servers	Cn = number of change-of-contact operations	

Figure 3-1: File state information kept by each server about the files replicated at the server.

During periods of failures, and after failures have been resolved, a server's state may be incorrect. The state information will never forget an agreement, but may show that an agreement exists when it no longer does. This situation may happen when an agent and contact cannot communicate, and the agent requests a change-of-contact. If the change is successful, odds are that the original contact will not know about the change and will believe that the agreement still exists. The original contact will notice the change-of-contact if it performs another open operation for the file, or possibly when the file is closed by the client.

The state information that is kept at each server is shown in Figure 3-1 and in Figure 3-2. The figures depict the logical form of the information and do not necessarily show the actual form as stored by the servers.

Write Ack Record =	Server Id Has Acknowledged Ack Timestamp
Write Entry Record =	Agreement Number Write Id Write Ack Record^{Sn} Start Length Data Result
Write Log =	[File Id Write Entry Record^{Wn}]^{Fn}

Fn = number of files
Sn = number of servers
	Wn = number of write requests

Figure 3-2: Information on write operations kept by contacts.

Each client machine has an agent process. The agent must keep information about what agreements are local to the machine. In contrast to the server state, an agent's state need not be stable. The agent state is used to determine what server to use as the contact for each agreement, to keep track of change-of-contact operations and their outcomes, and to track failed operations for use in server recovery. The information kept by an agent is shown in Figure 3-3.

Agent Agreement State =	[	
	Agreement Number	
	File Id	
	Mode	(read/write/read-write)
	File Pointer	
	Change^{Cn}	
	$]^{An}$	
Agent File State =	[	
	File Id	
	Caching Allowed	(true/false)
	Cached	(true/false)
	Version Cached	
	$]^{Fn}$	
Agent Failed Operations =	[	
	Operation Id	
	Server Id	
	Failure Code	
	$]^{On}$	
A_n = number of agreements	C_n = number of change-of-contact requests	
Fn = number of files	On = number of failed operations	

Figure 3-3: Information about local agreements kept by each agent process.

3.4.1 Determining Active Agreements

During open operations and change-of-contact operations, servers pass file state information between them. A contact needs to determine what agreements exist for the file being opened, and where the current copy of the file resides when performing either of these two operations. In order to determine this information, the server performing the operation requests state information from a majority of the servers and merges all of this state information to get the current state of the file.

To merge two states, a server combines all agreement entries in the states. Duplicates and outdated agreement records are then removed. Agreement information becomes outdated when the **lowest open agreement value** of the contact surpasses the agreement number. Each server keeps track of its own lowest open agreement number, and passes this to the other servers during communications for open, close, and change operations.

A state that has been merged using this process is not guaranteed to be complete. A majority of the servers must participate for the information to show all active agreements. Even if a majority do participate, some agreements may appear in the state as active, even after they have been closed. If a change-of-contact operation was performed on an agreement the state will show this, but the outcome of the change-of-contact operation is not guaranteed to be reflected. If a change-of-contact operation's outcome is not known, the change-of-contact is assumed to be in progress.

Although the state information may be incomplete, enough information can be acquired from the merged state. If an agreement has been closed, but the close is not reflected in the state, the server performing an open or close operation will 'think' that the agreement is active and act as if it were. This will guarantee that, if the agreement were active, no violations of UNIX semantics could occur. On the other hand, if the agreement has closed, there may be a loss of availability.

During an open, if there is a conflicting agreement that appears active, even though it has been closed, the open will fail. Since the conflicting agreement has actually closed, there is be no real conflict, only a perceived one. This becomes a problem only during periods of failures. A contact performing an open operation will always attempt to communicate with the contacts holding conflicting agreements. If the communication is successful, the opening contact will notice all closures, and the open will not fail. It is only when the opening contact cannot communicate with conflicting contacts that closure of the agreement will not be seen and will cause failure of the open.

3.4.2 Finding A Current File Copy

The contact for a file session is required to obtain a current copy of the associated file. When a file is opened, the contact merges the responses from a majority of the servers. This merged state will show the last close operation for the file, or will show active agreements for the file. The contact can deduce the contact for an active agreement or, if no active agreements exist, the contact for the last closed agreement. In this manner, the opening contact determines which server created the current version of the file, and what the current version number is. The merged state will also show the versions kept at each server; from this the contact can tell if its copy is up-to-date (i.e., if it is the same version as the server it has deduced to be current.) If the contact's version is not current, it will request a file transfer from a server that holds a current version.

3.4.3 Tracking In-Progress Updates - The Write Queue

All write requests are applied to a stable write queue before being applied to the actual file. The write queue is used to order and store all write requests so that each call can be returned to the requesting agent before the write is applied to disk. This is to remove the disk write from the critical path and speed up the write system call. When a write request is received by a server and there exists a conflicting agreement on this file, the write cannot be immediately applied to disk. First the write must be committed using a two phase commit protocol involving all contacts for the file. The write and all associated acknowledgments are kept in the write log and thus are stable for the duration of the 2PC protocol.

The write log also serves to uniformly serialize concurrent write requests at different servers by ordering the write requests through the use of timestamps. Exactly how the ordering is determined is examined later in section 4.3.

3.4.4 Stability of Server State

The state information kept at a server must be guaranteed to survive failures. If the information does not survive, consistency may be compromised. For example, if a server fails and loses all of its state, then a majority of the servers can participate in an open but not all active agreements will be known to the opening contact. Consider the following example in which there are five servers S1, ..., S5. S1 opens file *X*, and during the *open()* tells S1, S2, S3 about the *open()*. S1, S2, and S3 subsequently fail. On recovery, S3 forgets all about the agreement S1 has for file *X*. S5 then opens file *X* for a client. During the *open()*, S5 communicates with S3, S4, S5 to build the state for the file. A majority of the servers participated, yet they do not know of the agreement that S1 is participating in. If S1's agreement updates the file, the updates will not be seen by the S5's client.

The easiest method to ensure server state stability is to write all state changes to disk, or tape, before acknowledging or acting upon the change. For example, when a server is opening a file, the open request could be logged to disk, then the request for other server's state would be performed. Next the server would receive all of the state information, merge the states and determine the outcome of the open operation. The merged state and the outcome of the operation would then be saved to disk, the agent would be notified of the open result and in the case of the *open()* failing, the other servers would be notified. The problem with guaranteeing state stability in this manner is the speed penalty paid by writing the state to disk. As an alternative, the state can be kept in memory made stable by battery backup, or through the use of UPS's that can protect against power failures.

Battery backed memory provides better stability guarantees in that all servers could fail at once and no state would be lost. If only UPSs are used, then when a server fails for reasons other than a power failure, file system state is lost. The state will need to be recovered from other sources (other servers and agents) before the recovering server will be able to serve file system requests. If a majority of the servers fail within a short time period while using only UPSs to safeguard state information then irrecoverable information loss can occur.

State information for all files need not be kept in memory. When the last local agreement is closed, the state information for it can be written to disk. Writing outside of the critical path will not directly affect the response times of individual operations. Reading in a file state from disk will, however, affect individual open times. File state for agreements held by other servers will be kept in memory until the agreement closes, a generous time-out expires, or when the in-memory file state cache is growing too large.

When a power failure occurs, each server will force all in-memory file state information to disk. If UPSs are available they provide enough time for the file servers to get the information out onto stable storage before the servers shut down. In the case of server failures that do not allow a graceful shutdown, battery backed-up memory can keep file state stable until the server is rebooted. Also, due to the distributed nature of the algorithm, almost all information kept by a server can be recovered from the other servers and from the agents. Without the use of battery backed up memory and without saving the state information to disk, the file system can still perform correctly if only a minority of the servers fail within a τ second period. τ seconds is the amount of time required for a server to recover its state from the other servers and from the agents.

3.4.5 Stability of Agent State

Agent state information is not required to remain stable when a server's state is completely stable. This is due to the design decision to have all sessions on a failed client machine implicitly closed. All dirty cache entries are lost, except in cases outlined earlier, and all open files are automatically closed.

In the case where a server does not supply complete state stability (i.e., no UPSs/stable RAMs and not all updates are forced to disk), agents will use their state information to help the recovery of servers. Agents will keep a record of operations that were requested but received no response from the contact so that a recovering contact can determine any possible in-progress operations.

3.5 Cache Control

Our protocol covers client caching as well as replica control. This is necessary for the file system to guarantee UNIX semantics. The protocol uses whole file caching. In order to maintain consistency a client's contact can call-back the cached copies.

On open, the contact sends a current copy of the file to the agent if it is not already cached. If there is a pre-existing conflicting file session then the contact will not send the file to the agent, and will force the agent to send all requests to the contact. If, on opening a file session, there is no room left in the cache because it is in use by other file sessions, the new file session will be performed remotely, sending all file requests to the contact.

If a conflicting session starts while an agent has the file cached, the agent's contact will request a call-back. When an agent receives a call-back request from its contact, the agent will act as follows. If the cache entry is dirty, the file will be flushed back to the contact, otherwise the cache entry is just deleted.

On close the agent will send the cache contents back to the contact. This is to guarantee stability of the updates before completing the closure of the file session. Once the cache contents are safe at the contact, the close operation is returned to the agent. The agent will not discard cache contents after closing a file session in order to allow the cache to be reused if the file is opened again soon after closing. If the file is not opened soon after closing, the cache entry may be cleared to make space for other file sessions.

The protocol was designed to be used with whole file caching, but could easily use partial file caching, or no caching (although a performance penalty would be paid). We chose to use whole file caching as most files in the assumed environment are small, and 76% of the files accessed are completely read or written [BAKE91]. According to [TAIT91], whole file caching is a good method for files up to 100K, after which the benefits of transferring the whole file are reduced, as large files are not usually completely read or written during a file session. Our protocol can easily handle larger files by switching to a partial file caching scheme when files exceed some predetermined size.

4 Protocol Description

This chapter describes each of the operations of the protocol except for failure handling which is covered in the next chapter.

4.1 Open

The open operation serves to notify the file system of the intention of the client to process a file. This notification allows the file system to prepare for the file session by updating the chosen server, if it is stale, performing any necessary call-backs, and possibly filling the client's cache. It is during the open system call that the contact is chosen and the agreement is setup; an agent can have a different contact for each file session agreement. If the open operation succeeds, the contact will be the only server that the agent communicates with in performing operations related to the file session, except in the case of failures.

A summary of the open operation is shown in Figure 4-1. The first step in the open operation is for the agent process to choose a server to be the contact for the file session. The server is chosen based on physical proximity, and perceived availability. The contact would normally be located on the same local area network, and would not have recently failed to respond to the client. Once the agent chooses the contact, it sends the open request to the contact.

1) Agent, A1, chooses server, S1, that is to be the contact

2) S1 notifies a majority of the servers of the open request

3) All notified servers return their file state to S1

4) S1 determines the current state of the file

5) S1 notifies any contacts with conflicting agreements that were not notified in step 2.

6) S1 acquires a current copy of the file, if its copy is stale

7) S1 returns the result of the *open()* to the agent and if the *open()* failed, tells the other servers of the failure

Figure 4-1: The basic steps performed during a open operation.

Upon receipt of an open request the contact notifies a majority[1] of the servers. On choosing the set of servers that the contact should notify in step 2, the contact examines the local state for the file. Any server shown in the state to have an agreement for the file is included in the set. It should be noted that at this point the contact does not necessarily know of all the agreements that exist for the file, and its state may show agreements that no longer exist for the file. The opening contact is attempting to include all conflicting contacts for the file in the first phase of the open operation so that they will not need to be contacted at a later point. Those servers that have recently been noticed to be unavailable are not included in the set unless there are not enough other servers to make up a majority. Once the set has been determined, the open notification is sent.

In response to the notification, in step 3, each server notes the open operation in its local file state and returns its local copy of the file state[2] to the opening contact. If, upon receipt of the open notification, a server sees that it has a conflicting client for the file, and the client has the file cached, the server will request a cache call-back. The server will not respond to the contact until the call-back is complete or aborted due to a time-out.

If a call-back fails, the server performing the call-back will note this in its response to the contact performing the open operation. The contact will then determine, once the merged state is complete, if the open

1. This majority includes the contact itself.

2. The write logs are not included in the state passed between servers during open operations

operation can continue. If the client, for which the call-back failed, has changed contacts for the agreement in question and subsequently closed, the open operation can continue. Otherwise the open operation must be aborted.

In step 4, the opening contact collects all of the responses in order to piece together a current state for the file. As each response is received, the state included in the response is merged into the contact's local state. When all of the responses are received, the contact's state for the file will be complete enough to reflect all conflicting file sessions.

From the merged state, the contact can determine all of the conflicting contacts for the file. This list of contacts is compared to the list of servers that were notified of the open request. In step 5, any contact that was not sent in the original notification is then sent the open notification. The opening contact must wait for all conflicting contacts to participate in the open operation before continuing. Requiring that all conflicting contacts participate guarantees that all contacts become aware of each other.

Once it has been determined that all conflicting contacts know about the open request, in step 6, the contact makes sure that it has a current copy of the file. If the contact does not have a current copy of the file, the contact will look in the file's state to find all servers that do have a current copy of the file. One of these servers will be chosen, and a request for the file will be sent to that server. The requested server will then send the file back to the opening contact. If the contact cannot obtain a current copy of the file, the open will fail.

The final step, step 7, in the open operation is to return the result of the open operation to the agent and to notify the other servers if the open failed. The client is sent the result of the open, and possibly the file. The file will be sent to the client if there are no conflicting file sessions and the client does not already have the current version of the file cached.

4.1.1 Write-Shared Opens

In order to guarantee consistency, extra processing is required during an open operation when conflicting file sessions exist. The opening contact is required to notify all conflicting contacts, which will require an extra phase if all of the contacts are not included in the first notification phase. The opening contact will not send the file to the client machine for caching when write-sharing exists. Each contact will send a call-back request to any of their clients that have the file cached. Upon receipt of a call-back request, a client will delete the associated cache entry and send any cached updates for the file back to its contact.

4.1.2 Failure of Opens

In order for an open to succeed there are certain conditions that must be met. First, a majority of the servers must participate in the open process. If less than a majority participated, and the open were allowed to succeed, inconsistent file copies, or accesses to stale copies could result. For example, if two conflicting sessions start and the two contacts notify non-overlapping subsets of the servers, each session's contact will not know of the other file session, updates will not be propagated to both of the contacts, and the file copies at the two contacts will become inconsistent.

1) a majority of the servers must participate

2) all conflicting contacts must participate

3) a current copy of the file must be available

Figure 4-2: Necessary conditions for a successful open.

Every conflicting contact, if any, must participate in an open operation. In order to avoid the aforementioned inconsistencies, each updating contact must know of all contacts that exist for the file. If an updating contact does not know about another contact, the updating contact will not include the unknown contact in any write requests. The unknown contact would then be left out-of-date and would return stale data when fielding read requests, violating UNIX semantics.

The opening contact must be able to acquire a current copy of the file. If the opening contact cannot acquire a current copy of the file, the contact will not be able to perform any file requests on the file. Given that no file requests will succeed, there would be no reason to allow the open to succeed. The open could be allowed to succeed, but subsequent accesses would only fail, resulting in the client wasting time and resources on failed file operations in the future.

4.1.3 Alternate Design Choices

A read-one-write-all (ROWA) scheme could have been used, and the read performance would have been better, but the availability might have suffered. The availability, as measured on a per file session basis, would have improved as only one file server would be required for the majority of file sessions (i.e., read sessions), even though updating file sessions would have worse availability than in a non-replicated system. The problem is that, although the availability of file sessions would be improved, the ability to complete tasks would decrease. This is due to an assumption that most tasks that a user would want to perform, require updating at least one file. Since the ability of a user to perform a task is dependent on the lowest availability of all file sessions required, the ability of the user to perform tasks would decrease in a ROWA system.

A quorum based scheme could also have been adopted. This would require the contact to obtain a read-quorum or write-quorum of responses from other servers, during an open operation, depending on the file session type. We chose not to use this method, as our protocol is aimed at smaller replicated environments and with 3 servers a quorum based approach is either ROWA or uses majorities. When using 5 servers (e.g. Qw=4, Qr=2), and multicasting, the difference is a single message per read-only file session (i.e. the response message from a single server). This increase in efficiency is achieved at the cost of lowered availability in that, only 1 server failure could be handled without the loss of write availability, in contrast to two server failures, when using majorities.

4.2 Close

When a file session is closed, the agreement between the contact and the client ends. It is at this point in most file sessions that updates are propagated by the agent back to the contact.

1) The client requests a close, and the agent sends any cached updates back to the contact.

2) The contact accepts the updates, and returns the call to the agent

3) The contact propagates any updates to some, or all, of the other servers

Figure 4-3: Basic steps of a close operation

The agent sends the close request and any cached updates to the contact. Once the contact has appended the updates to the write log and noted the close in its state, it returns the call to the agent. The contact then propagates the close notification and the file, if it was modified, to some of the other servers. Updates will be applied to disk in the same manner as they would be in the case of a write operation that involves the contact.

4.2.1 Propagation of Updates and Close Notification

The result of the close operation is returned to the client before the contact propagates the **close notification** to the other servers. When the other servers receive the close notification, they note this in their state. This reduces the delay of a close noticed by the client process. If the client was made to block until propagation was complete, the client would be blocked unnecessarily. By not blocking the return to the agent, the client is allowed to continue processing, and the semantics of the file system are not affected.

Delaying the propagation of the close notification and updates to the other servers, in order to return the call to the agent sooner, can negatively affect the availability of the file. Not propagating the current copy of the file immediately, slightly increases the amount of time that only one current copy of the file exists in the system. Also, by delaying the close notification, the knowledge that the agreement is no longer active is limited to one server. Subsequent file opens may not see the close if the contact fails at an inopportune moment. Both of these design choices decrease the availability of the file, but they decrease it minimally as the propagation of the information is only delayed long enough to return the close call to the agent.

The protocol does not specify how many sites must be updated during a close operation. Notifying a majority about the close operation ensures that all successful future open operations will notice that the agreement has closed without requiring the closing contact's participation. If an opening contact does not have a record of the agreement closure, then the closing contact may be required to participate, possibly slowing the open request or causing the open to fail when the closing contact is unavailable. Propagating an updated file to more servers improves the availability of future file sessions. Propagating to more than a majority also reduces the probability that a file transfer will be required during a future open.

If failures exist in the system when an update file session is closed, the new version of the file may not get propagated to a majority of the servers. Also, the system may be configured not to update a majority of the servers during a close operation. Therefore, an open operation may not be able to find a current version of a file, even if a majority of the servers participate. It is because of this that condition 3 in Figure 4-2 is required.

When to propagate a file is a hard question to answer. If a file is propagated on close, then the file becomes replicated and provides better availability. Propagation on close also removes the cost from the **critical path,** since, during an open operation, the client will be kept waiting if the contact has to bring its copy up-to-date. To counter this, if most sessions on a file are sent to the same contact, then propagating to other servers is a price paid solely for availability.

4.2.2 Failure of a Close Operation

A close operation may fail when the client's contact is not reachable, or when the contact fails during the close process. If either situation occurs the agent process will time-out waiting for a response and will initiate a change-of-contact operation. If the change-of-contact operation succeeds the agent will retry the close operation at the new contact. If the change-of-contact operation fails, the close operation will return failure to the client process.

4.2.3 Alternatives

In order to increase the performance of the protocol, the propagation of the file to the contact could be delayed. Multiple close operations could be sent to the contact at once, reducing the messaging overhead, and allowing short lived files to be deleted before being transferred. This would make close operations that much faster but at the cost of availability and semantics. If the client fails, or becomes partitioned from the majority of servers, no processing can be performed on the file as the only current copy resides on the client machine. If the client's cache is in volatile memory, and the client crashes before sending the modified file to the contact, all of the updates will be lost. This is a weakening of the semantics that we chose to guarantee with this protocol and is why we do not use this alternative.

4.3 Write

Write operations can only be performed using agreements opened in write or read-write mode. Write operations are performed by the client's agent on the cache if possible, otherwise write operations are sent to the contact. For most write operations, the file will be cached as the environment has a low level of write-sharing and most files are small enough to allow caching. The steps that a contact takes to perform a write operation are shown in Figure 4-4.

1) Multicast the write operation to all of the other contacts for the file (if any) with an initial timestamp T_0 (the value of local timestamp counter at the contact).

2) $\forall i,\ 1 \le i \le n$, where n = *# of conflicting contacts*, contact i inserts the write into its local write queue according to $max(T_i, T_0)$, associates T_i with the write acknowledgment, increments the local timestamp counter to $T_i = max(T_i, T_0)+1$, and acknowledges the write.

3) The originating contact accepts all acknowledgments for the write request and returns the call to the agent.

4) The contact sets the write timestamp to $T_W = max(T_0,\ldots,T_n)$ and the commit notice is broadcast to all contacts with the new write timestamp.

Figure 4-4: Basic steps of a write operation performed by servers.

When the file is not cached, the write is sent to the contact which inserts the write into its write queue and multicasts a write notice to all existing contacts (note that write notices are sent to the other contacts for the file and not to every server) to initiate the 2-phase commit. The other contacts insert the write into their write queues in step 2, and then return an acknowledgment to the originating contact. During the time from when a contact acknowledges a write, until the contact receives the result of the write, the write is said to be **uncertain**.

When all acknowledgments are received in step 3, the contact returns the call to the client. A write commit notice is then broadcast to all of the contacts in step 4. Each contact will apply the write to disk at some point after the write has been committed.

4.3.1 Serializing Write Requests

Concurrent write requests need to be applied in the same order at all of the contacts. To serialize the write operations, each write request is assigned a timestamp[1] and all writes are applied in timestamp order. When a contact receives a write request, it gives the write an initial timestamp T_0 and inserts it into the local write

1. We assume each server keeps a local timestamp counter and that no two servers issue identical timestamps (ensured by appending the server Id to the timestamp.)

queue. The contact then sends a write notice to all other contacts. On receiving a write notice, server i will create an acknowledgment for it which includes a timestamp T_i. Server i will insert the write into its write queue at the position dictated by the write's timestamp. The timestamp used when inserting the write into the write queue is $max(T_0, T_i)$. The acknowledgment will then be sent to the originating contact. Server i updates its write timestamp counter to equal $max(T_0, T_i)+1$ so the next timestamp issued at server i will be later than the timestamp of the write notice that was just processed.

Once all acknowledgments have been received, the originating contact sets the timestamp for the write to T_W, the latest of all the timestamps returned with the acknowledgments, including the original timestamp assigned by this site. The originating contact also updates its local write timestamp counter to T_W+1. T_W is the final timestamp for the write and determines the order in which the write will be applied to the file. The write is then moved to its correct location within the write queue. The call is returned to the client, and a write commit notice is broadcast to the other contacts. The other contacts in turn update their local write timestamp counters and readjust the write in their write queues so that its position reflects the final timestamp, T_W. Write requests are placed in the write queue immediately when received so that they will be noticed by the server when processing other write requests.

4.3.2 Application of Writes to Disk

Updates are put into the write queue so that they are applied in the correct order. The write queue is ordered by the timestamps so the writes are applied at each server in the order they appear in the write queue. A committed write can be applied once all writes before it in the write queue have been applied or aborted. Once a write is committed, it is guaranteed that no writes will be inserted in the queue before it that do not already exist there. In order for a new write to be inserted before a committed write, the timestamp must be earlier than that of a committed write. The timestamp of a write to be inserted in the queue will be greater than the local write timestamp counter when the committed write was inserted. Thus, the new write cannot be inserted in the write queue before it. The timestamp mechanism also guarantees that the write queue is ordered the same at all contacts.

4.3.3 Write Failure

A write operation is not guaranteed to succeed when failures are present. In order for a write operation to succeed, all of the contacts for the file must acknowledge the write. If the originating contact times out while waiting for an acknowledgment, the contact will abort the write, return failure to the client and multi-cast a write abort notice to all contacts.

4.4 Read

Read operations can only be performed during the life of an agreement for a file session that was opened in read or read-write mode. Most read operations are performed on a cached copy of the file by the agent. The file will be cached for most file sessions as the assumed environment has a low level of write-sharing and most files are small enough to fit into the cache.

When there are conflicting agreements or the file is too large for the cache, the file will not be cached. The client will then send all read requests to its contact. The contact will read the data and send the result to the client.

4.4.1 Serializing Read Requests With Write Requests

In some cases there may be a write operation in-progress that affects the outcome of the read operation. The contact will not perform the read if there are any uncertain conflicting writes. A write conflicts with a read if the portion of the file written overlaps the portion of the file read. In order to guarantee that the file becomes stable at some point, the contact will not acknowledge any write notices until the read is fulfilled or aborted. This is used in achieving UNIX semantics by serializing the read operation between write operations. Any write operation that has not yet been acknowledged by this contact is in-progress and the read will then be performed before it. Any write that has been committed at this contact has already been returned to the writing client, so the read will reflect it and be serialized after it.

If a conflicting write exists, and the contact is not notified about the outcome of the write within some time-out period, the contact will query the other contacts for the file to determine the outcome of the conflicting write. Requiring the involvement of the other contacts is due to the well-known problem of dependent recovery inherent in 2PC [BERN87]. If the outcome of the write is still unknown, then the contact that originated the write must not be in the same partition otherwise it would have known the result. However, the writing client might be in the same partition, so, the contact serving the read operation will initiate a change-of-contact operation for the contact that originated the write operation. If the change-of-contact fails, then the read request will fail and the contact will return an error to the client. This is a version of the well-known problem of blocking inherent in 2PC [BERN87].

4.4.2 Failure of Read Requests

There are a limited number of situations in which a read request will fail. Reads will always succeed when the file is in the client's cache. If the file is not cached, then reads will succeed if the contact is available, and no conflicting writes exist. If a conflicting uncertain write exists and its outcome cannot be determined then a read request will fail. If the reader's contact is not available, and a change-of-contact operation fails, then the read request will fail.

4.5 Flush

A client process does not have the ability to determine if a file is cached or not. Therefore, the client does not know whether a flush will actually perform a useful function. If a client requests a flush when the file is not cached, nothing happens. When the file is cached, the updates are sent to the contact. Upon receipt of the updates, the contact treats them as it would a write operation.

The ability to flush cache updates allows the client application to control the permanence of updates during a file session. Without the flush operation, the only method an application would have to guarantee permanence of updates, would be to perform a close and then re-open the file for subsequent accesses. This would not be an appropriate method to use as opening and closing a file are expensive operations.

A flush operation will fail if, there are cached updates, the client cannot communicate with its contact and a change-of-contact operation is not possible. The result of a failure to flush updates from a client's cache is that the updates will not be made stable. The updates are not lost and a flush can be retried at a later point in order to attempt to make the updates stable. If a flush fails, the client may still process operations on the client's cache as normal.

4.6 Seek

In providing a UNIX style interface, one must provide for random file access. The seek operation allows for this. For most seek operations all processing can be accomplished locally at the client site as the agent keeps track of each agreement's file pointer. In the case where the client application is using a seek operation to extend the file, a seek is a special case of a write operation, logically increasing the size of the file and zeroing the new area. A seek operation usually remains local to the client; only in those cases where the file is extended and not cached does a seek require server participation.

4.7 Common Case Processing

Our protocol is quite detailed in order to handle many complicated scenarios. However, the common case for each operation is quite simple. Given the assumed environment, with write-sharing being rare, most file sessions do not require any communication with other contacts. Partitions and server failures are also considered to be relatively infrequent, so failure processing is not part of an average file session.

An average file session starts with an *open()* that requires only a majority of the servers to be notified, and the contact will allow the agent to cache the file. Until the *close()*, all subsequent file operations (reads, writes, seeks) are quickly fulfilled by the agent using the cache. The *close()* of a normal file session, includes the agent transferring the file back to the contact, if it was updated. The contact notifies the agent that the close operation has finished, allowing the agent to return the *close()* to the client. The contact will then propagate the file to a set of servers, again, if the file was updated.

4.8 Example File Sessions

To help in visualizing the steps the protocol takes, this section will step through a few example situations. For all of the examples, assume that the file system consists of three servers, S_1, S_2 and S_3, two clients sites each with one client process, C_1 and C_2, and all file sessions are on file X.

4.8.1 Normal Case Example

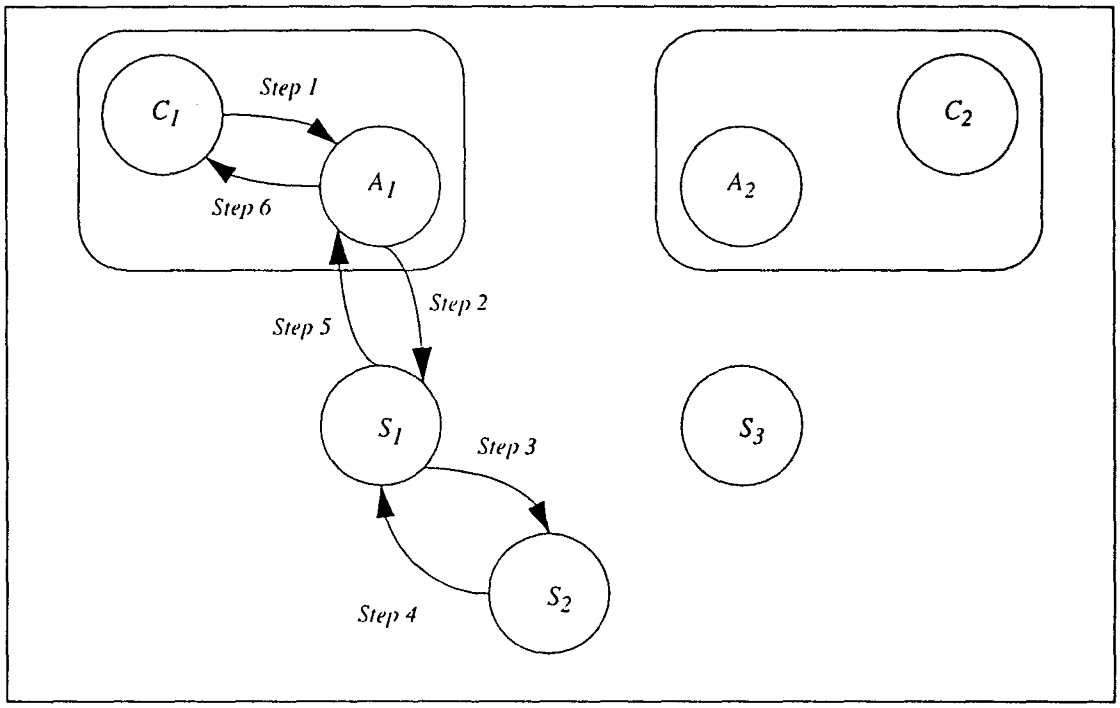

Figure 4-5: The normal processing of an open operation, when there are three servers.

The first example will be an average file session for reading a file, and an average file session for writing a file. Client C_1 will first perform a read-only file session on file X and then C_1 will perform a write-only file session, rewriting file X. The open operation is shown in Figure 4-5 taking 6 steps.

1) The client process performs an *open(X, read-only)* and the operation is intercepted by the agent, A_1.

2) A_1 chooses S_1 for the contact, sends the open request to S_1 and informs S_1 that the file is not cached.

3) The contact, S_1, sends the open notification to a majority of the servers, which, when including itself, requires one other server (the other server chosen is S_2).

4) S_2 responds back to S_1 with its file state for file X.

5) S_1 merges the state information, and determines that its copy of file X is current and that there are no conflicting agreements for file X. S_1 notes the agreement in its state, packages the file, and responds to A_1.

6) A_1 puts file X in the cache and returns the *open(X, read-only)* call to C_1.

The client, C_1, continues with the file session, performing 3 read operations on file X. Each read operation follows the steps shown in Figure 4-6.

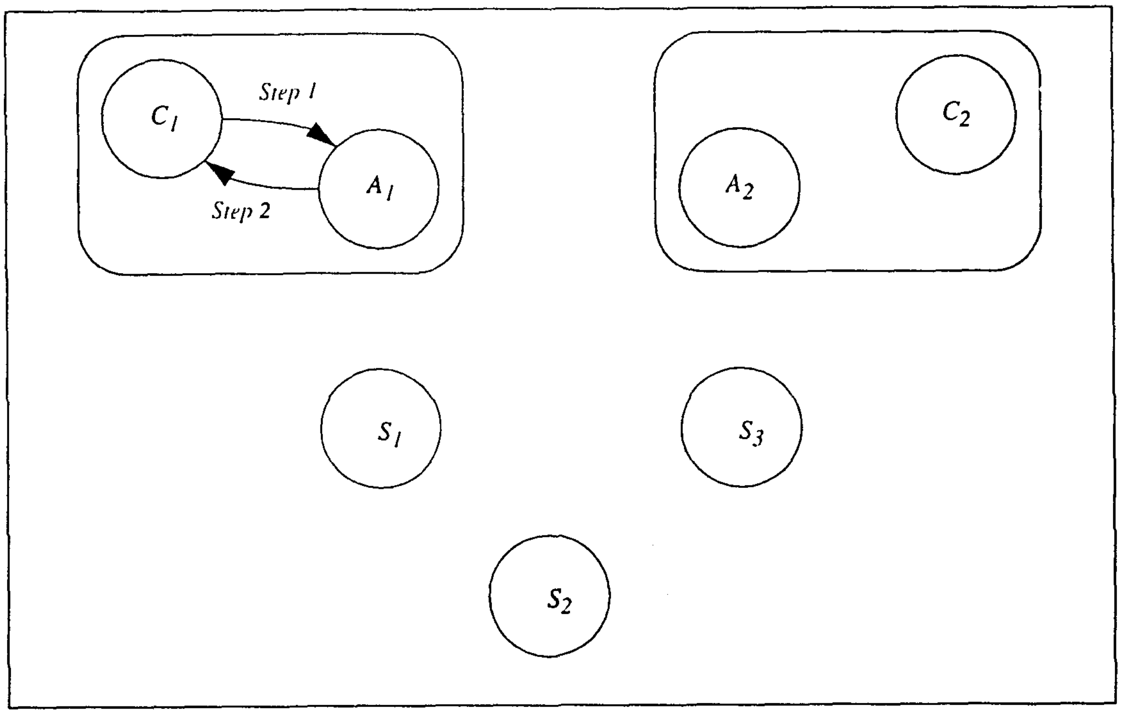

Figure 4-6: The normal processing of an read operations.

1) C_1 sends the read request to the agent, A_1.

2) A_1 performs the read on the cache and returns the result to the client.

After the client finishes accessing the file it requests a close operation which proceeds as depicted in Figure 4-7.

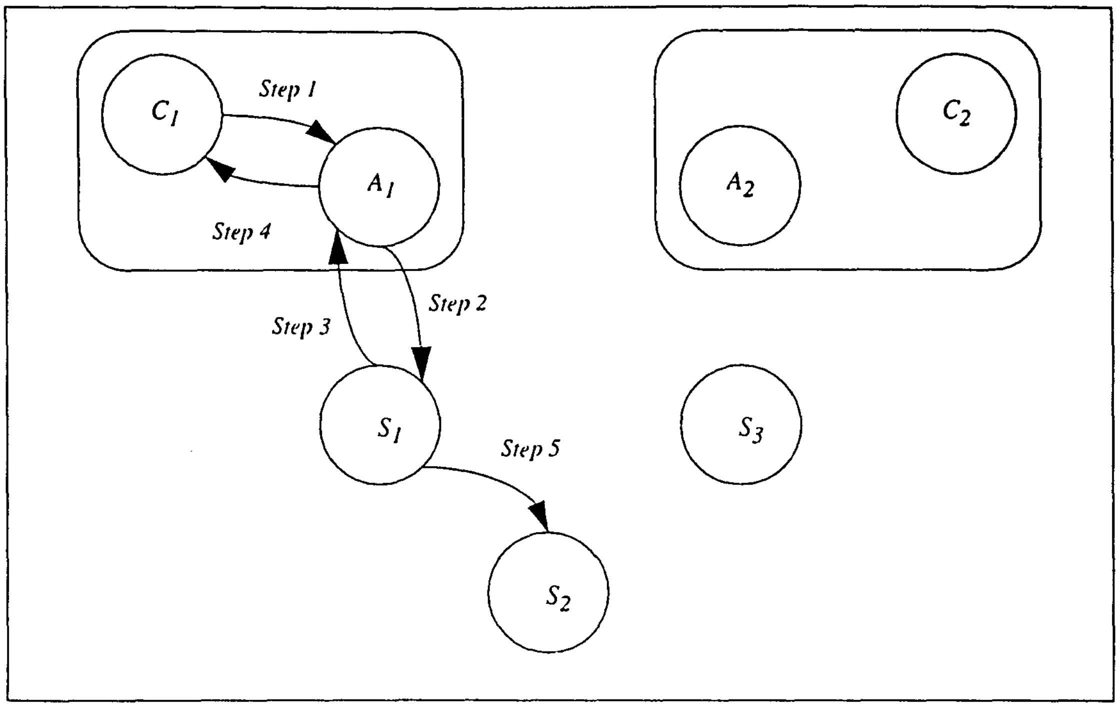

Figure 4-7: The normal processing of a close operation.

1) C_1 sends the close request to the agent, A_1.

2) A_1 sends the close request to the contact, S_1.

3) S_1 notes the close in its state, and return success to A_1.

4) A_1 returns success to the client.

5) S_1 sends notification of the close to all servers that know about the agreement. In this case S_2 is sent the notification and S_2 notes the close in its state.

4.8.2 An Example of Write-Sharing

In order to see exactly what occurs when conflicting sessions are present, this section steps through an example of write-sharing. The execution order is shown in Figure 4-8. In this example there are three file

1) C_1: *open(X, read-only)*

2) C_1: *read(X)*

3) C_2: *open(X, write-only)*

4) C_2: *write(X)*

5) C_1: *read(X)*

6) C_2: *close(X)*

7) C_2: *open(X, read-only)*

8) C_2: *read(X)*

9) C_1: *read(X)*

10) C_2: *close(X)*

11) C_1: *close(X)*

Figure 4-8: An example of sharing file sessions.

sessions chosen to highlight the actions taken during write-sharing and read-sharing.[1] The first two steps in the execution are performed as in the last example. The third step has C_2 opening the file; a break down is shown Figure 4-9.

C_2's open operation proceeds as follows:

1) C_2 performs an *open(X, write-only)* and the operation is intercepted by the agent, A_2.

1. Read-sharing refers to sharing of a file by processes that only read.

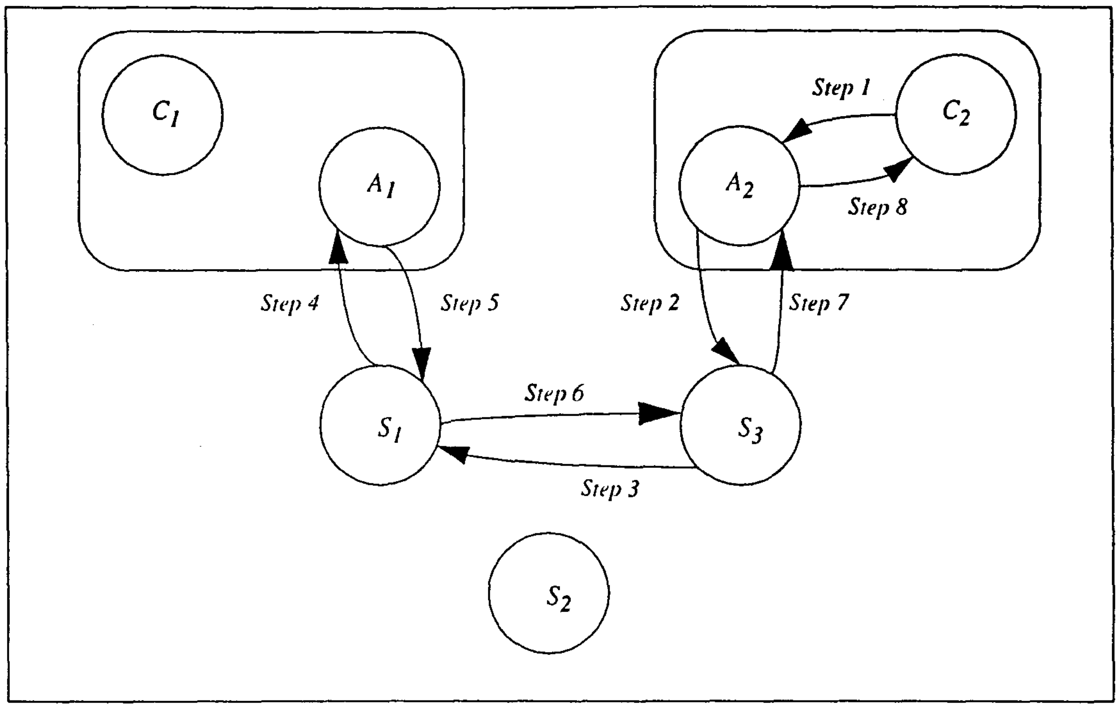

Figure 4-9: An open that creates a write-sharing situation.

2) A_2 chooses S_3 for the contact, sends the open request to S_3 informing S_3 that the file is not cached.

3) The contact, S_3, sends the open notification to a majority of the servers, which, when including itself, requires one other server (the other server chosen is S_1).

4) Upon receiving the open notification, S_1 checks its state and finds that it is the contact for a conflicting agreement. S_1 sends a callback request to agent A_1.

5) A_1 discards the cache entry, and responds back to its contact.

6) Once the callback completes successfully, S_1 sends its file state for file X to S_3.

7) S_3 merges the two states together and notices the conflicting agreement. A successful open result is then returned to the agent A_2 without a file to cache.

8) A_2 returns the successful result to the client process.

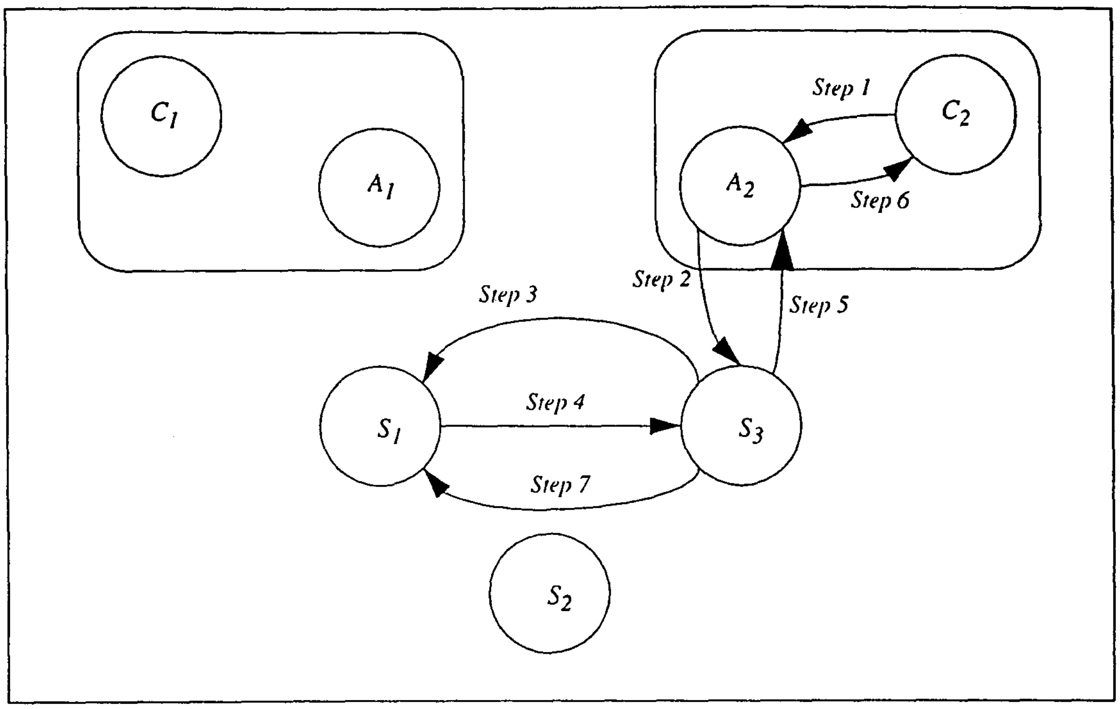

Figure 4-10: The steps taken by a write operation while in a write-share mode.

Next, C_2 performs a write operation. The steps performed to propagate the updates to both contacts are shown in Figure 4-10.

1) C_2 sends the write request to the agent.

2) The agent, A_2, finds that the file is not cached so the request is passed through to the contact, S_3.

3) S_3 assigns the write a timestamp, T_3, and inserts it into the write queue. S_3 then looks up all conflicting contacts in the file's state and determines that S_1 is the only other contact, and sends a write notice to S_1.

4) S_1 accepts the write notice, formulates an acknowledgment with a timestamp, T_1, and inserts the write notice to the write queue at $max(T_3, T_1)$. The acknowledgment is then returned to S_3.

5) S_3 accepts the acknowledgment, re-inserts the write at $max(T_3, T_1)$, and returns success to the agent, A_2.

6) A_2 sends the result to C_2.

7) S_3 sends a commit notice for the write operation to S_1.

The next operation performed is a read operation by client C_1. The operation is going to include the client's contact, S_1, and will involve ensuring that no conflicting writes are in-progress. In order to examine what will be done when conflicting writes exist, assume that the read operation is requested while the write operation that we discussed was still in the system. The steps in the read operation are shown in Figure 4-11.

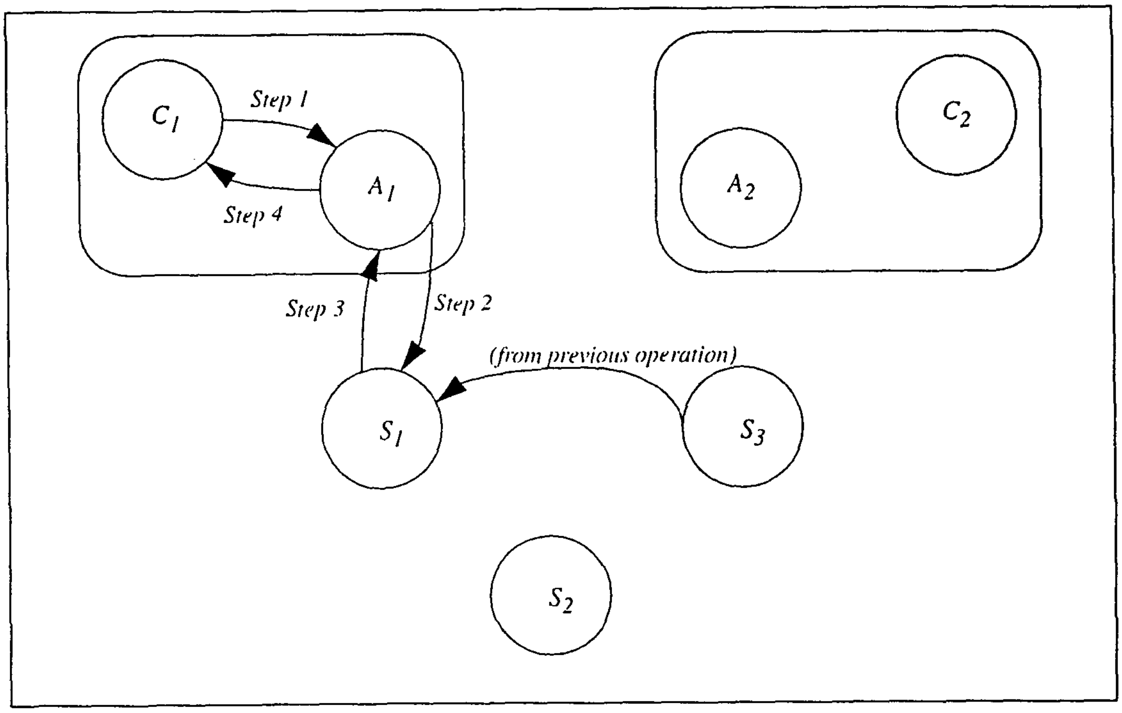

Figure 4-11: The steps taken by a read operation while in a write-share mode.

1) C_1 sends the request to the agent.

2) A_1 checks the cache, finds the file not cached, and sends the request through to the contact, S_1.

3) S_1 checks the write queue for conflicting uncertain writes, and in this case finds one. S_1 delays until the write result is received from S_3. S_1 returns the read result to the agent.

4) The agent passes the result to the client process.

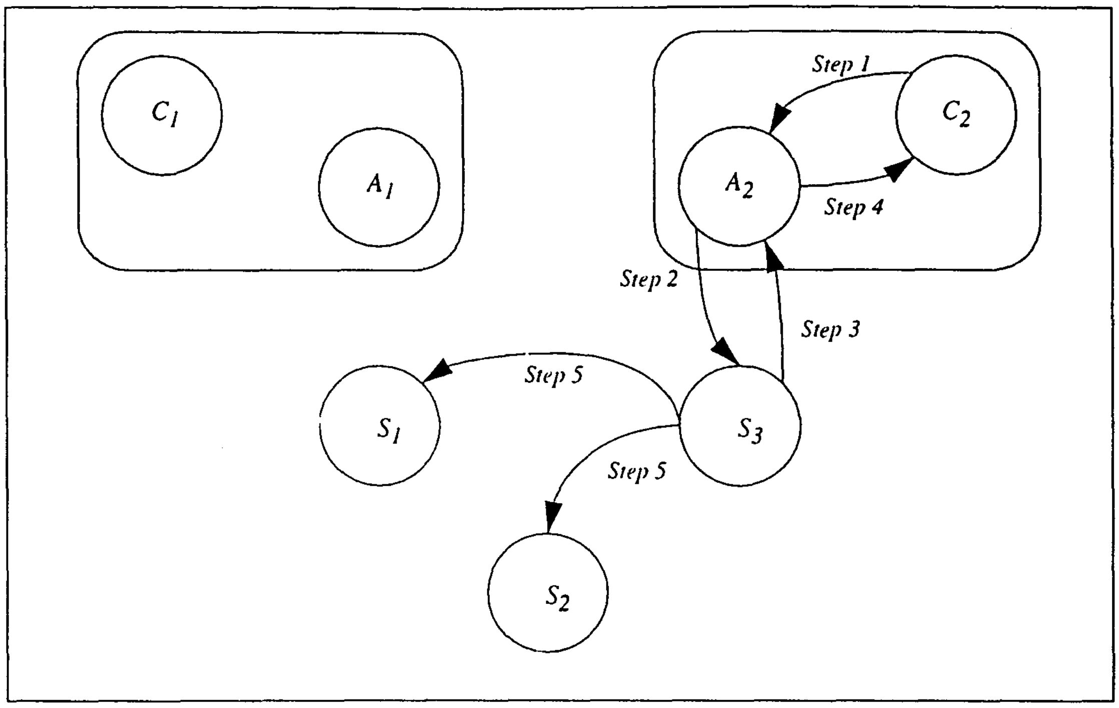

Figure 4-12: The steps taken by the close operation of a write session that ends a period of write-sharing.

The writing client, C_2, will now close its file session and the file will be propagated to the other servers as shown in Figure 4-12.

1) C_2 sends the request to the agent.

2) A_2 checks the cache, finds the file not cached, and sends the request through to the contact, S_3.

3) S_3 notes the close operation in the file's state and returns success to the agent.

4) The agent passes the result to the client process.

5) S_3 sends close notification to the other servers, and (optionally) sends the file to S_2. S_3 would not send the file to S_1 as S_1 is already up-to-date. The other servers note the close operation in their states.

Client C_2 will next re-open the file for reading. This open will proceed just as client C_1's open did at the start of this example. The open will result in client C_2 having the file cached, and the contact will be S_3. Next,

C_2, will perform a read, sending the request to the agent and having the agent fulfill the read on the cache. C_2 will then close the file in the normal manner as described in the previous example and shown in Figure 4-13.

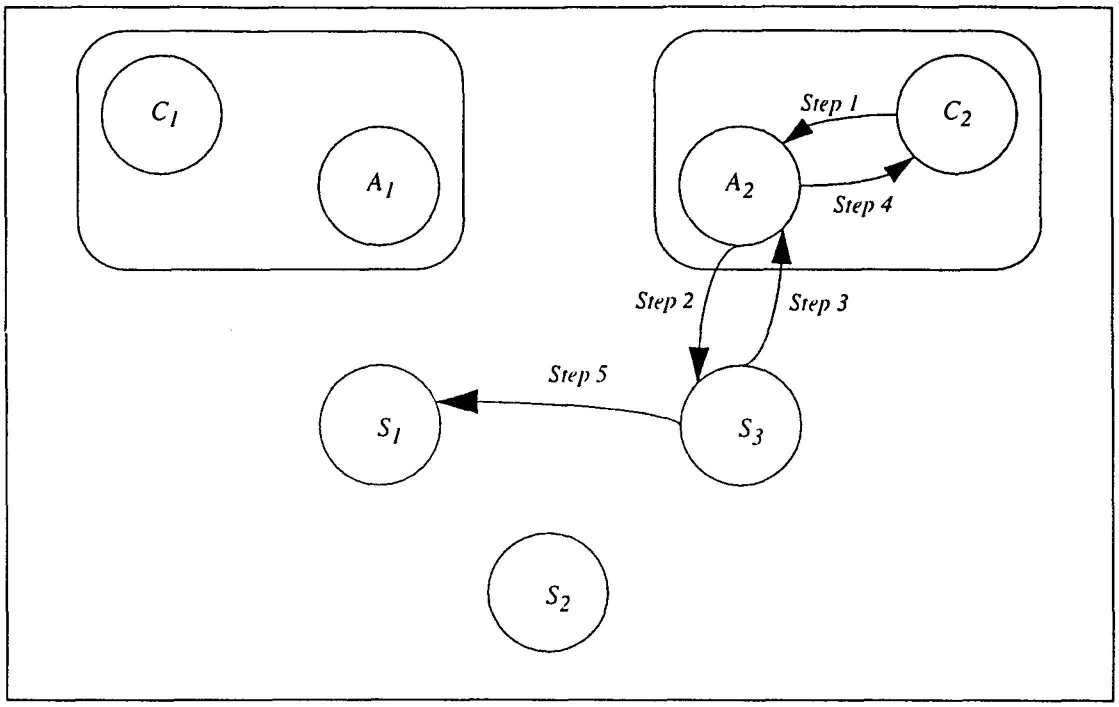

Figure 4-13: Client C_2 closing the read file session on file X.

Finally, C_1 will perform a read operation and then close the file. These operations will not be described but it should be noted that the read operation will be fielded by S_1 and that the file will not be sent to the agent to be cached (as files are only sent to client machines for caching during open operations.)

4.8.3 An Example Of Write-Write Conflicts

For the final example, let us examine what happens when two concurrent conflicting write operations are submitted at about the same time. Assume that our two clients, C_1 and C_2, have write-only sessions for file X. C_1's contact is S_1 and C_2's contact is S_2. Both clients issue a write request. Also, let the timestamp at S_1 be 10.1[1] and the timestamp at S_2 be 10.2. The discussion will pick up with each contact receiving the write request, and is shown in Figure 4-14.

1. A timestamp consists of {local count}.{server Id}.

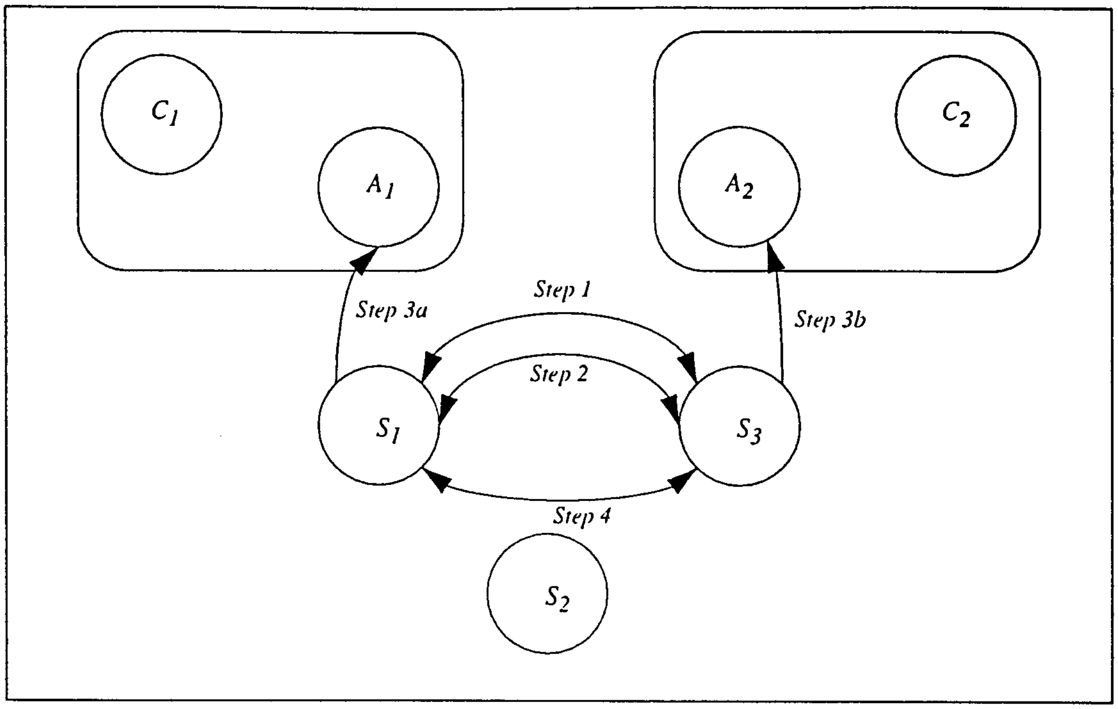

Figure 4-14: The steps taken when two conflicting writes enter the system at about the same time.

1) After receiving the write requests, the contacts timestamp them and insert them into their write queues. Contact S_1 timestamps the write, from C_1, 10.1 and updates the counter to 11.1. Contact S_2 timestamps the write, from C_2, 10.2 and updates the counter to 11.2. Each contact sends the write to the other contact.

2) Upon receiving the write notice from S_2, S_1 creates an acknowledgment and gives it the timestamp 11.1. S_1 inserts the write into its write queue at *max(11.1, 10.2)=11.1* and returns the acknowledgment to S_2. S_2 does this procedure for the write it receives, timestamping and inserting it using 11.2. S_1 and S_2 both update their counters to 12.1 and 12.2 respectively.

3) When each contact receives the other's acknowledgment, they re-insert the write at the higher timestamp; S_1 re-inserts its agent's write at 11.2, and S_2 re-inserts its agent's write at 11.1. They then return success to their respective agents.

4) Next the contacts create write-commit notices stating the final timestamps and send them to each other. The writes will be applied at some later point.

5 Failure Handling and Recovery

The main motivation for replicating a file server is to allow processing to continue even if some of the file servers are inaccessible. When a failure is present the replication protocol must take actions to facilitate continued service. Our protocol employs an algorithm called change-of-contact to tolerate server failures and network partitions.

5.1 Change-of-Contact

A change-of-contact operation can be initiated by an agent, or by a server. If an agent does not receive a response from its contact, the agent process will send a request to a new server to initiate a change-of-contact operation. Any server will suffice, but it is in the agent's best interest to choose a server that is physically close, as this server will become the new contact if the operation succeeds. Also, if a server notices an unresponsive server it may initiate a change-of-contact operation. A server will initiate a change-of-contact only if it requires a response from the unreachable server in order to complete an in-progress operation.

The purpose of a change-of-contact operation is to allow processing to continue when a server is inaccessible. If an agent cannot reach its contact, no file operations that require server participation can be performed for the affected agreement. In order to allow accesses to continue under the agreement, the change-of-contact operation attempts to modify the agreement to include a different server as the contact.

When a server initiates a change-of-contact operation because it requires an unreachable server to participate, the change-of-contact operation attempts to modify the agreement to include a different server so that the operation can continue.

If a server is inaccessible, it is probable that more than one agreement will be affected. If the failure persists, more than one change-of-contact would be required. For this reason, a change-of-contact operation attempts to change all of the agreements that a contact holds. In modifying all of the agreements with one operation, the cost of failure handling is reduced.

The basic steps involved in a change-of-contact operation are shown in Figure 5-1. The server, S1, that

Phase 1 1) Failure of server S0 is detected. A server, S1, starts the change process by communicating with a majority of servers to determine all of the agreements held by S0, and inform them of the change-of-contact operation
Phase 2 2) S1 notifies all agents of S0's agreements that a change-of-contact is in progress 3) S1 ensures all conflicting contacts are aware of the change-of-contact operation 4) S1 acquires the current copies of any files for which S0 had an agreement
Phase 3 5) S1 broadcasts the result of the change operation to all participating servers and agents

Figure 5-1: Steps and phases in a change-of-contact operation.

performs the change-of-contact operation is either the server that notices the failure, or the server chosen by the agent that notices the failure of server S0. In performing the change operation, S1 will attempt to have all of the agreements of S0 changed to show S1 as the contact.

In order to determine all of the agreements held by S0, S1 communicates with a majority of the servers. Each server that participates, responds with the file state for any file that S0 appears to have an agreement for, or for files that S1 has had an agreement for recently. Having received file states from a majority of the replicas, S1 can determine what agreements S0 holds because during the *open()* for each of the agreements, S0 made sure that a majority "knew" about it. The set of agreements is guaranteed to show all agreements held by S0, but may show some agreements of S0 that have been closed. This occurs for any agreement closures for which S0 did not inform a majority about and that are not implied closed by S0's lowest open agreement value.

From the set of agreements held, S1 determines all affected agents. Each of the agents is sent a notification of the change operation, and a list of that agent's agreements that are being changed. The agent will then acknowledge the notification. The agent will also send back information about the agreements held with S0, including whether the file is cached, if the cache is dirty, what operations the agent is currently performing on the file, and whether the agent believes that the agreement still exists.

S1 must also communicate with any contacts that hold agreements that conflict with those held by S0. This is done so that a conflicting contact can direct any communication for S0 to S1. At the same time, S1 should also acquire the current copy of each file for which an agreement is being changed. Both of these operations can be done while S1 is communicating with the agents.

Once S1 has become up-to-date, and the agents have responded, S1 can determine the outcome of the change operation for each agreement. It is important to note that a change-of-contact operation may only partially succeed. That is, the change may succeed for some of the unreachable contact's agreements, yet fail for others.

5.1.1 Necessary Conditions for Success

There are five conditions, shown in Figure 5-2, that must be met before an agreement's contact can be changed. If all of these conditions are met, the agreement's contact can be changed, but it does not guarantee that future operations will succeed. It is possible that the failed server may still be the contact for some conflicting agreements so write operations will fail.

The first condition for a change operation's success is that the agent participates. If the agent cannot be contacted, the change will not be successful. If the change were allowed to proceed without the agent's knowledge, the agent and the original contact could be in another partition performing file system accesses.

A change-of-contact would allow conflicting sessions to operate on the file in a different partition, causing violations of the consistency guarantees.

1) The agent participates

2) A majority of the servers participate

3) S1 acquires a current copy of the file, or the agent has the file cached

4) No uncertain write requests exist, for the agreement to be changed, for which the outcome cannot be determined

5) No other server is trying to change the agreements of S0 and started before S1

Figure 5-2: Requirements for a successful change-of-contact operation. The requirements are on a per agreement basis.

The second condition requires a majority of servers so that the new contact can collect all necessary agreement information and a majority of the servers know of the new contact. This guarantees that subsequent opens will notice the change-of-contact.

Either the agent has the file cached, or the new contact is required to obtain a current copy of the file. This is required so that operations can continue as they did before the change operation. Without a current copy of the file, read and write requests cannot be performed.

The fourth condition in Figure 5-2 refers to uncertain write requests. There must not be any in-progress write operation that were started by the contact that is unreachable, for which all other contacts have sent an acknowledgment, and none know the result. In this case, the change will fail as the state of the file is non-determinable.

The last requirement is in place to stop multiple change-of-contact operations from being performed at the same time for the same contact. Each change-of-contact operation is given a timestamp so that each change-of-contact operation can be distinguished and ordered. If during the execution of a change operation, the server performing the operation notices an earlier change that is in progress, the server will abort the operation.

5.1.2 Cost of a Change

This operation is significantly less expensive than a view change in a primary-copy/site system. No election is required, few files need to be transferred, and file system functions are not halted while the failure is being dealt with. Only one server may be required to have files brought up-to-date, and only a small fraction of the file system is affected. From the file system traces that we examined ([MUMM93]), a maximum of 16 concurrent file sessions were noticed. Also, it is trivial to determine what files need to be updated; unlike Deceit, Echo, and Harp, where the whole file system or file group needs to be checked. Given that only a few sessions are usually being served at any given time, the number of client machines that need to be contacted and the number of files that need to be updated is minimal.

The whole operation takes 3 phases to complete. During this time, the file system allows operations on non-affected files to continue. The change operation may slow down responses to agents due to the extra server processing and network traffic, but this is better than halting all operations while servers "exercise their democratic right."

Due to the assumed low level of write-sharing, the necessary conditions, shown in Figure 5-2, do not impose a heavy penalty on the success of failure handling. Conditions 3 and 4 only restrict failure handling when there are conflicting agreements. Therefore, most change-of-contact operations only require that the agent be in the majority partition.

5.2 Contact Recovery

When a server recovers from a crash it does nothing other than start where it left off. In contrast to a primary based scheme that starts a view change, or to Coda that updates the files on recovering servers, this protocol's server recovery is rather simple and efficient.

5.2.1 Contact Recovery Without Non-Volatile Memory

If the system does not provide extra hardware, i.e. UPSs and battery backed RAM, then the recovery of a server may become more complicated. The contacts can keep the file state stable by applying state changes to disk before acting upon the state change. This will allow all file state to be available to the contact upon recovery, but does impose an overhead whenever a server participates in an operation. An alternative method can be used that provides better efficiency, but handles fewer failure conditions.

If the server does not keep the file state stable, with the exception of the write queue, then parts of it will need to be rediscovered when a server recovers. Due to the distributed nature of the protocol, this can be achieved by communicating with other servers. The actions taken by a recovering server, S0, that did not keep its state stable, are:

- re-apply the entries in the write queue which is assumed to be stored on disk,
- query a majority of servers, that still have state information, to determine what agreements are held by S0, (note that the majority in this operation will not include the recovering server as it has no state information.)
- query a majority of servers to determine all active agreements, and relevant change-of-contact operations,
- query all clients of S0 to determine possible agreements that have closed,

The server will start reapplying the writes in the write queue to disk. There may be updates in the queue that have been committed, yet have not been applied. Those that the server is not sure about will be reapplied. The following actions will not wait for the file to become up-to-date before starting.

The second action allows the recovering server to reconstruct the list of agreements that it holds. This will also serve to notify the recovering server of all change-of-contact operations for which it was the old contact; in the assumed environment all of the recovering server's agreements would most likely have been changed to include a new contact.

The third action informs the recovering server of all active agreements. This is done to guarantee that the recovering server is notified of all agreements that it had participated in; ensuring that a majority of the servers know about all agreements in the system. If the recovering server did not rediscover what operations it has participated in, some agreements or change-of-contact operations would not be reflected at a majority of the servers. This information is retrieved at the same time, (with the same broadcast,) that the server finds out about its own agreements.

The last step allows the server to find out about agreements that were closed, where the closure was not known by the other servers. This step also allows the server to notify all of its clients that it has recovered, and can now fulfill requests.

During the rediscovery phase of the recovery process, the server will not participate in any file system operations. The recovering server will wait until its state is complete. If the server were to participate before its state is complete, it would be possible for open operations, or change operations to be performed without noticing all relevant agreements.

The requirements for this type of recovery to succeed are listed in Figure 5-2. The second requirement

1) a majority of the servers are available, excluding recovering servers as they do not have state information to share

2) less than a majority of the servers failed within *T* seconds, where *T* is the amount of time required for a server to rediscover its state.

Figure 5-3: Requirements for server recovery when file state information is not kept stable.

highlights an important fact. This method cannot handle the "near-simultaneous" failure of a majority of the servers. If a majority of the servers do fail within the time period *T*, then a non-recoverable loss of file state information can occur. The time period *T* is the amount of time required for a server to recover its file state.

5.3 Client Site Failure and Recovery

There are two cases to consider: failure of client machines with stable caches, and failures of client machines with unstable caches. When a client machine with a unstable cache crashes, all agreements are implicitly closed. Any updates that have not been sent to the server are lost. On recovery the agent process sends a message to each server stating that all agreements with the agent should be closed. When a client machine with a stable cache crashes, the agreements will remain in effect until the machine recovers, at which point the agent process will close all of the agreements and pass any updates in the cache to the contacts.

5.4 Availability Comparisons

Another important yardstick with which to measure the different file systems is the availability achieved by each. This is where our protocol pays the highest cost for ensuring UNIX semantics.

In choosing to enforce UNIX semantics, certain levels of availability are immediately unattainable. For example, one cannot allow updates to a file in one partition while any accesses are allowed to the file in another partition. One of the design decisions we chose was to benefit the reader, instead of the writer in times of conflict and failures. In doing so, we allow readers, that are able to open a file, to always read the file, if they can access a current copy, and the read does not conflict with an uncertain write. On the other hand, a writer is penalized in that the file is only available for update if all contacts for the file are available. This was chosen as update sessions make up only 10-20% of the file sessions ([MUMM93], [BAKE91]).

We have attempted to maximize availability without compromising the average file session cost. To this end, we have allowed any agent that obtains a cached copy to always access that cached copy unless the agent's contact performs a call-back. This lets all accesses to a agent's cache go unhindered by any need to check that the cache is still valid by contacting a server, or by having the server periodically contact the agent. On the other hand, the failure of a client machine can cause the loss of availability of a file until the computer is restored. The loss of availability is minimized by the requirement that the agent notify the contact on close, and flush any updates to the contact on close. This lowers the amount of time the availability of the file depends on a single site.

In the same manner as requiring agents to flush updates to the contact immediately, the contact tries to distribute these updates as soon after the close as is possible. The distribution is left until after the close call is returned to the agent so as to not delay the client while replication is achieved. The contact distributes the file to all available servers after returning the call to the agent.

To compare the achieved availability, the requirements for an operation to succeed are listed in Table 5-1 for each of the file systems is considered in this thesis.

To open a file in a normal situation when there are no conflicting sessions, our protocol requires that the agent be in a majority partition. This is the same requirement for Echo and for Harp. Only Deceit lets file sessions start in non-majority partitions, but Deceit also has the weakest consistency guarantees. Using our protocol, all conflicting contacts must also be in the majority partition along with the client site requesting the

Table 5-1: The Requirements For Availability Of Each Operation By File System

OPEN	
Our Protocol	1) the agent is in a majority partition
	2) all conflicting contacts are also in the partition or 2) all conflicting agents and 1 contact are in the partition
	3) there is a current version of the file available
Echo	1) the agent is in a majority partition
Deceit	1) the agent is able to contact a server
Harp	1) the agent is in a majority partition
CLOSE	
Our Protocol	1) the agent can reach its contact or 1) the agent and a current copy of the file are in a majority partition
Echo	1) the agent is in a majority partition
Deceit	1) the close is for a read-only file session or 1) the token holder is available
Harp	1) the close is for a read-only file session or 1) the agent is in a majority partition
READ	
Our Protocol	1) the agent has the file cached or 1) the agent's contact is available or 1) all conflicting agents, 1 contact, and a majority of the servers are in 1 partition

Table 5-1: The Requirements For Availability Of Each Operation By File System

Echo	1) the agent has the file cached and has not given up its read token or 1) the agent is in a majority partition
Deceit	1) the agent has the file cached or 1) the agent can contact a server
Harp	1) the agent has the file cached or 1) the agent is in a majority partition
WRITE	
Our Protocol	1) the agent has the file cached or 1) all contacts are available or 1) all conflicting agents, 1 contact, and a majority of the servers are in 1 partition
Echo	1) the agent has the file cached, has not given up its token, and flushes the updates from the cache before any conflicting agent gains the token or 1) the agent is in a majority partition
Deceit	1) the agent has the file cached or 1) the agent can contact a server
	2) the token holder is available when the updates are to be flushed from the cache
Harp	1) the agent has the file cached or 1) the agent is in a majority partition
	2) the agent is in a majority partition when the updates are to be flushed from the cache

open operation and all client machines that have the file cached. Requiring this limits our protocol's achieved availability during times of write-sharing, but since such scenarios are very rare, they do not drastically affect the overall availability.

Most read and write operations will be performed on cached copies in all protocols. Our protocol sacrifices availability during write-sharing in order to achieve the strict semantics. Write operations must be performed at all of the conflicting contacts in order to complete, or there must be enough servers, agents and contacts available to perform a change-of-contact operation. Read operations in write-shared situations are slightly better, requiring only the agent's contact be available or enough agents and contacts in order to perform a change-of-contact operation.

The failure of a client machine, or the partitioning of client machine appears to impose severe limitations on the availability that our protocol can achieve. This is not so. Only those files that a client machine is using will be affected by the inaccessibility of the client. Most files that other clients will desire are read-shared system files, and since no one is updating them the failure will not affect their availability. Those files that the inaccessible client may update will usually be personal files that no other clients will require. In the event that a client machine fails and the user wishes to switch to an alternate client machine, system support should be able to mark the failed client machine as crashed, allowing the servers to close all agreements held by the machine.

6 Comparisons

6.1 Operation Costs

This chapter presents a qualitative comparison of our protocol against Echo, Harp, and Deceit. Coda and the mobile system will not be included in the comparison as their semantics are sufficiently different that a comparison would not be useful. These comparisons are approximate, require a number of assumptions and are only presented to give the reader a general idea of how our protocol might perform in comparison to other replicated file systems.

The comparisons estimate the maximum expected costs for each of the basic file operations. For each operation, the different functions that a server and/or client may have to perform are listed with the cost of each.

In order to make the comparisons, we had to make some assumptions about the other file systems. The literature on Echo does not provide sufficient insight into what the file system performs during an open or close operation. The exact style of caching is also not covered. To overcome this, we assumed that an open operation entails acquiring the appropriate token, and fetching file data for the client's cache. A close operation is assumed to return the token and flush any cached updates to the primary. Since we did not know the caching style of Echo, we compared Echo using both whole file caching and disk block caching.

Harp and Deceit are both NFS server replacements and as such do not directly support open or close operations. For these two file systems, we assumed that an open would cause a fetch of file data, and a close would cause a flush of dirty cache entries. The flushing of dirty cache entries would be done outside of the critical path after the close call has been returned to the client. Both Harp and Deceit are assumed to use disk block caching.

Deceit allows a variety of operating modes. For the purpose of the comparisons, Deceit will be measured using the settings to achieve the strongest semantics it offers. This is so that it is similar to the other systems being compared. The Deceit settings are: no token generation, use stability notification, a minimum replica level of 1, no file migration, and a write availability level of low.

Deceit uses forwarding address to locate the token holder. In [SIEG90], this cost is stated to average to *log N*, where *N* is the number of servers. This value is used as the cost for finding the token holder in Deceit.

To describe the message costs performed by each file system, we broke down each operation into all the basic functions they perform. The functions and their associated costs are listed in tables 6-1 through 6-4. Each line in the tables shows what costs the file system would pay with and without hardware supported multicast.

Some of the operations are performed in the critical path, before the call is returned to the client; others are wholly or partially performed outside of the critical path. Message costs paid outside of the critical path are enclosed in square brackets in the tables. Costs paid outside of the critical path add to the network load and to the server's load, but do not directly affect the response time for the operation.

In order to determine the number of messages required, some constants are needed; these are shown in

N = number of replicas

F = number of client machines with existing, conflicting file sessions

C = number of existing, conflicting contacts

M = ceiling(N/2)

Figure 6-1: Constants used in the comparisons.

Figure 6-1. The number of existing conflicting file sessions, denoted by *F*, is the number of file sessions when a file is being write-shared. If the file is not being write-shared then *F* will be 0, even if there are concurrent file sessions reading the file. *F* is bounded by the number of client machines in the system, but will usually be small,e.g. less than 3. *C* denotes the number of contacts that are serving conflicting file sessions, and is bounded by *min(F, N)*.

The first four tables show the actions that can be performed by each file system operation. The message costs in Table 6-1 through Table 6-4, show the maximum cost for each action. For some of the operations, not all actions can take their maximum costs at the same time. For example, in our protocol, there are two actions shown for contacts performing cache call-backs. The total cost for these two operations is *2F*, but the maximum for each is also *2F*.

Read operations are effectively the same as open operations, except in our protocol. For this reason only our protocol is covered in Table 6-2.

The comparison of the different write operation costs is complicated. There are three issues: the cost of fetching the disk blocks into the client's cache, the cost of sending the updates back to the server and when these costs are paid. For our protocol, if the client's cache does not hold the required file, the update will be sent to the contact. The others will all fetch the required data, and send it to the client; this cost is the same as the open cost and so is not considered in Table 6-3. Therefore the table only shows the costs for updating the servers. In addition to the actual costs, the issue of when the costs are paid is also important. This will be addressed in the next section when session costs are discussed.

Table 6-1: Cost of Open Operations

OPEN	# Msgs w/o multicast	# Msgs w/ multicast
Our Protocol		
1) client sends msg to chosen contact	1	1
2) contact sends open notice to a majority	M-1	1
3) each server[a] recalls conflicting client caches	2F	2F
4) each server[a] responds to the contact	M-1	M-1
5) contact sends the open notice to all conflicting contacts not included in the majority	C	1
6) each contact[b] recalls conflicting client caches	2F	2F
7) each contact[b] responds	C	C
8) contact obtains a current copy of the file	2	2
9) the call and possibly the file are returned to the agent	1	1
Echo		
1) client sends msg to the primary	1	1
2) primary recalls all conflicting tokens	2F	2F
3) primary notifies all secondaries of each client that gives up its last token or receives its first token	(F+1)(N-1)	1
4) updates sent back by a conflicting client are sent to all secondaries by the primary	N-1	1
5) each secondary acks the updates	N-1	N-1
6) primary sends token and file data to client	1	1
Deceit		
1) client sends msg to the closest server	1	1
2) server finds token holder	log N	log N
3) server gets current file data from token holder	1	1
4) server sends file data to the client	1	1
Harp		
1) client sends msg to the primary	1	1
2) primary sends file data to the client	1	1

a. Each server that was included in step 2 performs this function; not all servers.

b. Each contact that was included in step 5 performs this function; not all contacts

Table 6-2: Cost of Read Operations
(When the read cannot be satisfied by the cache.)

READ	# Msgs w/o multicast	# Msgs w/ multicast
Our Protocol		
1) client sends msg to chosen contact	1	1
2) contact reads the data and returns the call to the client	1	1
Echo - same as open		
Deceit - same as open		
Harp - same as open		

Table 6-3: Cost of Sending Updates to the Servers

REMOTE WRITES	# Msgs w/o multicast	# Msgs w/ multicast
Our Protocol		
1) client sends msg to chosen contact	1	1
2) contact sends write notice to all contacts	C-1	1
3) all contacts ack the write notice	C-1	C-1
4) contact returns the write call to the client	1	1
5) contact broadcast write commit to all contacts	C-1	1
Echo[a]		
1) client sends msg to the primary	1	1
2) primary distributes updates to all secondaries	N-1	1
3) each secondary acks the updates	N-1	N-1
4) primary returns the write call to the client	1	1
Deceit[a]		
1) client sends msg to the closest server	1	1
2) server finds the token holder	log N	log N
3) token holder broadcasts a token pass	N-1	1
4) new token holder broadcasts instability notice	N-1	1
5) server sends the update to the other servers	N-1	1
6) server returns the call to the client	1	1
7) the other servers ack the updates	N-1	N-1
8) server broadcasts stability notice	N-1	1
Harp[a]		
1) client sends write to the primary	1	1
2) primary sends the update to the secondaries	N-1	1
3) all secondaries ack the update	N-1	N-1
4) primary returns the write call to the client	1	1

a. For Echo, Deceit, and Harp the costs in this table reflect the operations performed to propagate updates to the servers. The cost for pre-loading the cache is the same cost as for an open.

Table 6-4: Cost of Close Operations

CLOSE	# Msgs w/o multicast	# Msgs w/ multicast
Our Protocol		
1) client sends msg to chosen contact	1	1
2) contact returns the call to the client	1	1
3) contact sends close notice and any updates to all servers	N-1	1
Echo		
1) client sends the token and any updates to the primary	1	1
2) primary distributes updates to all secondaries	N-1	1
3) each secondary acks the updates	N-1	N-1
4) primary returns the call to the client	1	1
5) primary notifies all secondaries if this client gave up its last token	N-1	1
Deceit[a]		
1) client sends updates to the closest server	1	1
2) server finds token holder	log N	log N
3) token holder broadcasts a token pass	N-1	1
4) token holder broadcasts a token pass	N-1	1
5) server sends the update to the other servers	N-1	1
6) server returns the call to the client	1	1
7) the other servers ack the updates	N-1	N-1
8) server broadcasts stability notice	N-1	1
Harp[a]		
1) client sends any updates to the primary	1	1
2) primary sends the update to the secondaries	N-1	1
3) all secondaries ack the update	N-1	N-1
4) primary returns the call to the client	1	1

a. A close in Deceit and in Harp does nothing, but the costs of writing out the cache contents will be paid later, outside of the critical path.

6.2 Session Costs

Showing each operation's cost on its own does not completely describe the cost paid by a file system. The cost of an average file session, as computed from the costs of each operation, are shown in Figures 6-5 through 6-8. These are summarized in Figure 6-9. Figure 6-10 shows these costs with 3, 5, and 7 replicas.

The costs shown in the tables are for a normal file session; no error processing is included. For each protocol, the expected operations are listed with their associated costs. A normal file session will not conflict with any other sessions as the environment assumes a low level of write-sharing. Under whole file caching, the file is sent to the client on open, whereas with disk block caching the file is sent in pieces. For the purposes of the comparisons, we assume that disk block caching requires 2 fetches during an average file session in addition to the data transferred during the open operation. This definition of a normal file session has all protocols using client caching, and token based protocols only requiring 1 token acquisition. The calculations do not allow for file sessions that start with a file already cached.

The tables provide 3 types of measurements:

1) the cost in messages with and without hardware supported multicast,

2) the number of message rounds required, and

3) the number of file or data transfers required.

The data transfer measurement shows the number of times that the whole file (ft), or a portion of the file (dt) is sent between sites. These measurements are calculated for setups with and without hardware supported multicast. In Table 6-6, Echo's cost is computed for whole file caching and disk block caching; the main portion of the table gives the data transfer measurements for disk block caching.

Based on this cursory comparison, it can be seen that our protocol is only slightly more costly than the other protocols. Echo appears to be an excellent choice if whole file caching is used, but these comparisons do not show traffic caused by conflicting accesses, bottlenecks that show up at primaries, or the cost paid when a site fails or recovers. If these were factored in, Echo would not appear so efficient.

Table 6-5: Average Session Cost For Our Protocol.

		# Msgs		# Rounds	Data Xfers	
		w/o mcast	w/ mcast		w/o mcast	w/ mcast
Open						
1) client sends message to contact		1	1	1	0	0
2) contact sends open notice to a majority of the servers		M-1	1	1	0	0
3) a majority of the servers acknowledge the open notice		M-1	M-1	1	0	0
4) contact sends file to the client		1	1	1	1 ft	1 ft
Close						
1) client sends message (and updates) to contact		1	1	1	1 ft	1 ft
2) contact sends close result to client		1	1	1	0	0
3) contact sends close notice (and updates) to all servers		N-1	1	1	N-1 ft	1 ft
Average Session Costs	Read Sessions	2M+2+ [N-1]	4+M+[1]	6+[1]	1 ft	1 ft
	Write Sessions	2M+2+ [N-1]	4+M+[1]	6+[1]	2+[N-1] ft	2+[1] ft

Table 6-6: Average Session Cost For Echo.

		# Msgs		# Rounds	Data Xfers[a]	
		w/o mcast	w/ mcast		w/o mcast	w/ mcast
Open						
1) client sends message to primary		1	1	1	0	0
2) primary sends file data and token to client		1	1	1	1 dt	1 dt
Close						
1) client sends token (and updates) to primary		1	1	1	1 dt	1 dt
2) primary sends updates to all secondaries		N-1	1	1	N-1 dt	1 dt
3) all secondaries acknowledge the updates		N-1	N-1	1	0	0
4) primary returns result to client		1	1	1	0	0
Fetch (disk block caching only)						
1) client sends message to primary		1	1	1	0	0
2) primary sends file data to client		1	1	1	1 dt	1 dt
Average Whole File Caching Session Costs	Read Sessions	4	4	4	1 ft	1 ft
	Write Sessions	2N+2	N+4	6	N+1 ft	3 ft
Average Disk Block Caching Session Costs	Read Sessions	8	8	8	3 dt	3 dt
	Write Sessions	2N+6	N+8	10	N+3 dt	3 dt

a. The main section of this table shows data transfer measurements for disk block caching only. Whole file caching measurements for the open and close operations are the same except the units are *ft*s.

Table 6-7: Average Session Cost For Deceit.

		# Msgs		# Rounds	Data Xfers	
		w/o mcast	w/ mcast		w/o mcast	w/ mcast
Open						
1) client sends message to server		1	1	1	0	0
2) server sends file data to client		1	1	1	1 dt	1 dt
Write Back (after closing)						
1) client sends file data to server		1	1	1	1 dt	1 dt
2) server finds token holder		log N	log N	log N	0	0
3) token holder sends token pass to all servers		N-1	1	1	0	0
4) new token holder marks others as unstable		N-1	1	1	0	0
5) server sends updates to all servers		N-1	1	1	N-1 dt	1 dt
6) server returns call to client		1	1	1	0	0
7) the servers ack the updates		N-1	1	1	0	0
8) server sends stability notice to all servers		N-1	N-1	1	0	0
Fetch						
1) client sends message to server		1	1	1	0	0
2) server sends file data to client		1	1	1	1 dt	1 dt
Average Session Costs	Read Sessions	6	6	6	3 dt	3 dt
	Write Sessions	6+[5N-3+ log 3]	6+ [N+6 +log N]	6+ [7+log N]	3+[N] dt	3+[2] dt

Table 6-8: Average Session Cost For HARP.

		# Msgs		# Rounds	Data Xfers	
		w/o mcast	w/ mcast		w/o mcast	w/ mcast
Open						
1) client sends message to primary		1	1	1	0	0
2) primary sends file to the client		1	1	1	1 dt	1 dt
Write Back (after closing)						
1) client sends file data to primary		1	1	1	1 dt	1 dt
2) primary sends updates to all servers		N-1	1	1	N-1 dt	1 dt
3) all secondaries acknowledge updates		N-1	N-1	1	0	0
4) primary returns call to client		1	1	1	0	0
5) primary sends update commit to all secondaries		N-1	1	1	0	0
Fetch						
1) client sends message to primary		1	1	1	0	0
2) primary sends file data to client		1	1	1	1 dt	1 dt
Average Session Costs	Read Sessions	6	6	6	3 dt	3 dt
	Write Sessions	6+[3N-1]	6+[N+3]	6+[5]	3+[N] dt	3+[2] dt

Table 6-9: Average Session Cost Summary.

		# Msgs		# Rounds	Data Xfers	
		w/o mcast	w/ mcast		w/o mcast	w/ mcast
Our Protocol's Average Session Costs	Read Sessions	2M+2+ [N-1]	4+M+[1]	6+[1]	1 ft	1 ft
	Write Sessions	2M+2+ [N-1]	4+M+[1]	6+[1]	2+[N-1] ft	2+[1] ft
Echo's Average Session Costs (Whole File Caching)	Read Sessions	4	4	4	1 ft	1 ft
	Write Sessions	2N+2	N+4	6	N+1 ft	3 ft
Echo's Average Session Costs (Disk Block Caching)	Read Sessions	8	8	8	3 dt	3 dt
	Write Sessions	2N+6	N+8	10	N+3 dt	3 dt
Deceit's Average Session Costs	Read Sessions	6	6	6	3 dt	3 dt
	Write Sessions	6+[5N+ log N-3]	6+ [N+6+ log N]	6+[7+ log N]	3+[N] dt	3+[2] dt
HARP's Average Session Costs	Read Sessions	6	6	6	3 dt	3 dt
	Write Sessions	6+[3N-1]	6+[N+3]	6+[5]	3+[N] dt	3+[2] dt

Table 6-10: Average Session Cost Summary By Number of Replicas. Includes average for all sessions using a 90%/10% read/write mix.

		# Msgs					
		w/o mcast			w/ mcast		
		3	5	7	3	5	7
Our Protocol's Average Session Costs	Read Sessions	6+[2]	8+[4]	10+[6]	6+[1]	7+[1]	8+[1]
	Write Sessions	6+[2]	8+[4]	10+[6]	6+[1]	7+[1]	8+[1]
	All Sessions	6+[2]	8+[4]	10+[6]	6+[1]	7+[1]	8+[1]
Echo's Average Session Costs (Whole File Caching)	Read Sessions	4	4	4	4	4	4
	Write Sessions	8	12	16	7	9	11
	All Sessions	4.4	4.8	5.2	4.3	4.5	4.7
Echo's Average Session Costs (Disk Block Caching)	Read Sessions	8	8	8	8	8	8
	Write Sessions	12	16	20	11	13	15
	All Sessions	8.4	8.8	9.2	8.3	8.5	8.7
Deceit's Average Session Costs	Read Sessions	6	6	6	6	6	6
	Write Sessions	6+[23.1]	6+[33.6]	6+[44]	6+[11.1]	6+[11.6]	6+[12]
	All Sessions	6+[2.31]	6+[3.36]	6+[4.4]	6+[11.1]	6+[11.6]	6+[1.2]
HARP's Average Session Costs	Read Sessions	6	6	6	6	6	6
	Write Sessions	6+[8]	6+[14]	6+[20]	6+[6]	6+[8]	6+[10]
	All Sessions	6+[0.8]	6+[1.4]	6+[2.0]	6+[0.6]	6+[0.8]	6+[1.0]

6.3 Conclusions

From the tables presented in the previous sections, a rough comparison of the different protocol's performances can be drawn. When comparing the relative message costs, one should consider that our protocol is providing stronger consistency guarantees and therefore is required to perform more work.

The tables show that our protocol achieves similar performance to the other protocols for a small number of replicas. As the number of replicas grows it can be seen, from Table 6-10, that the cost of our protocol increases faster than the costs increase for the other systems. This shows that achieving such strict consistency in large networks with our protocol is not feasible, but for "smaller" systems we are competitive.

It is interesting to note that by using whole file caching, our system provides the fewer number of data transfer messages. This is not to say that we transfer less data, but that we pay a lower overhead cost. The benefit should be countered with the fact that an open operation, when using whole file caching, can take longer than when using disk block caching, and that whole file caching will send more data than is required for some file sessions.

These comparisons leave out many performance issues: the effects of load balancing on the servers, bottlenecks in the system, costly failure handling, and the effect of distance on communication costs. Taking into account these factors would benefit our protocol more than others.

7 Implementation

This section describes the implementation of a prototype and gives some test results. The tests were developed to compare the performance of the prototype to NFS and to a non-replicated version of the file system. The comparisons were designed to approximate the costs of the replication protocol and the costs of achieving UNIX semantics.

7.1 Prototype

The prototype was implemented on NeXT workstations. The choice of platforms was driven by the simple IPC interface, and elegant threads package offered by the NeXT mach operating system. The distributed nature of the project, and the structure of the protocol required good communication, and process management primitives.

The prototype did not contain all of the functionality of the protocol. The basic file operations, open, close, read, and write, were implemented. The failure handling mechanism, change of contact, was not implemented. The hardware did not include any UPSs, nor did it include stable memory.

Each server used the local file system of the computer to store its copies of the files. Access to these files was through the standard file system open, close, read and write system calls. This allowed us to concentrate on implementing the protocol, rather than spending time on other unrelated issues such as a block service, or a directory service. There was also no server caching included other than that provided by the operating system.

The prototype included a client interface, agent process code, and server code. The agent process is made up of a number of threads to service client requests, and a number of threads to service file server requests. Client requests are all sent to the agent; those destined for the servers are passed on, the others are performed on the cache by the agent. The agent also accepts commands from the server to destroy or flush files in the client's cache.

The server code consists of a number of threads, all in a single server process. There is a set of worker threads that accept and fulfill agent requests, and another set to accept and perform requests from other servers. There is also a thread responsible for applying all updates in the write log to disk.

Each replica of the file system was implemented by executing a server process on a NeXT workstation, using the local disk for file storage. There were 1024 files, numbered 0 through 1023, all in a single directory thus providing a flat file system. The files used range in size from a couple hundred bytes to 15000 bytes. The state information for each file is not saved to disk in the prototype. Instead, each file starts with an empty state (but the file is not empty) and the state grows/shrinks as the system accesses the file. The file state is kept only by the servers, and does not get propagated to the agents. The only state information seen by an agent process is the version numbers of the cached files and the agreement id's that involve that client machine.

The non-replicated version of our protocol has the same agent interface as the replicated version. This version provides the same consistency guarantees that the replicated version offers. The single-server version has less file state information to track, fewer situations to consider, and therefore requires significantly less processing.

7.1.1 Session Generator

In order to test the prototype, we developed a simple file system session generator. The generator creates file sessions with an exponential inter-arrival rate. There are a number of inputs that control the creation of the file sessions, and the accesses that they perform. Among those inputs are: file session inter-arrival time, % read/write/read-write file sessions, # of operations per session, operation inter-arrival times, % of the sessions that read the whole file v.s. a single part of the file v.s. randomly selected parts of the file, and the % of the file sessions that operate in write-shared mode.

For each session, the generator spawns a thread that performs all of the file accesses for the session. In order to keep the model simple, each file session has a constant operation inter-arrival time, along with the amount of data read/written per operation. This allows the operations to be performed without using a random number generator during the file session. As the system is run in real-time, and the delays are not controllable, accessing the random number generator within the file sessions would result in elements of the random number sequence to be used for different instances depending on which test is performed. We wanted to use the same executions for testing each file system setup, and this allowed us to achieve that without using some form of trace file, which would have competed for disk resources with the tests we were running.

7.1.2 Test Variations

There are a number of different aspects to the protocol that were examined. The effects of caching, replication, and strong consistency semantics were tested, along with server load capabilities. In order to perform these tests we created different instances of the prototype. The test variations performed were: the standard protocol with and without caching, a single server version of the protocol, and an interface to NFS. These variations allowed us to examine the performance of the protocol with respect to replication and caching, and to compare the performance to an existing file system.

7.2 Test Runs

All of the test runs of our prototype and its variations were made with 6 NeXT machines, 3 clients and 3 servers. All but one of the machines was a standard 68040 based NeXT workstation with 16 Meg of memory. The final machine was a Turbo NeXT 68040 with 48 Meg of memory. The faster machine was used as a server in our tests. For the NFS test, we used 3 NeXT machines that issued requests to a NeXT NFS server on the same LAN that the workstations were on. In order to reduce any conflicts with other processing all of our test runs were performed at night during low usage times. The computers were all running version 3.0 of the NeXT Mach O/S and were connected by a 10 Mbit ethernet.

Each client machine produced more file sessions than a 'real' client machine would in order to simulate the load of many clients. Each of the three client machines produced $\frac{1}{3}$ of the total load per experiment.

The parameters used to generate the file sessions were obtained from [BAKE91]. The general settings for the type of file sessions are generated are shown in Table 7-1. To determine the number of operations per

Table 7-1: Types of sessions generated ([BAKE91]).

Constraint	Read Sessions	Write Sessions	Read/Write Sessions
Session Type	88%	11%	1%
Whole File	78%	19%	3%
1 Contiguous Portion	67%	29%	4%
Random	0%	0%	100%

file session we processed some file traces acquired from ([MUMM93]). These traces also confirmed the read/write/read-write file session mix reported in [BAKE91]. The values used for the majority of the tests are shown in Table 7-2. The average observed was about 8 operations per session, but this was due to a few

Table 7-2: Average operations per file sessions generated in the tests

Read	Write	Read/Write
3.5	11	15

sessions having a very large number of operations, the vast majority of sessions consisted of 2 operations or less as can be seen in Table 7-3 . Tests were also run with 25 operations per session for all session types to determine how file sessions with significantly more operations would be affected by the protocol.

Due to the amount of time required to perform the experiments, only one parameter was varied from test to test, the file session inter-arrival time. The file session inter-arrival time controls the number of file sessions that are present in the system at any given time. This determined the load placed on our servers. For our comparisons to NFS, and the non-replicated prototype under expected loads, we performed tests where each client generated file sessions with the inter-arrival times: of 1.0, 1.5, 2.0, 2.5 and 3.0 seconds. Given that the tests were performed with 3 clients generating this load, these values correspond to 3.0, 2.0 1.5, 1.2 and 1.0 file sessions per second. The Sprite paper showed loads of about 1.0 file sessions per second. Since this was averaged over the period of a day., we took this to be the low end of the load that we should test.

Table 7-3: Observed operations/file session by file session type.

# Operations / Session	Read Session	Write Sessions	Read/Write Sessions
1	31.2 %	25.1	8.38
2	51.6	27.3	57.9
3	9.04	5.29	4.10
4	1.96	1.61	19.7
5	0.51	1.25	0.58
6	0.72	21.1	0.90
7	0.32	1.61	0.84
8 or more	4.58	16.7	7.56
Average	4.96	33.3	29.2

In order to determine when the servers would start to thrash, we decreased the inter-arrival times. For the stress portion of the test, we only tested the replicated and non-replicated protocols; we did not stress-test NFS. The inter-arrival times that were used were: 0.8, 0.7, 06, 0.5, 0.4 seconds (i.e., 3.75, 4.3, 5, 6, and 7.5 file sessions/second.) We attempted runs with an inter-arrival time of 0.3 seconds, (10 file sessions/second,) but the non-replicated version could not handle the load and would crash.

The single server test was used to determine the cost our protocol paid to achieve replication. We decided the best way to show the cost of replication was to compare our protocol with a similar structured non-replicated file system. In doing so we were able to remove bias due to code optimizations of well tested, mature, commercial grade file systems (NFS), and the differences due to the kernel space processing of NFS v.s. the user space processing of our prototype.

The comparison to NFS shows how our model compared to a widely accepted file system with weak semantics. This comparison is subject to those biases mentioned above. The biases can be partially factored out by comparing the costs obtained by the NFS test, and those shown by our single server tests. The bias cannot be completely determined as there are some costs our protocol must pay in order to guarantee UNIX semantics.

7.3 Test Output

The most relevant data produced by the testing shows the achieved response times of the different setups; how efficient the system appears to client processes. A summary of each tests' output is shown in Table 7-4. Table 7-5 and Figure 7-1 show the results of the stress testing of the replicated and non-replicated protocols. The results of testing non-caching client machines are shown in Table 7-6. An experiment to examine the effects of file sessions with more operations per session is summarized in Table 7-7. The measured response times for each operation and the file session times, all in milliseconds, are shown. All of the client machines exhibited roughly the same behavior, so rather than show the results generated by all of the clients, only those measurements collected by one of the client machines are summarized in the tables.

For all of the tests, NFS performs significantly better than both the replicated and the non-replicated protocols. This is true not only for open and close operations, that always include server participation, but also for write operations that only include the servers some of the time. We believe that this is due to optimizations achieved through years of fine tuning NFS, and due to the fact that NFS executes in kemel space, requiring fewer transfers between user space and kernel space.

From Table 7-4 it can be seen that under the expected load, the cost paid for replication is completely contained within the cost of opening the file. This is as expected as most reads and writes are fulfilled using the cache. The close operation for the replicated and non-replicated cases are effectively the same, as the cost of replicating updates and notifying servers are performed outside of the critical path. The extra cost paid on open ranges from 17 to 28% of the open cost, or 6.4 to 16% of the total session cost. The confidence intervals for the data produced are also shown in Table 7-4. The results are accurate for the lower load levels, but the confidence intervals drop off as the load increases. This is due to a lack of time in which to run enough tests to tighten the confidence intervals at all load levels.

The close operation costs are effectively the same in the replicated and non-replicated protocols. There is also no difference between the read and write operation times under normal loads. The timing of sessions shows the amortization of the open cost over the life of a file session. The session times include the time between file system operations when the client process is performing other functions. This is acceptable in that each protocol tested was run using the exact same set of file sessions, and therefore the same amount of extra processing time was included in the session time measurements for each system.

Table 7-4: Response times for expected loads.

LOAD	REPLICATED		NON-REPLICATED			NFS		
		Conf. Int		% below	Conf. Int.		% below	Conf. Int
OPEN								
3.0	98.8	4.96%	81.6	17%	13.21%	28.9	71%	11.14%
2.0	97.9	4.41%	75.3	23%	13.12%	27.1	72%	14.89%
1.5	107.5	3.95%	77.9	28%	4.16%	31.3	71%	11.63%
1.2	105.2	2.81%	81.2	23%	2.67%	29.3	72%	15.57%
1.0	106.4	2.73%	83.2	22%	6.59%	31.0	71%	12.09%
CLOSE								
3.0	29.3	2.48%	34.0	-16%	5.30%	13.2	55%	5.02%
2.0	27.6	0.20%	32.5	-18%	5.28%	11.9	57%	4.98%
1.5	26.0	0.62%	29.8	-15%	3.77%	12.1	53%	2.54%
1.2	25.5	0.75%	29.7	-16%	4.66%	11.9	54%	12.94%
1.0	24.8	1.40%	26.6	-7.3%	7.88%	12.1	51%	7.01%
READ								
3.0	3.3	0.00%	3.6	-9.9%	3.67%	2.3	31%	6.57%
2.0	3.4	0.33%	3.3	4.2%	1.57%	2.4	30%	25.02%
1.5	3.1	0.05%	3.2	-2.6%	5.36%	2.6	15%	8.21%
1.2	3.0	0.06%	3.0	-0.50%	3.38%	2.8	7.4%	16.21%
1.0	3.0	0.03%	3.0	-1.7%	2.30%	3.2	-6.9%	9.00%
WRITE								
3.0	5.1	1.33%	5.8	-13%	30.12%	2.4	52%	19.45%
2.0	4.9	0.73%	4.6	7.0%	7.82%	2.6	48%	18.62%
1.5	4.4	0.38%	4.2	4.1%	10.96%	3.1	31%	10.93%
1.2	3.9	0.28%	3.8	3.1%	6.59%	2.8	28%	21.02%
1.0	3.6	0.24%	3.5	2.7%	11.33%	3.3	9.0%	8.17%
SESSION								
3.0	153.9	4.00%	144.0	6.4%	7.12%	58.9	62%	5.87%
2.0	151.9	5.75%	132.8	13%	6.96%	56.0	63%	13.10%
1.5	157.2	3.92%	131.9	16%	1.60%	62.2	60%	5.11%
1.2	154.0	3.13%	134.1	13%	1.60%	60.6	61%	8.52%
1.0	153.6	2.42%	132.5	14%	4.23%	65.2	58%	8.93%

In order to test how the servers respond to increasing loads, the load was increased to 7.5 file sessions per second for the replicated and non-replicated protocols and is shown in Table 7-5. The replicated protocol was fairly constant in its behavior up to a load of 5 file sessions per second. Its performance dropped significantly once the load was increased beyond 5 file sessions per second. In contrast the non-replicated protocol remained constant only to about 3.75 file sessions per second and then exhibited a slower loss in performance.

There were not enough test runs performed for the higher load tests in order to make them statistically accurate as the machines being used were reaching their processing limits and would crash. Tests were attempted for 10 file sessions per second, but the non-replicated version would crash the machines almost

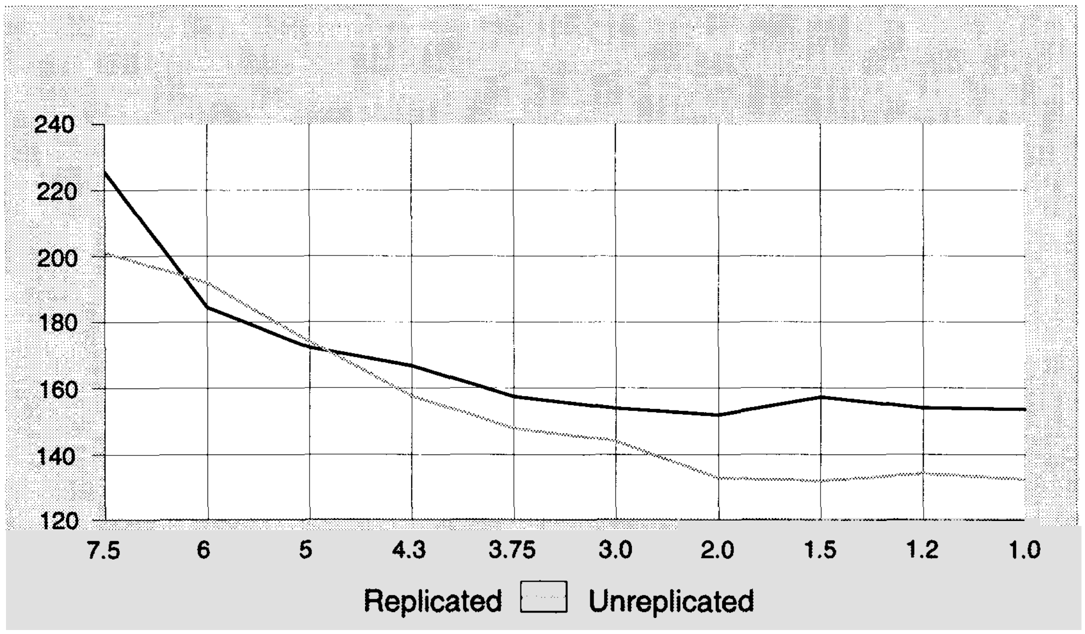

Figure 7-1: The effect of load on the replicated and non-replicated protocols.

immediately. The replicated version would execute completely, but would exhibit extremely high file session times, and many file sessions were aborted due to extreme response times from the servers. This behavior should be expected as at this load file sessions are being generated faster than they are removed from the system. Generating file sessions at a rate of 10 per second will quickly increase the number of file sessions being processed, from 2 or 3 in the system at a time, to 50 or more, if the system fails to service them fast enough.

Table 7-5: Response times for heavy loads to determine thrashing points.

LOAD	REPLICATED	NON-REPLICATED	
			ms below
OPEN			
7.5	123.0	101.9	17%
6	106.8	104.3	2.4%
5	100.2	96.4	3.8%
4.3	102.7	88.4	14%
3.75	98.4	83.2	15%
CLOSE			
7.5	37.6	51.3	-36%
6	31.4	46.6	-48%
5	31.8	41.1	-29%
4.3	29.8	37.9	-27%
3.75	30.3	35.0	-15%
READ			
7.5	9.1	5.7	38%
6	6.2	4.9	22%
5	5.3	4.5	16%
4.3	4.6	4.0	13%
3.75	3.6	3.7	-2.8%
WRITE			
7.5	13.3	12.3	7.6%
6	10.3	10.1	2.2%
5	8.4	8.3	1.2%
4.3	7.7	6.7	13%
3.75	6.5	6.5	-0.93%
SESSION			
7.5	225.6	201.1	10%
6	184.5	192.0	-4.0%
5	172.3	174.2	-1.1%
4.3	166.7	157.6	5.5%
3.75	157.4	147.8	6.1%

A set of tests was performed that shows the effect of caching on our protocol. The tests were executed

Table 7-6: Caching costs

	Replicated	No Caching
Session	141.7	192.8
Open	100.9	55.3
Close	25.4	16.9
Read	3.0	26.6
Write	3.1	24.9

with a load of 1.2 file sessions per second, and an average of 3.5 operations per file session. The results are shown in Table 7-6. The results show the cost of transferring a complete file on open, and also shows the cost of always sending file system operations to the server. The measurements are interesting because they point out the costs paid by workstations that cannot support caching or that can only support a very small cache. The costs saved by not sending the file on open could be capitalized on by returning the result of an open operation to the agent, and then sending the file after the client has been allowed to continue processing. Although this lowers the cost of an open operation, the load on the networks increases slightly, along with the load on the servers and agents.

To examine how an increase in the number of operations would affect the protocols, all of the tests were run with 25 operations per file session at a load of 1.0 file session per second. The measurements from these tests are shown in Table 7-7. The number of operations only had the effect of emphasizing the savings

Table 7-7: Response times for runs with 25 operations per file session.

	Replicated	No Caching	Non-replicated	NFS
Session	215.0	637.1	189.6	79.2
Open	106.1	61.9	82.3	33.3
Close	26.1	18.5	26.5	11.7
Read	2.9	21.2	2.8	1.1
Write	3.0	21.2	3.0	1.3

noticeable with caching. The replicated and non-replicated protocols exhibited the same session time difference of approximately 25 ms.

8 Concluding Remarks

8.1 Future Work

8.1.1 Directory Service

We have outlined a protocol for file replication and client cache control, but have not included methods for handling a corresponding replicated directory service. In keeping with the basic characteristics of our protocol, the directory service should supply UNIX semantics and do so using a decentralized method. Although neither of these features are necessary, we believe they can be achieved efficiently and effectively. This section will briefly sketch a method that might be employed to provide the directory service.

The protocol we have outlined for file replication can be altered to provide a method of replication for directories. When opening a file, all directories in the path of a file would be opened by the contact. When a contact sends the open notice to the other servers, each server performs the open operation for the file and for every directory in the path of the file that the contact does not already have open. The contact is required to communicate with a majority of the servers when opening the file, so each server will note that the opening contact is also opening all directories in the file's path. The state information for all of the directories in the path would be sent back along with the file's state, if necessary. If a server has the top *n* levels of a path already open, then the state for those n levels need not be passed to the contact.

Since opening a file will cause the opening of all directories in its path by the contact, the top levels of the directory hierarchy will usually remain open at all servers. This will help to keep the open time down, by having all servers keep up-to-date copies of these directories. Alternatively, the servers could always keep the top *m* levels open in order to lower the open cost, as the top levels are frequently accessed.

Updates to directories would be performed at all servers that have the directories open. This will allow quick reading of directories at the expense of directory updates. Reads would be allowed at any server that has the directory open without contacting other servers, thus a form of the read-one-write-all approach.

8.1.2 Write Operation Methods

8.1.2.1 Remote Writes Using Quorums

In designing this protocol, we discussed a variety of methods for handling write operations while conflicting sessions exist in the system. The method chosen appears to be a good solution, but a quorum based write may be better. An investigation into the performance costs would be of practical value.

$$\boxed{\begin{array}{l} Qw + Qr > N \\ 2Qw > N \end{array}}$$

Figure 8-1: Constraints on the Qw and Qr quorum values.

The quorums would not be derived from the total number of servers, but from the number of contacts for the file. The quorum based method would be used when conflicting file sessions exist in the system and there is more than 1 contact serving agreements for the file. In order for an update to be applied to a file, only a write quorum (Qw) of the contacts need participate, rather than all of the contacts. This would require that read operations, in the same scenario, read from a read quorum (Qr) of the contacts. The two quorum values must abide by the constraints shown in Figure 8-1.

The quorum based method may help write operations in times of failure, but only when there are more than 2 servers acting as contacts for the file. This method would hinder read operations, and it may be difficult to determine whether an improvement, or a degradation in availability is noticed in these situations.

8.1.3 Open Operation Methods

As mentioned in section 4.1.3, a ROWA scheme or a quorum based scheme for open operations could have been used. Although it does not appear that a ROWA scheme would benefit our protocol, a quorum scheme might. It would be interesting to compare the performances of our protocol using a quorum based scheme to the protocol outlined in this thesis.

8.1.4 Caching Schemes

8.1.4.1 Whole file vs. Disk Block Caching

The style of caching used by the clients need not be whole file caching. This choice of caching style was made to allow clients to continue to access their caches even if the contact is not available. This has the disadvantage of transferring the whole file to the client even if the client does not need the whole file, wasting network bandwidth and cache space. The alternative, disk block caching, has the disadvantage of requiring extra processing by the server to lookup, and package all of the different disk blocks each time a new block is fetched. Which one of these methods is better for our protocol, would be an interesting question to examine.

8.1.4.2 Write-through Caching

During periods of conflicting file sessions, writer clients are forced to give up their file caches. This can cause unnecessary processing. If there is only a single writer client in the system, then the protocol may be better off if the writer keeps the file in its cache, when writing through to the server. If a second writing process appears in the mix, the original writer's cache would then be disabled.

This style of caching would also benefit during situations where there are conflicting file sessions on the same client machine. The cache could be kept on the local machine, forcing all writes to be propagated through to the contact, before returning to the user. Meanwhile read requests could be satisfied by the cache as long as they do not conflict with any in-progress writes. This would have the benefit of guaranteeing that updates that are visible to other processes are stable, yet most read requests would remain as efficient as they would in non-write-sharing circumstances.

8.1.4.3 Non-volatile Caches

If client machines were to implement non-volatile caching, the protocol could be changed in order to take advantage of this stability. In cases of conflicting sessions on the same client site, the cache could be kept as long as all updates are written through to the local disk. Close operations could also be allowed to delay the timing of update propagation in order to wait out short lived files, although this could lower the availability of the file if the client becomes unreachable from the servers.

8.1.5 Variable File Control Methods

The Deceit file system provides variable control methods in order to allow faster accesses to files that are rarely updated, or in which the consistency of the file is not of extreme importance. This idea could be a valuable tool in providing better efficiency in any replicated file system. The consistency guarantees need not be compromised, but different access method could be exploited to benefit the different ways in which files are used. For example, executable files are rarely modified, so making read sessions efficient (read one server) at the expense of update sessions (write all servers) would be beneficial. The ability to set the mode of replication on a per file basis, will allow those who know the access patterns of the file to exploit them.

8.1.6 Disconnected Operation Support

In our protocol disconnected operation is not supported. A disconnected client cannot communicate with any servers and thus cannot perform any file opens, or closes. With a minor modification to the protocol disconnected operation can be supported. Disconnected operation refers to clients operating when not in contact with any servers, as might be done with laptop computers.

There are a number of issues to be addressed when allowing disconnected operation. The most important being that the client needs access to the required files. Coda provides a mechanism called **hoarding** that caches copies of files at client machines to be accessed when the machine is disconnected. A form of hoarding could be implemented with our protocol to allow clients to operate while disconnected. In order to hoard files, the client would 'open' all files in its working set in a mode applicable to the type of access that would be required. The open operation would be performed by a hoarding process which caches the files on a disk local to the client. Any open operation into a conflicting mode would fail as the reason for opening the files is to get a copy onto the local disk and since this would not be allowed, the open should fail. Once disconnected, client processes would be allowed to perform file sessions on the copies that the hoarding process had opened. The clients' close operations would guarantee that updates are applied to the clients' local disk, to be applied later to the servers' disks upon reconnection.

When the client machine reconnects to the server, the hoarding process would perform a 'close' for each file that it had hoarded. This close operation would allow other sites to then open the files in a mode that would have conflicted with the disconnected client's operation.

Using a method like this would allow voluntary disconnected operation, but would not be useful for **forced disconnected operation**[1]. There is also an associated availability cost that would be incurred. When a client has hoarded a set of files and left, no conflicting accesses would be allowed. This should not be too much of a problem as most of the files would be the user's personal data files, and read-only system executables. As would be expected, the user would be unable to perform certain functions that would be available if the user were connected to the network. The user should also be restricted as to which files are allowed to be hoarded in order to keep mobile users from strangling the rest of the sites. In such a system, most applications should be developed in such a manner that no global files are modified otherwise the application would not be a candidate for use during disconnected operation.

8.2 Conclusions

This thesis has presented a protocol for a replicated file system that provides strict semantics and does so in a decentralized manner. The notable features of this protocol are:

1) the enforcing of UNIX semantics under all circumstances,

2) stronger permanence guarantees than UNIX semantics,

3) the inclusion of client cache consistency control in the protocol,

4) the low cost paid to enforce UNIX semantics,

5) the comparably low cost of failure handling,

6) and the decentralized nature of file control and file state memory,

7) the amortization of the open and close operation costs over the whole file session in order to achieve efficient read and write operations.

In staying away from a primary-copy configuration we have allowed the failure handling operations to be made more efficient and less of a bother to those using the file system. This has been achieved mostly through the sharing of responsibility at the server level, and by not requiring that all servers be up-to-date. Allowing servers to acquire missed updates when they are needed keeps the cost of failure handling down.

1. Forced disconnected operation refers to disconnected operation due to failures in the system.

The effects of failure handling is also lessened by allowing server participation in other operations to continue while the servers perform the failure handling functions.

Achieving strict semantics in a replicated file system allows for replicated and distributed systems to behave more like their centralized counterparts. This better provides the transparency that is so strived for in computing systems. The low cost which is paid by the system to provide the stricter semantics makes this style of file system more desirable. A clear semantic definition is the basis with which a good file system design starts; the provision of strong semantics gives application designers the required knowledge of file system responses during conflicting accesses and in the face of failures. Our protocol provides strong consistency, and permanence without paying a high cost to attain them.

Bibliography

[BAKE91] Baker, M. et al. Measurements of a Distributed File System. In *Proc. 13th ACM Symp. on Operating System Principles*, October 1991, pp. 198-212.

[BERN87] Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Databases* Systems. Addison-Wesley, 1987.

[FLOY86] Floyd, R.Short-Term File Reference Patterns in a UNIX Environment. Technical Report 177, Computer Science Department, The University of Rochester, Rochester NY, March 1986.

[GIFF79] Gifford, D.K. Weighted Voting for Replicated Data. In *Proc. 7th ACM SIGOPS Symp. on Operating Systems Principles*, Pacific Grove, CA, December 1979, pp. 150-162.

[GUY90] Guy, R., Heidemann, J., Mak, W., page, T., Jr., Popek, G., and Rothmeier, D. Implementation of the Ficus Replicated File System. In *USENIX Conference Proceedings*, Anaheim CA, June 1990, pp. 63-71.

[HISG89] Hisgen, A., Birrell, A., Mann, T., Schroeder, M., and Swart, G. Availability and Consistency Tradeoffs in the Echo Distributed File System. In *Proceedings of the 2nd Workshop on Workstation Operating Systems*, IEEE Computer Press, September 1989, pp. 49-54.

[KIST91] Kistler, J.J., and Satyanarayanan, M. Disconnected Operation in the Code File System. *ACM 13th Symposium on Operating Systems Principles*, October 1991, pp. 226-238.

[LEVY90] Levy, E., and Silberschatz, A. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*. Vol. 22, No. 4, December 1990, pp. 321-374.

[LISK91] Liskov, B., et al. Replication in the Harp File System. In *Proc. 13th ACM Symp. on Operating System Principles*, October 1991, pp. 226-238

[MANN89] Mann, T., Hisgen, A., and Stewart, G. An Algorithm for Data Replication. Report 46, DEC System Research Center, Palo Alto, CA, 1989.

[MARZ88] Marzullo, K., and Schmuck, F. Supplying High Availability with a Standard Network File System. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, CA, 1988, pp. 447-453.

[MUMM93] Mummert, L. Efficient Long-Term File Reference Tracing. Carnegie Mellon University, manuscript in preparation, 1993.

[PARI86] Paris, J. Viewstamped Replication for Highly Available Distributed Systems. Technical Report MIT/LCS/TR-423, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[SATY90] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990, pp. 447-459.

[SIEG89] Siegel, A., Birman, K., and Marzullo, K. Deceit: A Flexible Distributed File System. Technical Report No. 89-1042, Department of Computer Science at Cornell University, November 1989.

[SIEG90] Siegel, A., Birman, K., and Marzullo, K. Deceit: A Flexible Distributed File System. In *USENIX Conference Proceedings*, Anaheim CA, June 1990, pp. 51-61.

[TAIT91] Tait, C. D., and Duchamp, D. Service Interface and Replica Management Algorithm for Mobile File System Clients. In *1st International Conference on Parallel and Distributed Information Systems*, Miami Beach FA, December 1991, pp. 190-197.

[TAIT92] Tait, C. D., and Duchamp, D. An Efficient Variable-Consistency Replicated File Service. In *Proc. USENIX File Systems Workshop*, Ann Arbor MI, May 1992, pp. 111-126.

[THOM87] Thompson, J. Efficient Analysis Of Caching Systems. Technical Report No. UCB/CSD 87/374 at Computer Science Division, University of California, Berkeley, CA, October 1987.