Performance in Flexible Distributed File Systems

Alexander Siegel
Ph.D Thesis

TR 92-1266
February 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

Performance in Flexible Distributed File Systems

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Alexander Siegel

May 1992

Performance in Flexible Distributed File Systems

Alexander Siegel, Ph.D.

Cornell University 1992

There are many existing distributed file systems. Each file system provides a different degree of performance and safety. In this context, performance is the time required to satisfy individual requests, and safety is the set of guarantees that the file system provides to users. In this thesis, we characterize many of the trade-offs between performance and safety. We include numerical relationships whenever possible. As a corollary, it is shown that a flexible file system – one that provides a wide range of possible safety properties – can also have very good performance.

This thesis uses two main approaches, practical and theoretical. The practical approach centers around the Deceit File System. Deceit supports file replication, file migration, and a sophisticated update control protocol. Deceit can behave like a plain Sun Network File System (NFS) server and can be used by any NFS client without modifying any client software. Deceit servers are interchangeable and collectively provide the illusion of a single large server machine to its clients.

The theoretical approach presents several results that are applicable to all distributed file systems. A careful analysis of many systems yielded insights into the behavior of successful file systems. We formalize the relationships between safety conditions exposed by this analysis. We also determine the cost of reading and writing a file given different sets of safety conditions.

In conclusion, we find that Deceit does not totally meet the goal of being efficient under all possible sets of requirements. Deceit is highly efficient for cases that require a high degree of replication and safety, but it is inefficient in cases where very specific optimizations are possible. However, the flexibility that Deceit provides is still very useful. For example, we show that writing to a file with three replicas costs a factor of 5 more messages more than writing to a file with one replica. Allowing asynchronous disk writes instead of synchronous writes can decrease latency by a factor of more than 30. Since Deceit allows the user to choose among these possibilities, dramatic performance gains can be achieved in many cases.

Biographical Sketch

Alexander Siegel was born in Lafeyette, Indiana on November 25, 1966. He spent the majority of his childhood in Albuquerque, New Mexico where he attended Highland High School. During this time, he also attended several courses at the University of New Mexico in Albuquerque. In the Fall of 1983, he enrolled at Rice University in Houston, Texas. In the Spring of 1987, he graduated *Magna Cum Laude* with a Bachelor of Science degree in Electrical Engineering. He also received a Bachelor of Arts degree in Computer Science and Mathematics at that time. In the Spring of 1987, he enrolled at Cornell University in Ithaca, New York as a graduate student in Computer Science. He continues to study at Cornell as of May, 1991.

Acknowledgements

I would like to thank my advisor Keith Marzullo. He kept me going whenever I was happy to stop. I would also like to thank Ken Birman. Without his generous support and insight, this work would have been impossible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There are two properties of distributed file systems that clearly conflict: the *performance* of the system and the *safety* it provides in the face of failures[Her85,Her87]. Furthermore, as the field of distributed file systems has matured, the emphasis has been to increase the performance at a cost of safety. Our thesis is to better understand in practical terms the nature of this trade-off. Additionally, we explore the notion of allowing the trade-off to be made a parameter of each file, thereby allowing the user to explicitly choose the degree of safety required.

This thesis uses two main approaches. The first approach is a detailed survey of the most important existing distributed file systems. This survey provides several examples of different combinations of performance and safety. It also establishes what types of safety are recognized by the research community as desirable. We also describe Deceit[SBM89,SBM90a,SBM90b], a uniquely flexible distributed file system. With Deceit, the core protocols can be controlled by users through a set of file parameters. Using this flexibility, we explore performance and safety trade-offs that normally would be too exotic to implement.

This survey into the behavior of real distributed file systems provides many informal insights that reveal themselves as recurring patterns of behavior. In our second approach, after formalizing these insights and presenting a reasonable system model, we derive several technical results as performance/safety trade-offs. Given these results, it is possible for file system designers to intelligently decide which safety properties are worth the cost.

The rest of this chapter describes the methodology we use in the file system survey. Chapter 2 surveys existing file system efforts excluding Deceit. Chapter 3 provides a detailed description of the Deceit file system. Chapter 4 provides a technical analysis of Deceit and of performance/safety trade-offs in general. Chapter 5 summarizes and concludes the thesis.

1.1 Survey Method

The first question usually asked about a new file system is "how fast is it?" This is similar to comparing CPUs by clock rate only. Many other factors influence the usefulness of a file system, and in general a system that can handle a wide variety of requirements with reasonable efficiency is more useful than a system that solves one particular problem very well. Also, a file system with strong safety properties may be more widely applicable than an unsafe one. By assuming a single focused set of requirements, a system may be inefficient or unusable under slightly different conditions.

A substantial amount of analysis is required to thoroughly compare two distributed file systems. Furthermore, since implementations vary widely in the environment in which they run and the usage made of them, it is difficult to directly compare one against another. In our survey, each file system is distilled down to a small set of attributes that depend only on the system design and not on implementation details. In particular, absolute performance figures will not be presented, since these values are strongly dependent on the processors and disks used.

Network and Failure Model

To strengthen the comparison, a common network and failure model is applied to all file systems. Some file systems require unusual network hardware, and corresponding adjustments to the model are made for these systems. The network model is very simplistic in order to keep the comparison manageable. We assume that all servers are attached to a linear network as shown in Figure 1.1. Each server has a single disk subsystem. A *network segment* is a portion of the network between two adjacent servers. One client is attached to one end of the network, and other clients may be attached elsewhere.

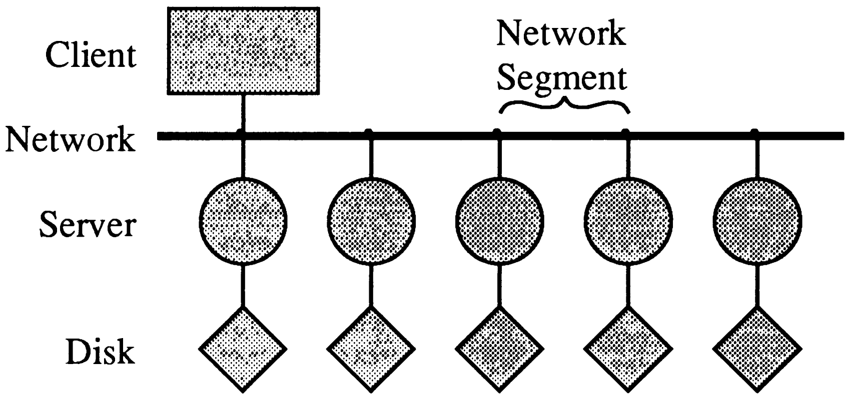

Figure 1.1: Network Model

Table 1.1: Failure Probabilities for System Components

Component	MTTF[1]	MTTR[2]	Availability
network segment	10 months (7200 hours)	6 hours	99.9 %
server host	16.6 days (398 hours)	18.2 hours	95.6 %
disk/media	4 years (35064 hours)	4 hours	99.989 %

[1] Mean time to failure
[2] Mean time to recovery

We use the failure times shown in Table 1.1. Component failures are assumed to be independent of each other. The server host times were taken from [LCP90] assuming a Sun 3/280 workstation. This type of workstation is used commonly for file servers, and it is well represented in the sample. The other figures are from [Gra85,Gra90], and are based on the assumption that replacement hardware is readily available. Clearly, these failure times may not apply to all environments, but they will illustrate general features of availability and reliability.

On course, it would be very useful to have a more detailed set of statistics, but in general it is very difficult to collect reliability statistics about computer system components. Failures can come from an alarming variety of sources and are notoriously bursty. Additionally, the independent failure assumption is very unrealistic. For example, a software upgrade can render an entire local area network unreliable for several days. Computer hardware is extremely reliable until the first electrical storm or fire. A virus can slowly destroy every computer on a network. We have little alternative but to be satisfied with imprecise models and results.

We assume that client caching is used, since this facility is a key component of a practical file system. We model a cache using a fixed probability of a client cache miss γ where $0 \leq \gamma \leq 1$. Cache design and cache consistency are specific to the client/server communication protocol in a file system. This thesis will not emphasize this protocol; instead, it will emphasize the server itself. In particular, we emphasize the problem of coordinating file access and updates in a file system that provides fault tolerance. Is possible to provide fault tolerance using any reasonable client protocol: a fault tolerant file server can emulate a normal file server. Therefore, we will not study this protocol in detail.

Performance Comparison

The first comparison measure is *performance.* Since few of the file systems are available on identical hardware in an identical environment, a direct timing comparison is usually impossible. Even small details (e.g. data placement on disk) can have a dramatic effect on total performance. Fortunately, the efficiency of a file system can be partially described by the amount of network traffic and the number of disk operations required for an operation. We assume that the software execution time is negligible, and network message size does not significantly influence the total transmission time. This is currently a good assumption: the cost in transmitting a message is dominated by the overhead of preparing the message for transmission and delivering the message to the process at the destination [SB89]. These assumptions will become more valid as networks and CPUs increase in speed.

More specifically, the cost of running a protocol will be broken into four categories. A *network phase* is the time required to transmit a packet from one server to another. For example, we have measured a network phase at 2 milliseconds on a 10 megabit/second Ethernet under UNIX[3]. A network phase can be classified into one of 2 categories: *synchronous network phases* (SNPs) and *asynchronous network phases* (ANPs), where a SNP must be completed before the client can continue, and an ANP merely must be completed eventually. For example, assume that a `write` operation is issued to a file with three replicas, and an additional round of garbage collection is required among the servers after the client has received its reply. This example is illustrated in Figure 1.2. The initial broadcast and reply are synchronous since the client must wait for them to complete, and the garbage collection afterwords is asynchronous since it can be delayed without delaying the client, so the total cost is 2 SNPs, 2 ANPs, and 10 messages.

The other two categories are *synchronous disk accesses* (SDAs) and *asynchronous*

[3]UNIX is a registered trademark of AT&T.

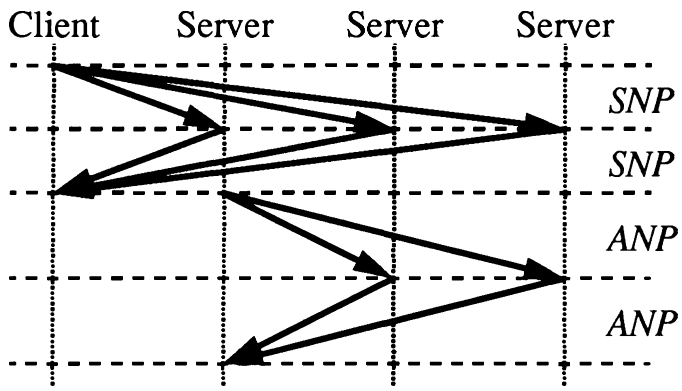

Figure 1.2: Example Network Cost

disk accesses (ADAs). On a medium speed hard disk, we have measured a SDA takes at 50 milliseconds. As with network phases, an SDA must be completed before the client can continue, and an ADA need only be completed eventually. ADA operations are uncommon in file system specifications due to the possibility of server failure causing the operation to be aborted without client notification. Ironically, ADAs are much more common than SDAs is real file systems since SDAs are very expensive.

We will compute the number of SNPs, ANPs, SDAs, ADAs, and total number of messages for the following base file operations:

- `create`* - create a normal file
- `createdir`* - create a directory
- `delete`* - delete a normal file
- `deletedir`* - delete a directory
- `open` - prepare a file for reading or writing
- `read` - read file data
- `write`* - write file data
- `close` - discontinue access to a file

- `readdir` - list the contents of a directory

Operations which are marked with a $\star$ are referred to as *update operations* since they modify the disk data. These operations are not a complete list of all file system operations, but they do characterize the core functionality in a file system.

An additional measure of cost is the total number of network messages required for an operation. This cost is particularly important under heavily loaded conditions. For `read` and `write` operations, the total number of messages will be computed.

Availability Comparison

Availability is the expected percentage of time such that an operation can be completed. *Reliability*[4] is the expected amount of time that the system remains available if it is currently available. There are different values of availability and reliability corresponding to each of the different file operations and their inherent tolerance to failures. For this analysis only *read availability* and *write availability* is considered. Also, availability is computed assuming only one client; if there are multiple clients, then availability for all clients will be lower since the system must be available to all clients.

In the analysis, the availability and reliability for `read` and `write` operations are explicitly computed using the times from Table 1.1. The analysis methods are described in [BP75]. We assume component failure distributions are exponentially distributed. In other words, system components have a uniform probability of failure regardless of age. To compute availability, we sum the probabilities associated with each available system state. To compute reliability, we sum the conditioned decay components associated with each system failure transition. Due to the large number of formulas involved, we do not show the detailed computations involved in deriving

[4] This value is also called the "mean time to failure" or MTTF.

these values.

To demonstrate the analysis, and to motivate replication for availability, we now present an example. Assume that there are three file servers and three file replicas (the variable n will be used to denote the number of replicas). Further, assume that a failure of any single component disables the entire system. Under these conditions, the availability of the system is 86.72%, and the reliability is 126.6 hours (5.3 days). Now, assume that it is sufficient for only two servers[5] to be accessible to the client. Under these conditions, the availability of the system is 99.299%, and the reliability is 1,259 hours (52 days). These values are a factor of 19 less unavailability and a factor of 10 more reliability than a single server.

The two extremes of availability and reliability are as follows: a system has *minimum availability* if the loss of any single component causes the entire system to become unavailable, and a system has *maximum availability* if operations can complete when any server is accessible to the client. Table 1.2 and Table 1.3 show the availability for the two extremes assuming the component failures rates from Table 1.1. Figure 1.3 shows the same data in graphical form. Note that maximum availability has an asymptotic value for large n because network failures tend to eliminate the value of adding new servers.

The availability of 4 or more servers with maximum availability suggests an incredibly reliable server, but this reliability is unachievable in practice. Many events affect every server at a site and will tend to cause all servers to fail. For example, power lines have about a mean time between failures of 6 months and a mean time to recovery of 2.5 hours [Gra90]. Administrative activities (e.g. software upgrades) are particularly notorious in this respect. For large n, the independent failure assumption is insupportable.

[5] A server is a combined host machine and disk.

Table 1.2: Maximum Availability

n	Availability	Reliability
1	95.42%	393.5 hours (16 days)
2	99.786%	4,335 hours (6 months)
3	99.986%	41,508 hours (4.7 years)
4	99.9956%	101,895 hours (11.6 years)
5	99.9960%	111,873 hours (12.8 years)
6	99.9960%	112,505 hours (12.8 years)
7	99.9960%	112,540 hours (12.8 years)
8	99.9960%	112,542 hours (12.8 years)

Safety Comparison

These are a variety of *safety* properties that a file system can provide to a client. These properties allow users and applications to depend on the file system to behave predictably. Often, the correctness of an application will depend on these properties. Unfortunately, optimizations for high performance usually requires sacrifices to safety. Each file system designer must make a trade-off between safety properties and performance.

We have chosen five safety properties for the comparison which we describe below. Informally, the properties are:

- *Client cache consistency* describes the ability of the file system to maintain valid client caches. A stale client cache can lead to confusion between clients about file contents.
- *Data consistency* extends the issue of cache consistency to include all file replicas. If file replicas are inconsistent, then the result of a `read` can be

Table 1.3: Minimum Availability

n	Availability	Reliability
1	95.42%	393.5 hours
2	90.97%	191.5 hours
3	86.72%	126.6 hours
4	82.68%	94.51 hours
5	78.82%	75.41 hours
6	75.15%	62.73 hours
7	71.64%	53.70 hours
8	68.30%	46.95 hours

unpredictable.

- *One-copy serializability*[6] is another method of describing safe replication. A file system satisfies one-copy serializability if replication is invisible to clients.
- *Update stability* is a form of consistency over time. Update stability is satisfied if a file never loses completed updates.
- *Atomicity* describes consistency in the face of crashes. Atomicity is satisfied if the file system remains in a consistent state after recovering from a failure (i.e. no operation had a partial effect).

Client Cache Consistency

Client caches must be kept up to date with respect to server data, but file systems vary with their approach to this problem. Client cache consistency will be described using *values* that are determined by when the client cache data is refreshed. For this property, we assume that there are no failures. If client cache consistency is *false*

[6]This term is actually a misnomer in this thesis. Originally, one-copy serializability was a well defined database term, but the file system community grabbed it, and the meaning has drifted.

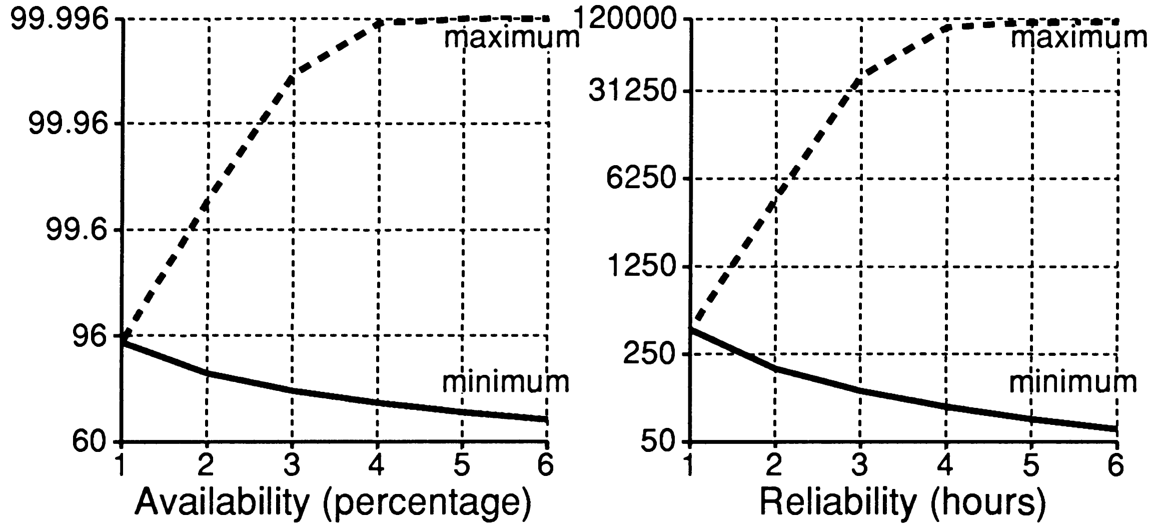

Figure 1.3: Availability and Reliability Plots

then client cache data can remain invalid indefinitely; there is no guarantee about when the data is refreshed. A value of *true* means that all client caches are refreshed during every `write` before it returns to the client. A value of *timeout* means that cache data is refreshed periodically while the file is being accessed. A value of *on open* means that cache data for a client is refreshed when the file is opened at the client. A value of *on close* means that all client caches are refreshed when the file is closed. Values can be combined to describe a combination of approaches to cache consistency. For example, *on open* can be combined with *timeout* to describe a file system that refreshes the cache both when the file is opened and periodically.

Data Consistency

Failures can prevent updates from being delivered to all replicas of a file. As a result, it is possible for replica contents to diverge from each other. If a client is reading from one replica and switches to a different divergent replica (e.g. in response to a failure), then confusion can result. Different file systems provide different guarantees about consistency between replicas during or after a failure. At one extreme, a file system can guarantee that all updates are applied to all replicas. If a server is down, then the update is aborted or blocked. At this extreme, data consistency will have a value of *true*. At the other extreme, there are no guarantees, and data consistency will have a value of *false*. If data consistency is *eventual* then replica consistency will be eventually achieved, provided that there are no updates for a sufficiently long period of time, and failures do not permanently block communication. A value of *user* means the same as *false*, except inconsistency is always detected and brought to the attention of the user for resolution. A value of *non-directory* means the same as *user*, except directory inconsistency is automatically resolved in some way.

One-copy Serializability

It is a valuable simplification for the client to be able to assume that there is only one copy of file data. Replication can be exposed when `read` and `write` operations are allows to interact without global ordering. For example, in the Ficus file system (Section 2.8), it is possible for `writes` to two different files to cross over each other, as shown in Figure 1.4. As a result, a `read` on one client will show the order of updates to be different than a `read` on another client. There is no global order of `reads` and `writes` that can account for the results. For this property, we assume that there are no crashes and the client cache has perfect consistency. If one-copy serializability has a value of *false* then it is possible to detect the underlying replication, and client software must be prepared for strange results. A value of *true* means that it appears that there is only one copy of all data.

Update Stability

Normal file system semantics assume that a completed update has been committed to stable storage, and any overwritten data is lost. If a file system does not synchronously update to stable storage, then an ill-timed crash can cause data to

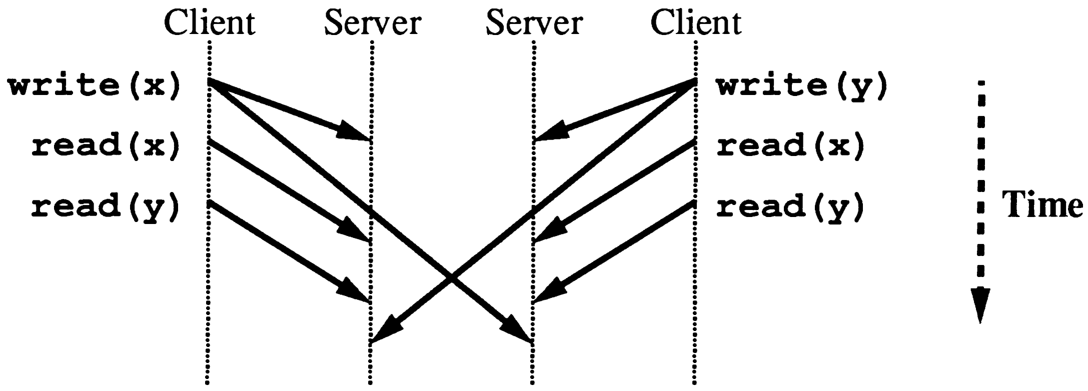

Figure 1.4: Ficus Update Example

be lost. Also, if a file system uses replication and fail-over[7], a failure can cause a client to switch to a server with obsolete file data. If update stability has a value of *true*, then update stability is guaranteed after the client has received the reply to a `write`. After that time, obsolete data will never be returned from a `read`. A value of *false* means that data may regress at any time. A value of *timeout* means that data becomes stable after a known period after the `write` has returned, but a crash can cause the data to be lost until that time. A value of *on close* means that data is stable only after the file has been closed.

Update stability is weaker than one-copy serializability (1CS). A file system that satisfies 1CS will have the property that `read` operations are consistent with previous `write` operations since they must be serialized in that order. Data consistency is independent of update stability and 1CS since a file system can nearly decouple the state of replicas from the view that a client sees. For example, a file system that only allowed the client to see one particular replica would satisfy 1CS trivially, and the state of the other replicas would be irrelevent. In practice, there is a relationship since the client can see those replicas during a failure.

[7]Fail-over occurs when a client switches servers in response to a server crash.

Atomicity

Failures of system components can cause partially completed operations to be stopped without completing or being cleanly aborted. A system has the property of *atomicity* if all operations are eventually completed or have no effect. In other words, each file system operation is an atomic transaction. If atomicity has a value of *true* then the system has atomicity. A value of *false* means otherwise.

Chapter 2

File System Survey

Distributed file system surveys are common in the literature: [Hac85] and [Sat90a] are two recent examples. Such surveys usually focus on a narrow aspect of the systems (e.g. client caching) since a survey on all of the aspects of existing file systems would be huge. In our survey, we focus on the performance and the safety provided by each type of file server, following the methodology described in Section 1.1.

The total number of existing file systems is enormous, and our survey covers only eight of them. We included file system that have distributed capabilities and are well-known. If the reader is interested, then the bibliography contains many references to other file systems that are not discussed in this chapter.

Section 2.1 presents the terminology used in the description of each file system. Each of the remaining sections is devoted to a specific file system and provides a brief summary of the architecture followed by an analysis. These sections are separated into three categories. In the first category, Andrew (Section 2.2) and NFS (Section 2.3) are two file systems that do not provide replication and will not necessarily contribute to the main thesis. However, many later file systems heavily depend on the development of Andrew and NFS, and the presentations are correspondingly dependent. In the second category, RNFS (Section 2.4), Echo

(Section 2.5), and HA-NFS (Section 2.6) provide replication based on large objects (e.g. whole disks). The designs of these systems are simplified by this coarseness, and they have many design similarities. In the third category, Locus (Section 2.7), Ficus (Section 2.8), Coda (Section 2.9), and Deceit (Chapter 3) provide replication based on small objects (e.g. files). We summarize all file system characteristics in tabular form in Section 2.10. We conclude the chapter with a summary of our observations in Section 2.11.

Most of the material describing individual file systems was copied directly from the referenced documents, with some editing for terminology and conciseness. Due to the editing, the copied text is not explicitly quoted; instead, we note here that such text has been taken from the referenced sources.

2.1 Definitions

We present the following glossary of terms:

atomic action - an action (e.g. a `write`) that is guaranteed to execute at either all involved sites or at no site.

atomic broadcast, abcast - a multicast operation in which all messages are received atomically and in the same total order.

capability - a *unique identifier* that is very difficult to duplicate or manufacture without access to privileged information. Holding a capability authorizes the holder to perform some action.

client - a computer or a program that issues requests to a *server* and receives requests from a *user*. Clients can also be servers to other clients. See *user* and *server*.

directory - a list of pairs associating names with file system objects such as a file or directory. See *name space*.

distributed - a system that runs concurrently on multiple processors. We assume that the communication between processors is inherently unreliable or asynchronous.

dual-ported disk, twin-tailed disk - a disk drive that can be controlled from two separate hosts. Special hardware is required to handle the situation where both hosts try to simultaneously use the disk.

file handle - a *unique identifier* referring to a file or directory. See *unique identifier.*

hint - cached information that can be wrong, but its correctness is easily determined at the time of usage.

logical volume - a *volume* that appears to be a single block of files to the client. A volume can be physically replicated or distributed across multiple servers. See *volume.*

meta-data - information that describes naming, ownership, protection, locking, page location, etc. for files. Meta-data does not refer to the actual contents of files.

name space - the function that maps textual file names to physical or logical data segments. For example, "/usr/bin/sh →< server=odin, disk=sd1h, inode=14985 >" might be an element of a name space.

partition - a loss of communication between two components of a distributed system in which the separate components continue to function.

process group - a set of cooperating processes or machines that work on the same task or data object.

RAM disk, silicon disk - a large memory buffer designed to emulate a very fast disk. It often used conjunction with an *uninterruptible power supply.* See *uninterruptible power supply.*

remote procedure call (RPC) - a message exchange between one process and another that emulates a normal procedure call, i.e., is synchronous. The process that initiates the exchange is often called the *client process* and the other process is often called the *server process.*

scalability - the ability of a system to grow to a very large size. In a distributed file system, *scalability* can refer to the number of clients per server, the total number of servers, or the total number of files.

serializable - a system behavior that is consistent with some serial ordering of operations.

file server - a computer with a non-volatile storage system which forms a permanent file repository. Occasionally, a *file server* is a service which is composed of several computers.

server disk - the non-volatile storage system of a file server.

server host - the computer component of a file server.

site - a single computer or a small set of tightly coupled computers.

unique identifier (UID), handle - a name which maps to at most one physical or logical object in a system. Unique identifiers can be capabilities, but there is no necessary relationship.

upcall, call-back - a RPC that is issued in the reverse direction than the direction used in typical operations. The difference between an *upcall* and a normal *RPC* is often just a matter of convention. An *upcall* also reverses the communication synchronization and can be used as an interrupt to the client process. See *RPC.*

user - a human being or program that initiates operations.

volume, partition - a large block of files that can be manipulated (e.g. backed up) as a unit.

2.2 Andrew

Andrew is a distributed workstation environment that has been under development at Carnegie Mellon University since 1983. The primary data-sharing mechanism is a distributed file system [HKM+88,Sat89a,MSC+86,SS90a] spanning all the workstations. Using a set of trusted servers, collectively called *Vice*, the Andrew file system presents a homogeneous, location-transparent file name space to workstations.

Scalability is the dominant design consideration in Andrew. Many design decisions in Andrew are influenced by its anticipated final size of 5000 to 10,000 clients. Careful design is necessary to provide good performance at large scale and to facilitate system administration. Scale also renders security a serious concern, since it has to be enforced rather than left to the good will of the user community.

The shared name space is partitioned into disjoint subtrees, and each such subtree is assigned to a single server, called its *custodian*. Each server contains a copy of a fully replicated *location database* that maps files to custodians. This database is relatively small because custodianship is on subtrees, rather than on individual files. These databases are treated as hints.

Files in the shared name space are cached on demand on the local disks of workstations. A cache manager, call *Venus*, runs on each workstation. When a file is opened, and if a local copy is not available, an up-to-date copy is fetched from the custodian. `Read` and `write` operations on an open file are directed to the cached copy. If a cached file is modified, it is copied back to the custodian when the file is closed. Cache consistency is maintained by a *callback* mechanism. When a file is cached from a server, the latter makes a note of this fact and promises to inform the client if the file is updated by someone else. The use of callback, rather than checking with the custodian on each open, substantially reduces client-server interactions. Andrew caches large chunks of files (64 kilobytes) to exploit bulk data transfer.

Older versions of Andrew cached whole files. Coda (Section 2.9) still uses this approach. Doing so provides better fault tolerance and consistency. Unfortunately, it also prevents access to extremely large files. Also, if a large database file is only slightly modified, then it would be inefficient to load and store the entire file. The choice of 64 kilobytes is a compromise solution.

Since Andrew does not provide replication, there will be no technical analysis.

2.3 NFS

The Sun Network File System[Sun86d,Sun86b,SM89,Sun90] was developed at Sun Microsystems. It provides transparent remote access to shared file systems over local area networks. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of RPC primitives built on top of a standardized External Data Representation (XDR). NFS has become a *de facto* industry standard, and almost all operating system venders now support at least NFS client capability.

NFS uses a nearly stateless protocol. That is, a server does not need to maintain much state information about any of its clients in order to function correctly. The only necessary client state is a cache of recent replies. Some RPCs are not idempotent, and retransmission can cause trouble. The reply cache allows the server to resend the original reply rather than retrying the operation. A summary of the NFS protocol can be found in Appendix A.

NFS does not support replication, and NFS servers are independent of each other. As a result, if a server crashes, its data becomes unavailable. NFS deals with this problem by forcing clients to block until the server recovers. The operation is retried periodically until it succeeds.

Since NFS does not provide replication, we include no technical analysis.

2.4 RNFS

RNFS[MS87] is a distant progenitor of Deceit which was developed at Cornell a few years before Deceit. RNFS implements a network file service that is tolerant to crash failures and can be run on top of NFS (Section 2.3). The fault-tolerance is completely transparent, so the resulting file system can support the same set of workstations and applications as NFS. NFS was chosen because it is available and easily used, and its simple stateless protocol made the task of replication easier.

RNFS achieves fault-tolerance by replicating the server and using an atomic broadcast protocol for communication. Transparency is achieved by directing all of the client requests to an intermediary (called an *agent*) that in turn broadcasts to the servers. The agent uses virtual file handles to identify each file or directory that it manages. The agent maintains a *replicated file list* for mapping virtual handles to real NFS handles and vice versa.

The agent performs a `write` by forwarding the request to each server with the virtual file handles replaced by real handles. The agent replies to the client when all of the writes are completed. This ensures that the NFS write-through semantics are preserved. The agent performs a `read` by forwarding the request to one of the servers.

When a server fails, the agent marks all file replicas on that server as *down*. When a `write` is done to a file with a replica on a failed server, the agent marks the unavailable replica as *invalid* in its replicated file list and updates all of the available replicas. When a server recovers, the agent changes the state of all replicas on that server to *up*. Each replica that was marked *invalid* is brought up to date. Recovery is accomplished by transferring contents from *valid* replicas to *invalid* replicas. These transitions are shown in a state diagram in Figure 2.1. Partition failures are handled by ISIS.

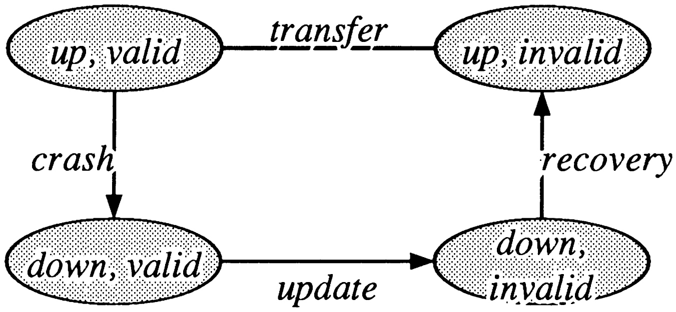

Figure 2.1: RNFS File State Transitions

In order to protect against agent failure, the agent maintains a stable version of the replicated file list in stable storage. At a minimum, the stable list must contain the file handle mapping and whether each file replica is *valid* or *invalid.* The stable list is itself replicated on all the servers in a special file. Changes to this file are only necessary when a file is created, deleted, or changes state.

The agent is replicated for fault tolerance, and it uses ISIS for communication. Because clients do not broadcast their requests to all agents, some other mechanism is needed to order updates to a file. Agents mutually exclude each other when writing to a file with a *write-token.* An agent must acquire a *write-token* before it is allowed to update any replica of the corresponding file. Information about the current token holder is replicated at every agent. Should the holder of the token crash, another agent is chosen arbitrarily to inherit the token. Since `write` operations tend to come in bursts from a single source, the cost of acquiring the token is amortized over a large number of `write` operations.

Performance Comparison

RNFS simply replicates update operations over all servers. Token acquisition may be required for the first update in a sequence. `Read` operations can be handled by any server, but they have to go through a level of indirection. Let n be the degree of server replication, and let m be the degree of agent replication. Table 2.1 shows the cost of each type of operations.

Table 2.1: RNFS Operation Cost

Operation type	SNP	ANP	SDA	ADA	Total Messages
Update (with token)	4	0	1	0	$2n+2$
Update (without token)	6	0	1	0	$2n+2m$
Non-update	4	0	1	0	4

Availability Comparison

RNFS provides maximum availability, but a least one agent has to be available in addition to a server. Table 2.2 shows the availability of RNFS assuming that there are n servers and m agents. Note that a system with 1 server and 2 agents is as available as a system with 2 servers and 1 agent.

Table 2.2: RNFS Availability

n, m	Availability	Reliability
1, 1	91.05%	196.7 hours
1, 2	95.22%	360.8 hours
1, 3	95.43%	389.8 hours
2, 2	99.572%	2,168 hours
2, 3	99.772%	3,925 hours
3, 3	99.972%	20,754 hours

Safety Comparison

RNFS has the following values for each safety property:

- client cache consistency = *timeout*

  Client cache data is maintained with a 30 second timeout according to the NFS protocol.

- data consistency = *false*

  RNFS can not tolerate partitions. If updates are issued on both sides of a partition, undetected inconsistency will result.

- one-copy serializability = *false*

  `Read` requests are not synchronized with updates if they go through a different agent.

- update stability = *true*

  RNFS uses a protocol to determine the last process to fail[Ske85]. Therefore, it is guaranteed to recover from the latest file data.

- atomicity = *false*

  RNFS to careful to maintain atomicity, but a crash in the middle of an operation can leave the underlying NFS servers in an inconsistent state.

2.5 Echo

In Echo[HBJ+90,HBM+89,MHS89], disks are multi-ported and connected to several server computers. Figure 2.2 illustrates the Echo architecture. Each such disk has a multi-port arbiter which recognizes at most one connected server host as its owner at a time. Ownership is subject to timeout. Echo can also make use of single-ported disks; in this case, software on the single physically-connected server simulates the multi-port arbiter, and other logically-connected servers access the disk via this server over the network. We will assume that Echo uses real multi-ported disks in the analysis.

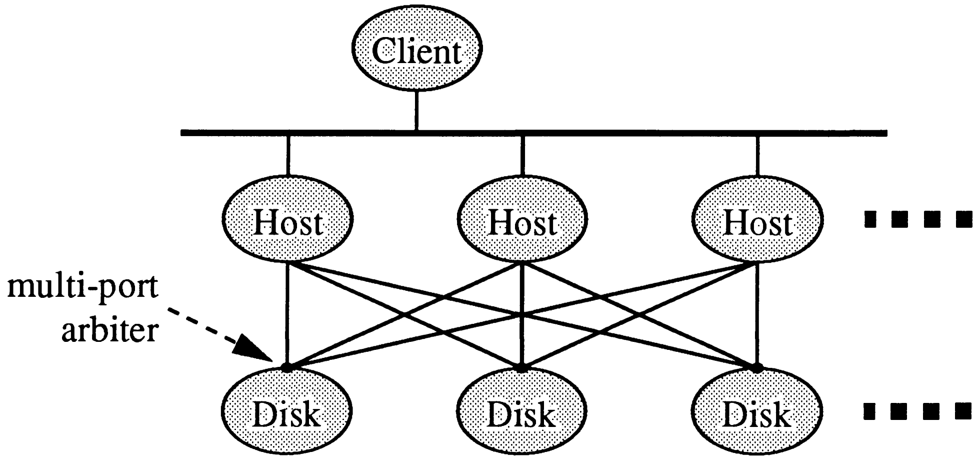

Figure 2.2: Echo Wiring Diagram

The server hosts are organized in a primary-secondary scheme. Servers communicate amongst themselves to choose a primary. A primary must be able to claim ownership of a majority of the disks. Because disk ownership is subject to timeout, the current owner must refresh its ownership periodically. After a failure, a secondary must wait for the disk ownership timeouts to expire before it can become the new primary. The waiting is required in order to guarantee that there is never more than one primary. All client operations are sent to the primary, and this requirement guarantees one-copy serializability.

Echo employs a distributed caching algorithm between clients and servers. Servers keep track of which clients have cached what files and directories. This caching information is replicated in the main memories of the server computers. The caches use delayed write-back instead of immediately sending new data to the servers. A token is used to provide cache consistency during concurrent access. A callback is used when clients caches need to be purged so that a token can be moved.

In Echo, replication is based on disk blocks. The granularity of replication is an array of disk blocks. A replica, called an *EchoDiskRep*, is configured from one or more physical disks. Two or more EchoDiskReps are combined to form an *EchoDisk*, which provides an array of highly available logical blocks to the next layer up. Echo volumes (subtrees of files and directories) are layered on top of the EchoDisk interface. Multiple Echo volumes may be stored in a single EchoDisk. Users control file semantics by placing files in a volume with the desired properties.

Performance Comparison

Since a single server can access all disks, all operations can be handled by a single server without network communication. All operations take 2 SNPs and 1 SDA; the total message count is 2 (request and reply). Due to the unusually connectivity, it may be fair to consider that the server/disk connections form a secondary network. Using this assumption, all operations take 4 SNPs and 1 SDA; the total message count is $2n + 2$.

Availability Comparison

Echo requires that a primary be able to own a majority of the disks. It is sufficient for a single host to be available to the client, and a majority of the disks to be functional. If the desired level of replication is even (e.g. 2 or 4), then an extra server can act as a witness[P$\hat{8}$6] to provide a useful majority. For the usual component availability values, an Echo system with 3 servers has an availability of 99.986% and a reliability of 41,475 hours (4.73 years).

Safety Comparison

Echo safety is as follows:

- client cache consistency = *true*

  Tokens and callbacks are used to maintain cache consistency.

- data consistency = *eventual*

  A majority of disks is required for any operation.

- one-copy serializability = *true*

  A majority of disks is required for any operation.

- update stability = *true*

  Each operation is forced to every disk. This task is accomplished in parallel.

- atomicity = *true*

  Echo uses a replicated log to insure atomic updates.

2.6 HA-NFS

HA-NFS[BEM90,BEM91] servers implement the standard NFS protocol (Section 2.3) and offer applications transparent recovery from server failures. Disks are dual-ported and can be accessed by both hosts as shown in Figure 2.3. Each HA-NFS server host is assigned a backup host that can access the server's disks if the latter fails. The backup impersonates the failed server on the network. The backup maintains no information about the server's state during normal operation, and is itself a file server for a different set of disks.

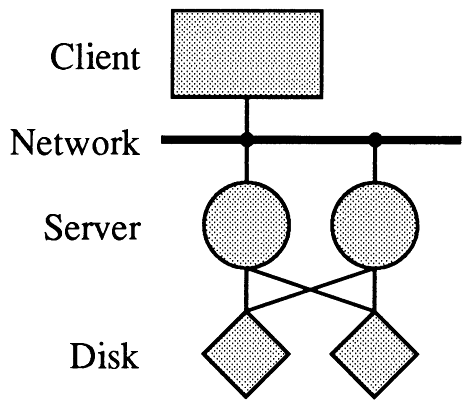

Figure 2.3: HA-NFS Communication

HA-NFS forces updates synchronously to disk as required by NFS semantics.

Servers record changes to file system meta-data in a disk log, ensuring the atomicity of file data. Disk logging also improves the performance of operations by obviating the need to synchronously force meta-data to their home locations on disk, thereby minimizing disk seek time. If a failure occurs, the backup uses the log to restore the file system structure to a consistent state. The log allows the backup host to be completely unloaded during normal operation. Integrating a recovering server into the system only requires regenerating the server state from the log.

HA-NFS uses *AIXv3's logical volumes*[1] to tolerate media failures. A logical volume is composed of equal-sized disk partitions that give the abstraction of a logical disk. To tolerate media failures, a logical volume can have up to three replicas. Replicated logical volumes are controlled by a single file server, reducing the cost of maintaining consistency. No message overhead is required beyond what normal NFS operations incur.

Performance Comparison

Since HA-NFS can update replicated volumes in parallel using the dual-ported disks, no network communication is needed. Therefore, all operations cost 2 SNPs and 1 SDA.

Availability Comparison

HA-NFS is limited to at most two servers. However, due to the cross connected disks, HA-NFS can tolerate the loss of the host of one server and the disk of the other server. With intelligent disk control hardware, even a loss of the network segment can be tolerated if the primary server host is still up. Using the usual component availability values, the configuration in Figure 2.3 produces an availability of 99.805% and a reliability of 4,583 hours (6.4 months). Note that this availability is slightly higher than the normal 2 server maximum availability.

[1] Logical volumes are an IBM software product for use with AIX.

Safety Comparison

HA-NFS safety is limited by the NFS protocol. However, HA-NFS does work hard to provide as much safety as possible. The values for each safety property are as follows:

- client cache consistency = *timeout*

  Client cache data is maintained with a 30 second timeout according to the NFS protocol.

- data consistency = *false*

  If one disk fails, and another recovers, inconsistent data will be stored. HA-NFS does not have a mechanism for determining the last process to fail, so it can not always recover from the latest version.

- one-copy serializability = *true*

  All operations go through a single coordinating server.

- update stability = *false*

  If one disk fails, and another recovers, obsolete data will be exposed.

- atomicity = *true*

  All operations are logged to insure atomicity.

If only one disk is used instead of two, then data consistency will be *true* and update stability will be *true*. Only a slight loss of availability will result since disk failure is very uncommon.

2.7 Locus

The LOCUS file system[WPE+83] presents a single tree structured name space to users and applications. The single tree structure covers all objects in the file system on all machines. It is not possible from the name of a resource to discern its location in the network.

Files in LOCUS can be replicated to varying degrees. A given file belonging to a *file group* X may be stored at any subset of the sites where there exist physical containers for X. The number and placement of replicas is chosen for each file when it is created. LOCUS keeps all copies up to date and assures that requests are served by the most recent available version. In the case where not all sites are communicating, not all the copies of the file are necessarily up to date.

For any file, sites are divided into three logical categories. A *client* issues requests for the file. A *storage site* stores the data for the file. The *current synchronization site* (CSS) enforces global access synchronization and selects storage sites for each request. There is only one CSS for any file group.

In LOCUS, as long as there is a copy of the file available, it can be used. If there are multiple copies, the most efficient one to access is selected. The CSS keeps track of the status of each replica, only the latest version of a file is visible to clients. The CSS maintains file state information in memory for all open files.

The basic approach in LOCUS is to maintain strict synchronization among copies within a partition. However, each partition operates independently. Upon merge conflicts are reliably detected. The appropriate application or user is responsible for reconciling inconsistencies.

Performance Comparison

All operations are required to go through the CSS before reaching a storage site. There are two cases depending on whether the CSS is a storage site with the most recent version. Table 2.3 show the cost of Locus for these two cases assuming that n is the degree of replication.

Table 2.3: Operation Cost in Locus

	CSS is a storage site	CSS is not a storage site
Network phases	2 SNPs	4 SNPs
Disk access	1 SDA	1 SDA
Messages for a `read`	2	4
Messages for a `write`	$2n$	$2n + 2$

Availability Comparison

Locus provides maximum availability: files can be read or written whenever any replica is available. Table 1.2 on page 9 shows the availability of a file for different levels of replication. The `read` and `write` availability are the same.

Safety Comparison

Locus has the following values for each safety property:

- client cache consistency = *true*
- data consistency = *user*

  A `write` gets sent to any available storage site regardless of consistency, but the user is notified if inconsistency results.
- one-copy serializability = *true*

  The CSS enforces one-copy serializability by acting as funnel for all operations related to a file.
- update stability = *false*

  A client can fail-over to an obsolete replica.
- atomicity = *true*

  Updates are atomic due to the shadow-page transaction mechanism.

2.8 Ficus

The Ficus file system[PGJH90,GHM+90,GP90,JPGH90] is an indirect descendent of the Locus file system presented in Section 2.7. A main focus of Ficus is stackable layers. Each file system layer contributes a small component to the overall functionality. By structuring the file system as a stack of modules, each with the same interface, modules which augment existing services can be added transparently.

The Ficus layered file system model comprises two separate layers constructed using the *UNIX vnode* interface as shown in Figure 2.4. The Ficus *logical layer* presents its clients with the abstraction that each file has only a single copy. The logical layer performs concurrency control and implements a replica selection algorithm. The *physical layer* implements the concept of a file replica. NFS (Section 2.3) is employed as a transport mechanism between the two layers, and can also be used as a means for non-Ficus hosts to access Ficus file systems. Using the vnode interface allows Ficus to utilize the UNIX file system as the underlying storage layer below the physical layer.

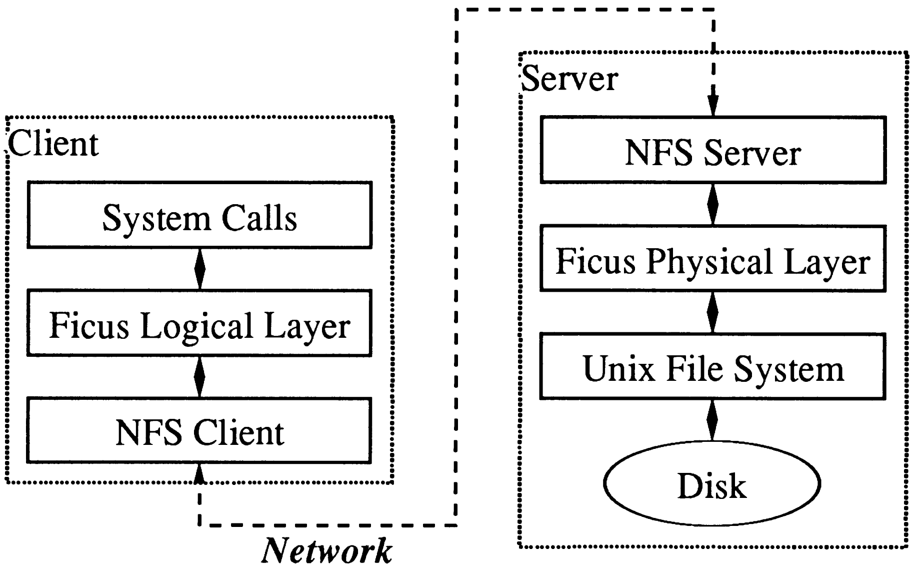

Figure 2.4: Ficus Layers

Each file and directory in Ficus may be replicated, with the replicas placed at any set of Ficus servers. The number and placement of file replicas is unbounded. A client may change the location and quantity of file replicas whenever a file replica is available.

Ficus has an algorithm for automatically reconciling directories after recovery. This algorithm determines which directory entries have been added to or deleted from remote replicas, and applies appropriate operations to the local replica. Directory update propagation is provided by the same algorithm. The execution proceeds concurrently with respect to normal file activity, so that client service is not blocked.

Ficus uses volumes as a basic structuring tool for managing disjoint portions of the file system. Ficus volume replicas are dynamically located and mounted as needed. The tables used for locating volume replicas are replicated objects similar to directories, and are managed by the same reconciliation algorithms used for directory replicas.

Update propagation is slightly unusual in Ficus. Only a single replica is synchronously updated, and this replica is chosen by examining all replicas. While a file is open, only this replica receives updates. When a file is closed, updates are asynchronously propagated to other replicas by transferring whole file contents.

Performance Comparison

Assume that an average of l_n messages are required to transfer a file over the network, and an average of l_d I/Os are required to write a file. Table 2.4 describes operation cost in Ficus.

Availability Comparison

Ficus provides maximum availability: files can be read or written whenever any replica is available. Table 1.2 on page 9 shows the availability of a file for different levels of replication. The `read` and `write` availability are the same.

Table 2.4: Operation Cost in Ficus

Operation	SNP	ANP	SDA	ADA	Total Messages
`open`	2	0	1	0	2n
`close`	2	$l_n + 1$	0	l_d	$2 + (l_n + 1)(n - 1)$
`read`, `write`, `readdir`	2	0	1	0	2
`create`, `createdir`, `delete`, `deletedir`	2	2	1	1	$2n$

Safety Comparison

Ficus has the following values for each safety property:

- client cache consistency = *timeout*

  Client cache data is maintained with a 30 second timeout according to the NFS protocol.

- data consistency = *non-directory*

  Data conflicts are left to the user to resolve, but there is automatic reconciliation for directories.

- one-copy serializability = *false*

  If there is concurrent activity on more than one file, then there is no mechanism for atomically ordering `reads` with respect to `writes`.

- update stability = *false*

  A client can switch to an obsolete replica due to a crash. This is a serious problem due to the lazy update propagation to replicas.

- atomicity = *true*

  The directory update propagation algorithm insures the integrity of metadata.

2.9 Coda

The Coda file system[Sat90b,SKK+90,Sat89b,SKS90], a descendant of the Andrew file system (Section 2.2), is substantially more resilient to server and network failures than Andrew is. Coda provides users with the benefits of a shared data repository but allows them to rely entirely on local resources when that repository is not available. Coda provides maximum availability regardless of failures in the system.

Clients cache entire files on their local disks as in early versions of Andrew. Whole-file transfer offers a degree of intrinsic resiliency. Once a file is cached and open at a client, it is immune to server and network failures. Cache coherence is maintained by the use of callbacks. When the server is lost, clients resort to *disconnected operation*, a mode of execution in which the client relies solely on cached data.

Coda uses server replication to provide a highly available shared storage repository. The unit of replication is a volume, and these are very similar to Ficus volumes (Section 2.8). A *replicated volume* consists of several physical volumes that are managed as one logical volume. Individual replicas are not normally visible to users. The degree of replication and the identity of the replication sites are specified staticly when a volume is created.

Coda integrates server replication with caching, using a variant of the read-one, write-all strategy. When servicing a cache miss, the client obtains data from one replica known as the *preferred server*. Although data is transferred only from one server, the client contacts all other replicas to collect their versions and other status information. The client uses this information to check whether the accessible replicas are equivalent and to insure one-copy serializability.

When a file is closed after modification, it is transferred to all replicas of the volume. This approach maximizes the probability that every replication site has current data, and all sites have the same data. Server CPU load is minimized because the burden of data propagation is on the client rather than the servers. Operations that update directories are also written through to all replicas.

Because its replication scheme is optimistic, it is possible to produce conflicting versions of a file. The update protocol guarantees eventual detection of conflicts when the file is read. A server performs no explicit remote actions upon recovery from a crash. Rather, it depends on clients to notify it in case of stale or conflicting data.

Coda checks for version conflicts on each server operation. When a conflict is detected, Coda first attempts to resolve it automatically. If automated resolution is not possible, Coda marks all accessible replicas of the inconsistent object and moves them to their respective *covolumes.* Covolumes are special storage areas associated with each volume similar to "lost+found" directories in UNIX. Coda provides a repair tool to assist users in manually resolving conflicts. It uses a special interface to the server so that requests from the tool are distinguishable from normal file system requests.

Performance Comparison

Most file activity takes place on the local cached copies of files, and a result is that most network traffic is avoided. Coda pays for this efficiency with expensive `open` and `close` operations. The file must be transferred to the client on `open` if there is a cache miss, and the file must be transferred back to the server on `close` if it has been modified. This transfer uses a bulk transfer protocol that can be modeled as a stream of asynchronous operations followed by a single synchronous operation. Assume that an average of l_n messages are required to transfer a file over the network, and an average of l_d I/Os are required to read or write a file to disk. For the sake of consistency with other performance computations, we will assume that the cache always misses. Let ω be the ratio of `write` operations to `read` and `write` operations. Table 2.5 shows the latency of each operation as well as the cost in total messages.

Table 2.5: Operation Cost in Andrew

Operation	SNP	ANP	SDA	ADA	Total Messages
`open`	3	$l_n - 1$	1	$l_d - 1$	$l_n + 2n$
`read,write,readdir`	0	0	1	0	0
`close`	2ω	$\omega(l_n - 1)$	ω	$\omega(l_d - 1)$	$\omega(n * l_n)$
other operations	2	0	1	0	2

Availability Comparison

Coda provides maximum availability: files can be read or written whenever any replica is available. Table 1.2 on page 9 shows the availability of a file for different levels of replication. The `read` and `write` availability are the same. Coda availability is complicated by the fact that Coda clients can operate even if no server is available. The actual availability is even better than normal maximum availability, but that analysis is not directly relevant to this discussion.

Safety Comparison

Coda pays for high availability with lost safety. Assuming that replication is used, the safety values are as follows:

- client cache consistency = *on close*

  Data is flushed back when the file is closed, and client callbacks are used to notify other clients of the change.

- data consistency = *non-directory*

  Data conflicts are left to the user to resolve, but there is automatic reconciliation for directories.

- one-copy serializability = *true*

  All replica versions are checked when a file is opened.

- update stability = *false*

  A client can switch to an obsolete replica due to a crash.

- atomicity = *true*

  Coda uses a built in atomic transaction mechanism.

It is valuable to compare these safety values with Andrew's safety values.

2.10 Summary Tables

Table 2.6 summarizes type of replication offered for each file system in this survey.

Table 2.6: File System Replication Summary

File System	degrees of replication	availability
Andrew	1	maximum
NFS	1	maximum
RNFS	any	almost maximum
Echo	any	majority quorum
HA-NFS	1,2	maximum
Locus	any	maximum
Ficus	any	maximum
Coda	any	maximum

Table 2.7 summarizes the safety properties for each file system in this survey.

Table 2.7: File System Safety Summary

File System	cache consistency	data consistency	one-copy serializability	update stability	atomicity
Andrew	true	true	true	true	true
NFS	timeout	true	true	true	false
RNFS	timeout	false	false	true	false
Echo	true	eventual	true	true	true
HA-NFS	timeout	false	true	false	true
Locus	true	user	true	false	true
Ficus	timeout	non-directory	false	false	true
Coda	on close	non-directory	true	false	true

2.11 Connections

The file system survey produced insights into the past, present, and future direction of file system projects. Section 2.11.1 describes these general observations about distributed file systems. It includes observations derived from file systems that were not presented in this survey. Section 2.11.2 is specific to file systems that replicate based on large objects (e.g. whole disks). Section 2.11.3 is specific to file systems that replicate based on small objects (e.g. files).

2.11.1 General Observations

In the early efforts, such as DFS[SMI80], the designers attempted to provide strong guarantees of consistency since the users were accustomed to working on a single mainframe. Users wanted to have deterministic, atomic operations in the file system. Database techniques and terminology were heavily used since database consistency was a well defined property. As a result, heavy transaction mechanisms were used and the performance was poor. Two-phase protocols were used to commit updates, and mandatory locks were used to insure serializability.

In the later efforts, such as Andrew, the designers felt that they had more latitude. Users had become more sophisticated and comfortable with the network environment. Also, most applications were still not written to generate commits and tolerate aborts, so older experimental file systems did not work. Andrew is an example of a later file system that is simpler and has much better performance than its predecessors. NFS has been simplified to the point where commitment and locks have been totally abandoned. This trend towards simplicity and performance appears to be continuing. Future file systems will depend on the application or user to solve many of the problems associated with distributed systems. It is interesting that the more "sophisticated" file systems are actually simpler.

Another early motivation in these file systems was the economics of disk storage. By using large file servers, the total cost of the system would be reduced because clients would not need individual disks. In more recent times, disk technology has become very cheap, and essentially every workstation has at least 100 Mbytes of local storage. Ironically, the original economic motivation is still valid, but the cost has shifted. Centralized storage reduces the cost of administration. Managing a few large disks is much cheaper than managing many small, scattered disks.

All of the file systems were developed in an academic or research environment. The primary activity of researchers is editing and compiling. Therefore, the file systems are tuned to perform those applications well. Other applications, such as large database processing, have been ignored. Since the author has little experience with these other applications, it is difficult to speculate on what changes to the file system would be necessary. However, it is clear that some changes are necessary. The business community has been very slow to accept new types of file systems, and the narrow scope of these file systems is partly to blame.

In reality, nearly all file systems support replication in the form of caching. Most designers consider caching to be easier and simpler than "real replication." The assumption is that the cache can be dumped if it is obsolete or full, but a replica must be repaired. Also, a stale cache can be tolerated for a short period of time. This assumption is wrong in file systems such as Andrew where the cache is the primary store for new data. The cache in Andrew really serves as a temporary replica with a delayed update propagation strategy. In NFS, the cache can become stale for as long as 33 seconds, and this behavior can cause many problems in applications.

2.11.2 Static Replication

The second category consists of file systems replicate large objects, and these file systems are Echo, RNFS, and HA-NFS. A large unit of data, either a disk or a volume, is completely replicated onto one or more backup disks. All changes are immediately sent to both disks. If the primary host fails, then a backup host can connect to both disks. In Echo and HA-NFS, special disk controller hardware can be used to allow a single host to control all of the disks. High availability and simplicity are the main thrusts.

In these systems, the designers tried to provide high availability with no cost in performance or consistency in comparison to a normal server. They made the smallest possible step from a normal server to a highly available server. The client/server relationship was not addressed as a research issue. Indeed, in HA-NFS and RNFS, the normal NFS communication protocol is used. This approach was first popularized by Tandem Corporation in their proprietary systems. It will continue to be viable in environments where failure is very costly.

2.11.3 Dynamic Replication

The last category consists of file systems that use fine grained replication, and these file systems are LOCUS, Coda, Ficus, and Deceit. They try to use the power of replication to increase the fundamental functionality of the system beyond high availability. Replication can be used on objects as small as individual files.

To a large extent, Coda, Ficus and Deceit are spiritual descendents of LOCUS. The basic LOCUS philosophy is maximum availability: a `read` or `write` must be satisfied unless it is totally impossible. It is only necessary that a single replica is available. The result is that it is possible to produce inconsistent replicas and disjoint file system behavior. Coda takes this concept a step further. In Coda, a `read` or `write` will succeed even if only a *cached* copy is available. Deceit backs off slightly from the LOCUS extreme. In Deceit, users have the option of trading availability for consistency guarantees.

All of these file systems have sophisticated mechanisms for tracking the status of replicas. LOCUS, Coda, and Ficus use version vectors. Version vectors are variable length arrays of counters that summarize the history of updates for a given replica. Deceit uses a more convenient, but less precise, approach involving explicit file versions. In all cases, it is necessary to be able to quickly determine the relationship between two replicas of a file. For example, when a server recovers, it is necessary to check every replica stored on that server against replicas elsewhere.

The problem of inconsistency is particularly acute in directories. Concurrent updates to a single directory are common. Coda, Ficus, and Deceit use automatic directory reconciliation protocols (ADRs) to resolve this situation. Originally, it was intended that LOCUS have an ADR, but that effort was not successful. An ADR uses the semantics of directories to reconstruct a globally consistent directory structure after a partition. In theory, these semantics are well understood, and the solution is apparent. In practice, it is extraordinarily difficult to handle every case correctly. Applications often do perverse sequences of file system operations, such as writing to a file and then immediately deleting it. All of the existing ADRs leave minor gaps and can break applications.

Update propagation strategies differ between the file systems. LOCUS maintains strict synchronization among copies. A `write` operation will not return until all replicas have been updated. This strategy provides highly predictable and safe behavior, but it is unnecessarily expensive for most application since the performance of a `write` is bounded by the slowest server. It is typical of early file system design decisions. Coda synchronizes all replicas after a file has been closed, but changes are locally cached until that time. This strategy is more efficient in general than LOCUS, since communication with servers is less frequent. However, incomplete work can be lost due to a single failure, and shared file access between clients is expensive. Ficus synchronously updates only a single replica, and the other replicas are asynchronously updated by transferring the entire file. This strategy allows the file system to perform as fast as the fastest server, but there is a period of vulnerability after any `write` when the loss of a single server can cause the loss of new data. Also, a small write to a large file will produce unnecessary transfer traffic. Deceit allows a range of behavior from the style of LOCUS all the way to fully asynchronous update propagation. The benefit is that the file system can be tuned to an application, but there is an administrative cost in setting the parameters.

Chapter 3

Deceit Architecture

The Deceit File [BEMS91,SBM89,SBM90a,SBM90b] was designed to allow the semantics of files to be expressed as file parameters rather than be assigned *a priori*. Deceit has the most flexibility among the file systems compared in (Chapter 2). In this chapter we describe the design principles and architecture of Deceit in detail. We follow the description with a performance analysis and a critical discussion of the system design.

The Deceit server program can be coarsely divided into three components as shown in Figure 3.1. The first component is a name service that provides a shared data space for storing immutable values by key. The second component is a segment service that provides bulk data storage and replication. On top of both services is a full NFS file service, called the *NFS envelope*, that uses the two underlying services for storage and communication. The name service stores small blocks of immutable data with very high availability. The segment service stores large blocks of mutable data with a lower availability.

Deceit is a very large and complex program since it must implement many possible file semantics. There are more than a dozen distributed protocols to control different aspects of the system ranging from reliable broadcast to automatic name

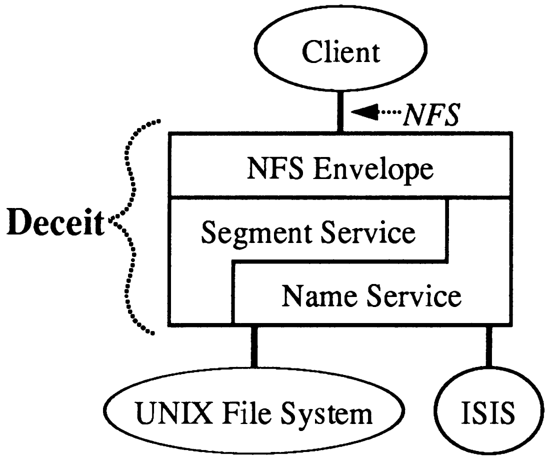

Figure 3.1: File System Components

service reconciliation.

Section 3.1 discusses the overall design of Deceit as well as the environment of the Deceit implementation. Section 3.2 describes the internal name service. Section 3.3 describes the segment service control protocols. Section 3.4 describes mechanisms for handling crashes and recovery in the segment service. Section 3.5 summarizes the parameters that can be used to control several of the replication protocols. Section 3.6 describes several optimizations used to improve the system performance. Section 3.7 describes the NFS envelope. Section 3.8 evaluates Deceit performance using several timed benchmarks. Section 3.9 summarizes Deceit functionality and criticizes some design errors. Section 3.10 speculates on future work for Deceit.

3.1 General Architecture

The main goal of Deceit was to maximize functionality. We intended the file system to provide a wide a range of behavior so that is would be appropriate for almost all applications. One limitation with the current implementation of Deceit is its inability to tolerate partition failures. This limitation is a result of being built upon ISIS.

3.1.1 Design Methodology

Originally, Deceit was proposed as a pure learning experience. Unlike Andrew or Sprite, Deceit was not intended to become a widely used file system. Instead, Deceit was a laboratory for exploring different file system protocols and data structures. In order to maximize the benefit of the experience, we tried to pack as much functionality as possible into Deceit. For many years, file system designers have explored fault tolerance, file migration, flexibility, autonomy, load balancing, and other desirable properties. In Deceit we tried to create an architecture which could capture as many of these properties as possible.

The main result of this approach is complexity. Instead of a clean, simple file system, Deceit has a bewildering variety of protocols and invariants. The value of Deceit stems from the fact that these protocols actually work as a harmonious whole rather than as a nest of conflicts. It was not easy to achieve this position. Deceit has been designed, implemented, redesigned, and revised many times. During this process, we often ran into hard problems or unmanageable explosions in protocol complexity. Some traps were very subtle and depended on the implementation environment. There is no doubt that Deceit still contains design and implementation errors.

Deceit does not tolerate partition failures because ISIS can not. Deceit can tolerate a related failure, called a *virtual partition*[ASC85]. In a virtual partition, two servers can not communicate because there is no period of time in which neither have crashed, yet both servers run independently for some time. For example, server A runs for a few minutes after server B has crashed, then A crashes, then B recovers and runs while A is down. A virtual partition is similar to a real partition since two hosts must act independently yet must not interfere with each other. A virtual partition "goes away" when both servers are running simultaneously and can effectively communicate with each other.

3.1.2 Contrast between NFS and Deceit

One way to understand the Deceit architecture is to contrast it to the NFS architecture, described in Section 2.3 and Appendix A. In a normal NFS implementation, each server machine maintains a set of files disjoint from the sets maintained by all other servers. These sets are structured into directory trees, and each server may provide more than one directory tree. The file name space[1] is built by linking together the directory trees provided by the servers into a single tree. This linking is done separately at each client. Figure 3.2 illustrates a typical NFS client architecture. Clients may communicate with any subset of the NFS servers, but servers normally never communicate with each other. Of course, servers may act as normal clients to each other.

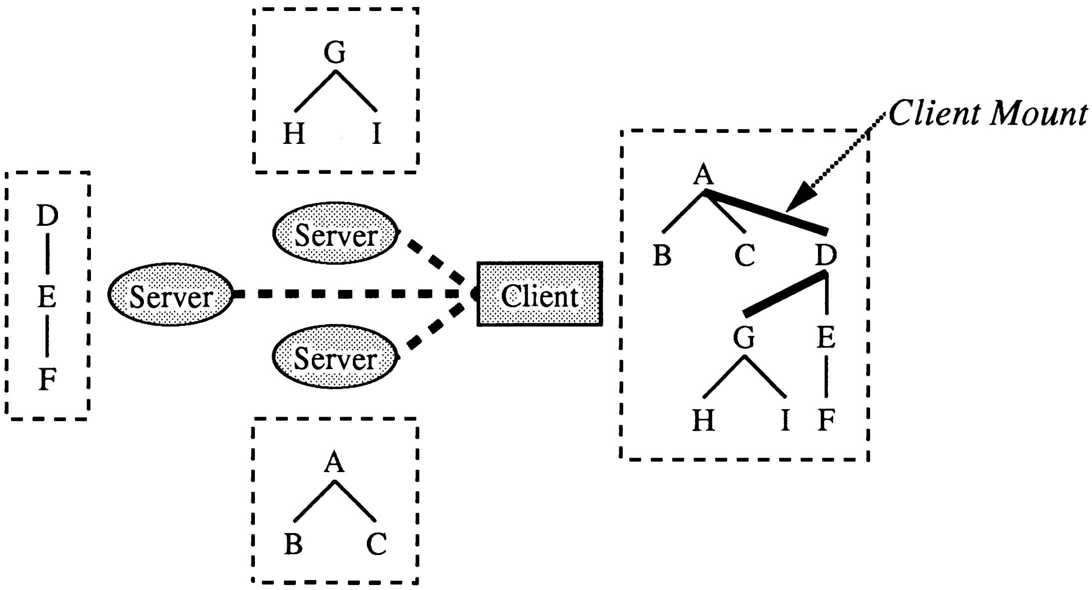

Figure 3.2: NFS Client Architecture

Deceit and NFS use the same client/server communication protocol (i.e. the same transport and RPC interface), so a Deceit server appears to a client to be a

[1]The *file name space* for a file system is the mapping between full path names and logical or physical files.

NFS file server. As a result, Deceit provides a normal NFS name tree. All NFS operations are supported with no change to any client software.

A main difference between Deceit and NFS is that files are not statically bound to any particular server; with Deceit, files may move freely between servers. If a client request arrives for a file at a server which does not have that file, then the request is automatically forwarded to a server that does have the file. The reply is propagated backwards along the same path. All servers provide an identical file service to clients so that clients have to explicitly connect to only one server in order to access the entire Deceit service. Figure 3.3 illustrates a typical Deceit client architecture. The intent of this architecture is to allow clients to switch between servers when a server fails. Unfortunately, the standard Sun RPC implementation does not support this feature.

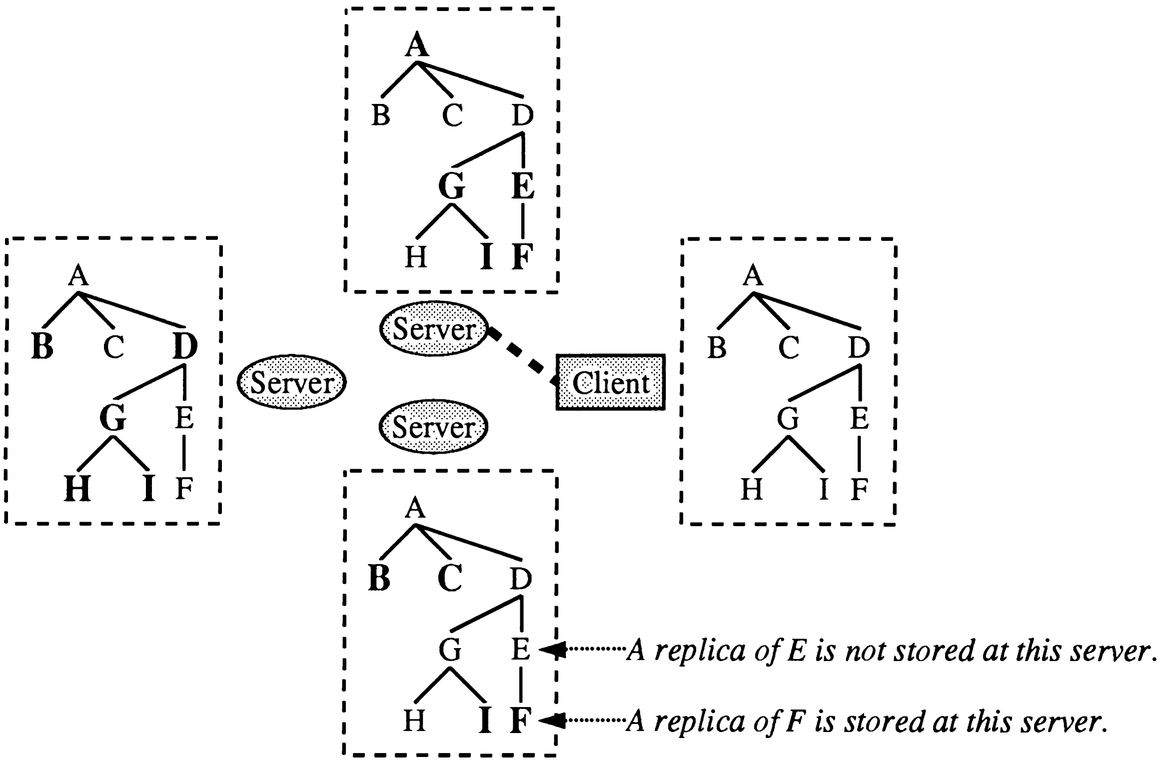

Figure 3.3: Deceit Client Architecture

A second difference between Deceit and NFS is that with Deceit, files may have non-volatile replicas on any subset of the servers. It is important for disk and communication efficiency that files are not always replicated on every server, since such a high degree of replication is unnecessary for most applications. To this end, the user has the option of specifying a desired amount of replication and can provide explicit control over the placement of file replicas.

A third difference is that Deceit supports a file version control mechanism. A user may explicitly create, edit, and delete specific versions of a file. This version control mechanism is "blended" into NFS semantics, and it is accessed by using an optional file name extension. The main purpose of the file version mechanism is to allow the user to resolve conflicting replicas. These conflicts can occur when a file is written concurrently on both sides of a network partition. This failure will be discussed in more detail in Section 3.4.

Deceit provides a superset of NFS functionality. To allow the user to access this functionality, Deceit has additional commands and file operations beyond normal NFS operations. Clients access these features by using special RPCs and by reads and writes to invisible control files. *Special commands* are provided to list all versions of a file, locate all replicas of a file, modify file parameters, reconcile versions, and provide other functions.

3.1.3 ISIS

The Deceit software is not the only component in the Deceit server process. Deceit uses the ISIS Distributed Programming Environment[Mul89,BJKS,BJ87a,BJ87b, JB86] for several fundamental system components. Some features that ISIS provides are: several group broadcast protocols, atomic group membership change, mechanisms for locating group members by group name, light-weight processes with signals and semaphores, architecture independent communication, and process state transfer. ISIS proved to be an excellent environment for prototyping Deceit. Several months of implementation and debugging was avoided, and design changes required far less time to implement. Since Deceit experienced at least a dozen major revisions and countless minor fixes, we greatly appreciated the convenience of ISIS.

ISIS, however, did not provide uniformly positive results. ISIS is a very large, experimental system which constantly evolved during the course of Deceit development. As ISIS developed, Deceit had to be frequently adapted. The reverse was also true: Deceit stressed ISIS in new ways and forced changes. Early versions of ISIS had very poor performance and tended to crash, but recent versions have improved greatly.

3.1.4 Design Assumptions

A list of assumptions about the environment where a system will be used is fundamental to any design. We will provide a short summary of our assumptions here. The assumptions are grouped into three categories: network architecture, failure model, and typical operational behavior.

Network Assumptions

The target environment is a network of computers used in a client/server fashion. Some of the computers may be diskless, and some may be large dedicated file servers. We believe that NFS offers an adequate file system interface for our purposes, and NFS is widely accepted, hence all file requests from the clients are via the standard NFS interface. Under normal conditions, all machines can communicate directly with each other through an underlying network. Communication is symmetric: if a can send a message to b, then b can send a message to a. Since all communication between servers is through the ISIS, all of the ISIS communication assumptions are present in Deceit.

Failure Assumptions

We assume that machines can crash without prior notification[2], and messages can be lost during transmission. Since ISIS does not tolerate long-term network partitions, Deceit can not either. All machines have roughly independent failure probabilities, as described in Section 1.1.

Operational Assumptions

Predictable file access patterns are central to the design and performance of Deceit. Many of Deceit's design decisions were based on results from studies which were done in an academic environment[LZCZ86,OCH+85,Flo86b,Flo86a,Sta88].

Deceit's operational assumptions are as follows. Files tend to be written or read in their entirety with a stream of operations. Nearly simultaneous writes by two clients to the same file are very rare. Files experience long periods of total inactivity punctuated by high activity where they may be rewritten several times in a few minutes. File activity tends to cluster in a small number of directories. The vast majority of NFS operations are `GETATTR` (get basic file attributes), `LOOKUP` (find a file by name in a directory), `READ`, and `WRITE`. Most files are small, i.e. less than 20 kilobytes.

3.1.5 Security

All of the assumptions in Section 3.1.4 skirted the issue of security. Deceit does not directly address most security issues. It is assumed that communication between instances of the server is secure (e.g. encrypted or physically secure). Also, the local files used for storage by the server are inaccessible to unauthorized users. Client/server communication is secured, and client authentication is provided, using DES encryption in the NFS interface.

[2] For a more detailed discussion of machine and communication failure models, refer to [BJ87b].

Clearly, security is a critical facet of a practical file system. However, file system security must be an integrated component of a comprehensive authentication system, such as Kerberos[SNS88]. Since Deceit was built in an environment which lacks an effective authentication system, a decision was made early in the project to provide only minimal security. NFS suffers from notorious security holes for the same reasons.

3.1.6 Implementation Details

Some of the high level architecture in Deceit was driven by very low level implementation details. Deceit is implemented as a user level server process, one process per server machine. All data is stored in local UNIX[3] files. It is implemented on the Sun SPARC architecture under SunOS 4.1.1. It uses Sun RPC facilities for the NFS protocol. It is written in standard ANSI C with full function prototypes. The segment service is 4577 lines long, the name service is 2254 lines long, and the NFS envelope is 4226 lines long as of January 29, 1992.

3.2 Name Service

Deceit contains a name service as a core component. In many ways this service is the glue that holds the entire system together. The name service allows the other components in the system to be much simpler and easier to comprehend. Deceit uses its name service in three ways. First, the name service records a list of locations describing each replica of each normal file: one name service entry for each replica. Second, it stores the contents of each Deceit directory: one entry for each hard link. Third, it stores security information for each file and directory (i.e. owner id, group id, and protection): one entry for each file or directory. The only file system data that is not stored in the name service is the data of the files themselves.

[3]UNIX is a registered trademark of AT&T.

The name service provides the illusion of a shared storage area for data. All servers have a complete copy of the name service data and can independently access it. The name server aggressively provides consistency of the entry data despite this high degree of replication. Updates can originate from any server at any time without preliminary locking or other order determination. These properties can be provided without excessive cost because there are few ordering or exclusion guarantees in the semantics.

The name service data space is divided into *tables* which are identified by an integer value. Each table is an independent instance of the name service. The basic function of the name service is to store a list of *mappings*, each mapping consisting of a *key* and a *value*. Keys and values are simply variable length blocks of data. Keys and values are totally non-exclusive, the same key or value can be used in more than one mapping. Indeed, exactly the same key/value pair can appear more than once. Internally, the name service identifies mappings with unique identifiers, but these identifiers are not exposed by the interface.

There are six core interface routines to the name service:

`init_name(table)` - initialize and join the name service for a table.

`create_name(table, key, value)` - create a key/value mapping in a name service table.

`delete_name(table, key, value)` - delete a key/value mapping in a name service table.

`lookup_name(table, key) -> [key, value] list` - return the list of values for a given key in the form of [key, value] pairs.

`prefix_name(table, key) -> [key, value] list` - return the list of values such that the specified key is a prefix of the key in the mapping. This operation is similar to `lookup_name`.

`flush_name(table)` - atomically flush pending updates for a table. `Create_name` and `delete_name` requests are buffered in memory until `flush_name` is called.

`Flush_name` can be used to form atomic actions. If several updates are flushed together, then the name service joins them together into an indivisible unit. It is impossible for only some of the actions in a unit to be applied at a name server. This property can be used to provide consistency and atomicity for complex operations despite failures in the name service. This guarantee is provided by logging all of the updates atomically in a log file, and by broadcasting all of the updates with a single message. Also, all name service updates are idempotent. Unique identifiers (hidden within the name service) are used to detect duplicate instances of a single creation or deletion.

The name service assumes that all updates are commutative; they may be reordered arbitrarily without breaking the application. However, the order of updates that are grouped together by `flush_name` in maintained. For example, a `RENAME` operation requires a hard link deletion and creation. If these two operations are grouped together with a `flush_name`, then the name service guarantees that they are applied in order and atomically at all servers.

The name service is optimized for the following usage: keys and values are short (i.e. less than 100 bytes), creation and deletion is infrequent, and lookup operations are very frequent. The name service goes to some expense to reliably store data, so it is inefficient to use it for short term or volatile storage.

3.2.1 Name Service Data Structures

All mappings are stored in memory in a single linked list[4]. Skip lists[Pug90] are used to provide fast access. Skip lists provide $O(\log n)$ insertion, deletion, and search with a simple auxiliary data structure. Since the mappings are stored in dictionary order by key, the `prefix_name` operation is also $O(\log n)$.

[4]This list can become very large and will limit scalability. A solution would be to page in/out mappings as in a virtual memory system. We have not implemented this procedure at this time.

It is necessary that every name server of a table have a list of all replicas of that table. This list is called the *site list* for that table, and it is similar to an ISIS group membership list, except that it is not effected by failures. The site list only holds server host addresses and not full ISIS process addresses. When a name server fails and recovers, the site list will not be changed since the host address was already present. The site list is used to determine when an update has been sent to all servers, including servers that are currently down. The maintenance of the site list is discussed more fully in Section 3.2.2.

Update propagation requires some form of stored knowledge about who has seen each update. The name service explicitly represents this knowledge using a bit vector for creation and another bit vector for deletion. For example, let a name server s_1 for table t store a mapping m. Let name server s_2 be in position i in the site list for table t. If bit i be set in the creation bit vector of s_1 for m, then s_1 knows that s_2 knows about the creation of m. Similarly, if bit i be set in the deletion bit vector, then s_1 knows that s_2 knows about the deletion of m. If there are n sites in the site list for table t, and the creation bit vector is equal to $2^n - 1$, then s_1 knows that everyone has seen m.

Servers need to be able to distinguish between multiple instances of a [key, value] pair so that deletions can be consistently executed across many servers. Therefore, two additional integers are stored with each mapping: a creation time stamp (in seconds), and a 32-bit value chosen to be globally unique with very high probability. Together, these two integers constitute a unique identifier since there is a very small probability that two mappings will have the same table, key, value, and unique identifier.

3.2.2 Name Service Update Propagation

The site lists mentioned in Section 3.2.1 need to be in identical order at all servers since they are used to generate bit vectors. The name service depends on the ISIS group membership protocol to accomplish this. An ISIS group is associated with each name service table. When `init_name` is called, the server joins the corresponding ISIS group. The ISIS state transfer mechanism allows the new member to receive a correct copy of the site list that all other group members are using. The site list is also logged to allow correct recovery from total failure. All members monitor group membership and will add a new site to the site list when necessary. Since group membership changes are guaranteed to be ordered in a globally consistent way in ISIS, the site list will remain identically ordered at all servers that can mutually communicate. This protocol is very sensitive to partition failures, therefore it is important that there are no partitions when servers are initially installed in the file system.

The primary method of update propagation uses the ISIS group associated with each table. Refer to Figure 3.4 for an illustration. Let a creation or deletion for mapping m originate at server s. In the first communication round, s broadcasts m (with the operation type) to all members of the ISIS group and collects the replies. Using the reply addresses, s can determine which knowledge bits for m can be safely set. Under normal conditions, nearly all of the bits will be set. In the second round, s asynchronously broadcasts m to all members and does not collect replies. All members update their local knowledge bits from the bits that s generated. If all name servers are up, then this protocol not only gets the update to all servers quickly, but all servers know that all servers have seen the update. The cost is one synchronous round and one asynchronous round of communication.

It is possible to greatly optimize this update protocol by making fuller use of underlying ISIS guarantees (e.g. causality and failure agreement). It is only

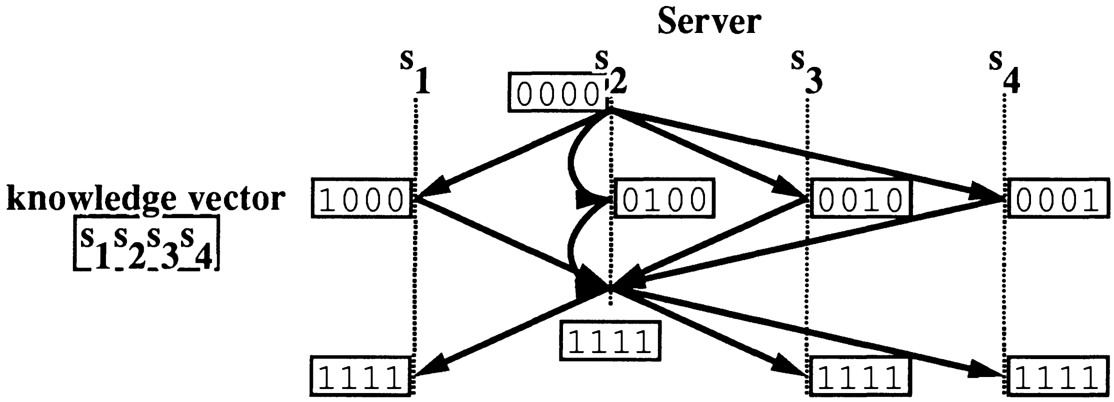

Figure 3.4: Update Propagation in the Name Service

necessary for s to asynchronously broadcast m and for all servers to do a periodic (flush) broadcast. If server u receives a message from v after s has broadcast m to u and v, then u can safely assume that v has received m by causality. Doing do would eliminate the synchronous replies and asynchronous secondary broadcast. Unfortunately, we did not observe this optimization until well after Deceit was built.

The second method of update propagation is invoked on server recovery. Refer to Figure 3.5 for an illustration. When a server s_1 recovers, it requests a `push_name` from some randomly chosen server s_2. To perform a `push_name`, s_2 examines all locally known mappings. If s_2 does not know that s_1 has seen the creation or deletion of mapping m, then s_2 sends m to s_1. The knowledge bit vectors make this search simple. Then, s_1 asynchronously echos each received mapping to all group members so that they will know that s_1 has seen the updates. This protocol insures that a recovering server will quickly see all of the missed updates.

The third method of update propagation is similar to epidemic propagation [DGH+87]. During recovery from total failure, it is possible that some updates are not fully propagated by the previous protocols. To correct this problem, each server performs a `push_name` to another randomly chosen server every two minutes. The secondary echo to all servers is not used in this case. The number of servers that

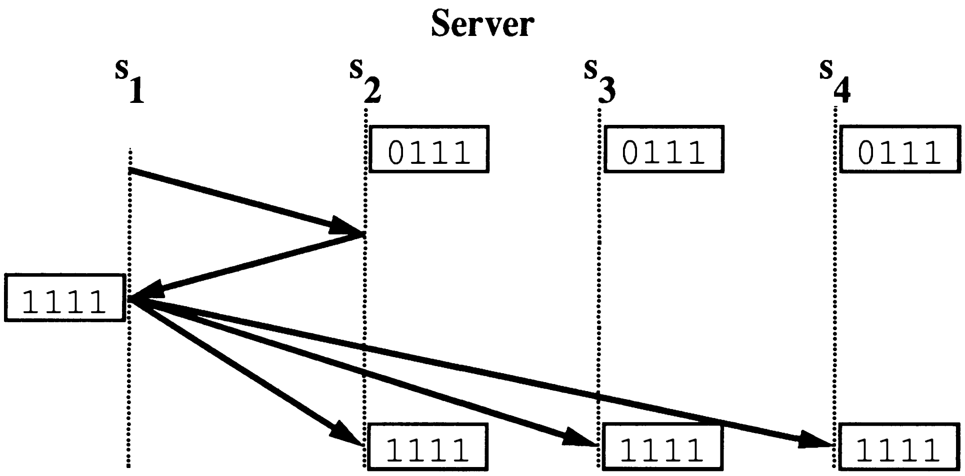

Figure 3.5: Recovery in the Name Service

have seen an update will grow exponentially in time. This protocol is an efficient way of insuring eventual consistency. It also insures that every server will eventually know that every server has seen an update. However, this protocol is not suitable as the primary method of update propagation because it costs $O(n^2)$ messages if only one server has seen an update.

We acquired our experience with these protocol only during debugging and performance evaluation. We believe that the performance will be satisfactory during normal usage, but we can not establish this fact without using Deceit in a real setting. We also believe that the bulk of updates will be propagated with the primary protocol, which is the most efficient method.

3.2.3 Name Service Logging

In order to recover from failure, a name server must log information to stable storage. If server s broadcasts that it has seen a mapping m, then other servers will never be send m to s again. Therefore, before s can broadcast its knowledge of m, s must log m into a file. To prevent unnecessary message traffic after recovery, it is desirable to log m after every change in the knowledge bit vectors.

The policy is that a server s logs a mapping m with its knowledge bit vectors whenever s receives m from another server. The first update propagation protocol will typically lead to two log entries at every site. The first log entry will show only local knowledge of the update, and the second log entry will show global knowledge. A `push_name` will typically cause one log entry at all servers. The receiving server will log the update upon reception, and the echo broadcast will cause all other servers to log the mapping with new knowledge bits.

The knowledge bits are logged intact with each mapping. In order for these bits to be sensible upon recovery, the original site list must also be recovered. Whenever a server joins a name table for the *first* time, it is logged by all servers. During recovery, the site lists are reconstructed in the order that they were originally generated. The consistency guarantees provided by the site list generation protocol carry over into the recovery protocol.

3.2.4 Garbage Collection

There are two important garbage collection problems in the name service: collecting deleted mappings and compacting the log file. When a `delete_name` operation is executed, it only effects the deletion knowledge bits. A mapping which has non-zero deletion knowledge bits becomes invisible to `lookup_name` and `prefix_name` operations, so the deletion appears to occur immediately outside the name service interface. Internally, the propagation of a deletion, and propagation of knowledge of a deletion, can take a long time because of failures.

The policy of the name service is to free a mapping when the creation and deletion knowledge bit vectors are full. More specifically, let server s_1 be a member of a table with n members. If both knowledge bit vectors of mapping m are equal to $2^n - 1$, then s_1 frees all memory associated with m. At this point, s_1 knows that all servers have seen the creation and deletion of m. However, s_1 does not know that all servers know that s_1 has seen it[5]. Therefore, it is possible that another server, s_2, will push m to s_1 later. As a result, s_1 will allocate and log a fresh copy of m. Since s_2 knew that m was deleted, this stutter event will be invisible outside of the name service interface, and the only problem will be a temporary consumption of resources.

The other garbage collection problem is compacting the log file. The name service keeps track of the number of log entries in the file and the number of mappings in memory. When the log file is over two thirds garbage, a synthetic log file is generated containing only current mappings. The old log file is replaced with the new one atomically using a UNIX `rename` system call. Using this policy, the amortized cost of logging a mapping is $O(1)$ including log garbage collection. Also, the size of the log file is bounded to be a constant times the size of the name service data base.

This method of garbage collecting the log file has the advantage of simplicity. Generating a synthetic log file can be accomplished by scanning the linked list in memory and using large file writes. Unfortunately, if there are a very large number of mappings, then this process will become slow and expensive. The name service must halt while the log file is being rewritten. A better method would be to use a generational garbage collector. Mappings are automatically grouped into different log files depending on their age. In this way, garbage collection can be fast and efficient despite a huge number of mappings. This mechanism should be integrated into the mechanism for paging out inactive mappings.

[5]The garbage collection problem in this protocol is identical to the garbage collection problem is causal broadcast

3.3 Segment Service

The segment service is the largest component of Deceit and it contains several interleaved replication protocols. Since it provides a simple, powerful function, it is designed to be reusable in other applications without significant rewriting. It complements and utilizes the name service presented in Section 3.2.

The segment service provides a reliable bulk data storage service. A *segment* is a logical array of bytes that can be indexed by an offset. A segment is referenced with a fixed length handle that is generated by the segment service. The segment service does not implement hierarchal storage, user level security, or user naming.

There are six core interface routines to the segment service:

`create_segment(mode) -> handle` - create a segment and return its handle. The initial protocol control parameters for the segment are encoded in `mode`.

`delete_segment(handle)` - locally delete all storage associated with a segment.

`read_segment(handle, offset, size) -> data` - read data from a segment.

`write_segment(handle, offset, size, data)` - write data to a segment.

`truncate_segment(handle, size)` - truncate a segment to the specified size.

`setattr_segment(handle, access_time, modify_time, mode)` - set the time stamps and control parameters for a segment.

Segment behavior is controlled with several file control parameters which are encoded in the `mode` parameter to `create_segment` and `setattr_segment`. These parameters control the degree of replication, read and write availability, and load balancing. These parameters are described below concurrently with the corresponding protocols, and they are summarized in Section 3.5.

The segment service complements the name service since it is optimized for a different style of data access which is important in the case of normal file contents. Normal files tend to be written in large blocks, and they are frequently rewritten in place, therefore the name service can be very inefficient in this case. In particular, the name service provides a high degree of replication and an inability to update data in place, which could lead to excessive disk traffic. As a result, the contents of all normal files in Deceit are represented with a single segment.

3.3.1 Replica Generation

In the segment service, a logical segment is represented by a number of physical replicas. We will represent the set of replicas associated with a file f with the symbol G_f. Under quiescent conditions, the replicas are symmetric: all replicas are identical and available for reading. During an update this symmetry is temporarily broken, as discussed in Section 3.3.4.

Associated with each segment f is a *minimum replica level*, r, as a control parameter. The value of r is used to control the fault tolerance of f; replicas used for caching are controlled by a different parameter. The segment service maintains at least r valid replicas of f as long as enough servers are available. To do so, new replicas may need to be generated. A new replica of f is placed on a server directly handling a request for f or on the server with the most available storage. In general, there are four ways that a replica can be generated. Method 1 and 2 provide fault tolerance. Method 3 and 4 allow better performance using caching.

1. The number of correct replies is counted after every update broadcast. If the number of replies rep drops below r, then the segment service creates $r - rep$ new replicas.

2. If r is increased, then a sufficient number of new replicas will be created.

3. A user may request the creation of a replica on a specific server with a special command. Users also can inquire about the current location of all replicas of a segment.

4. A server may generate a local replica in order to improve read performance.

Method 4 is simply read caching for performance, and it is controlled by a separate file parameter, not r. Caching imposes a cost is storage space and `write` performance, so it can be turned off by the user. Caching is triggered when a client accesses segment f through a server s that does not have a replica of f. At first, the operation is forwarded to another server that has a replica of f, and the operation is immediately completed. As a background activity, a replica is generated on s to speed future reads and help insure availability. In this manner, whole file caching is achieved with the replication mechanism. Each client slowly gathers its working set of files to the server to which it has connected.

Replicas are generated with a file transfer protocol from an existing replica. A replica holder feeds a copy of the file to the site where the replica is being generated through a steam of ISIS messages. Asynchronous I/O and careful buffer management allow the connection to run at high efficiency. Updates are blocked during replica generation to prevent inconsistency.

If `read_segment` is requested at a server that does not have a replica of the segment, then the request is forwarded to another server. This server is chosen by broadcasting the request to all servers with a replica. The first reply that is received by the original server is used. The address of that server is cached as a hint, so that a broadcast will not have to be used in the future. This method tends to choose servers that are lightly loaded and physically close, and it is easy to implement.

The name service is used to announce the creation or deletion of replicas. When a replica is created, a mapping from the handle to the location is created. When a replica is deleted, this mapping is also deleted. Using the name service, any server can conveniently find all of the replicas of a segment. This search does not require any communication with other servers.

3.3.2 Write-tokens

To serialize updates to replicated segments, we use a *write-token protocol.* This protocol is based on one presented in [Mul89,Sch88]. Exactly one *write-token*[LeL78, LeL81] is associated with each segment. A write-token is similar to an exclusive lock, except that it is held at all times by one server. Only a server that holds the write-token is allowed to distribute updates to the corresponding segment. The holder of a token only requires one communication round to update data, and the update broadcasts can be streamed without waiting for acknowledgements. A write-token protocol works well when update streams tend to originate from one source for long periods of time as in a file system; under these conditions most updates will require only one broadcast.

The ISIS atomic broadcast (abcast) is an alternative to a write-token protocol[6]. Standard abcast is not suitable because it does not provide enough control over the timing of updates. The write-token is a powerful tool for controlling concurrency and recovery. Other protocols in Deceit depend on possession of the write-token for mutual exclusion.

Let S be a server that intends to distribute an update for a file. First, S must acquire the token, which requires one round if S does not already possess the token. To acquire a token, a server broadcasts a `token request` with a *causal broadcast*[BSS90] to all servers in G_f. The current token holder broadcasts a `token pass` in response to change the token address.

After acquiring the token, S broadcasts the actual update to all servers in G_f with a causal broadcast. The use of causal broadcast for token passing and update distribution ensures that updates will arrive in identical order at all sites regardless of token movement. S synchronously collects only the first s correct replies, where s is the *write safety level* of the file. After these s replies have been collected,

[6]In fact, ISIS ABCAST was once implemented using a token passing protocol

the original client RPC that requested the update will return, and the remaining replicas receive updates asynchronously. The semantics of ISIS causal broadcast ensure that all members of G_f will receive the update eventually if they do not fail.

If the write-token holder fails, then the write-token must be implicitly passed to some other server so that `write` operations can continue. Thus, each segment has a *coordinator* server that receives such implicit token passes. The token passing protocol requires that no more than one server holds the token, and that server must know that it holds the token. The coordinator is defined as the server that has been up the longest of all the servers that have a replica of the segment. ISIS provides consistent group membership, and the name service provides consistent lists of replica locations, so any server can independently determine the coordinator for any segment. This determination does not require any communication.

3.3.3 File Groups

The simple definition of G_f in Section 3.3.1 was not totally accurate. G_f, or the file group of f, is the set of all servers that have a replica of f or have any cached information about f (including just a timestamp of f). A server joins G_f before performing any operation on f and leaves after discarding all information related to f. Thus, in the steady-state the membership of G_f is exactly those servers which need to be informed about changes to f.

The fundamental operation within a file group is *update distribution.* An update to f originates from a client and is given to its server. This server then broadcasts the update to all members of G_f. Other segment-related operations, such as token request messages, token pass messages, and other control broadcasts are also restricted to the file group of f. The concept of a file group is fundamental and the primary mechanism to support scalability.

A natural implementation would prefer a file group to be implemented with an

ISIS group, but ISIS process groups have a heavy overhead. Instead, an ISIS group is used to represent each distinct subset of servers[7]. A file group is represented by a specific ISIS group as long as the membership does not change. Thus, when a server joins or leaves a file group, the segment service switches the segment to a different ISIS group instead of changing the membership of the original ISIS group[8] In this way, many segments can share a single ISIS group, and the total number of ISIS groups is limited. Also, joining or leaving a file group can be accomplished with a few causal broadcasts. Unused ISIS groups are slowly garbage collected.

Joining a file group results in obtaining state information about the segment, and this information is specific to a particular point in a stream of updates. Therefore, a join operation must be synchronized with update distribution so that all group members will have a consistent view of the segment state. The only server that can safely perform this operation is the current token holder.

Let s be a server that is to join a file group G_f. Server s sends a message to the coordinator for f requesting the join. The coordinator is always a member of G_f, so the coordinate will have recent knowledge about the location of the token holder and can forward the request to the token holder. The token holder accomplishes a membership change by broadcasting the address of the new ISIS group to all members, including the new member. The current segment state is sent with this broadcast, so that the new member can be immediately synchronized with the other group members. The segment state is expressed in a small fixed-length block of information, therefore this broadcast will not involve large messages. Refer to Figure 3.6 for an illustration.

Since a group join request needs to reach the token holder, a write-token re-

[7]It is possible to need $O(2^n)$ ISIS groups, which can be very expensive for large n. A solution would be to limit the total number of ISIS groups, but occasionally use groups that are too large.

[8]Groups are shared through the simple name service provided by the ISIS group join mechanism.

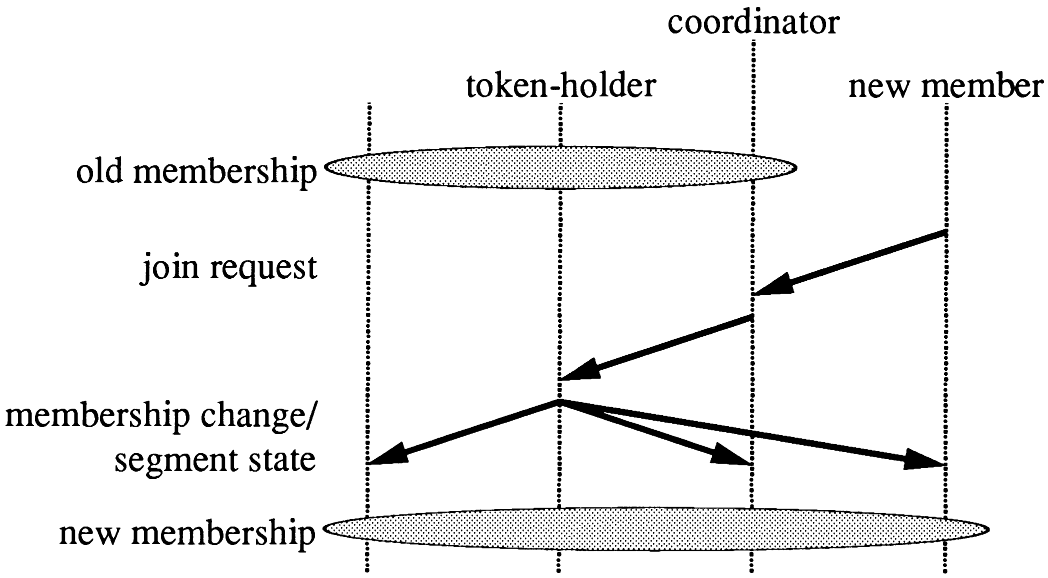

Figure 3.6: Group Join Protocol

quest can be included as an optimization. This action is appropriate when a server wants to join a group and then immediately write to the segment. By using this optimization, extra SNPs and messages can be avoided. Unfortunately, for reasons explained in Section 3.3.4, a server S must determine that all group members have seen the write-token request before S can broadcast updates using that write-token. Therefore, all servers must reply to S after having received a request from the original token holder. Such a form of redirected reply is difficult to accomplish in ISIS, and so we did not implement this optimization.

3.3.4 One-copy Serializability

Since the segment service can return from a `write_segment` before all replicas have been updated, it is easy to devise scenarios where clients see updates appear and unexpectedly disappear. Further, even if the client that initiated the write blocks until the write completes at all servers, other clients that are reading from the segment can read inconsistent intermediate segment states. For example, assume that segments f_1 and f_2 are initially empty. Client c_1 appends to f_1 and then appends to f_2, and c_2 reads f_1 and then reads f_2. If c_2 uses a different set of replicas than c_1, then it is possible for c_2 to read the new version of f_2 and the old version of f_1.

The above scenario is a violation of the safety property *one-copy serializability*[9]. One-copy serializability asserts that to all clients, it should appear that there is only one replica of any file. An update appears totally completed before replying to a `write_segment` operation. This property simplifies client software by elimination of a dangerous source of uncertainty.

Deceit provides one-copy serializability with a simple mechanism called *stability monitoring*. It works by only allowing the clients to use the coordinator's replica if there are concurrent `write` requests. The main benefit of stability monitoring is that updates become visible to all clients simultaneously. By default, Deceit uses stability monitoring, but since there is a small overhead from the extra token requests and the forwarded `read_segment` requests, the user can turn stability monitoring off. Note that stability monitoring does not guarantee one-copy serializability in the presence of failures. A crash can force a client to switch to another replica, and the other replica can be in an earlier state.

The stability monitoring protocol is as follows. A server holding the token must broadcast a token request before updating a segment if it has seen no token request for more than 10 seconds, and all replies are synchronously collected for this broadcast. This policy insures that all servers will be warned before a segment is modified. If a server s sees a token request for segment f, then s knows that its cached state of f can be inconsistent for the next 10 seconds. More importantly, if s has not seen a token request for 10 seconds, then s knows that all replicas are

[9] In file systems one-copy serializability is usually defined assuming all interprocess communication is through files.

identical. Of course, there is a little bit of uncertainty in these times because update propagation is not instantaneous. The policy is that all `read_segment` requests are forwarded to the coordinator for 15 seconds after a token request. After that time, `read_segment` requests are sent to any convenient replica.

The use of a 10 and 15 second time-out is a somewhat arbitrary design choice. The protocol is not particularly sensitive to this value except in the extreme cases. We chose 10 because it is longer than most update bursts which reduces `write` overhead, and it is shorter than the write/read cycle for most users which reduces `read` overhead. Although it could be expressed as a file parameter, we believe that arriving at a more appropriate value is too subtle for most users to understand.

An earlier version of Deceit used an explicit round of communication to terminate stability monitoring. The token holder would broadcast a message after a sufficient period of inactivity. Unfortunately, this method led to a complexity explosion in the recovery protocols. On recovery, a server had to determine which servers were forwarding `read_segment` requests and when the last update occurred in order to bring the group back to a consistent state. Often, this process involved blocking `read` and `write` operations temporarily which in turn led to more fault tolerance problems. A time-out protocol has the nice property that it will always recover to a consistent state without any global operations.

3.4 Partition Failures

During a partition, it is possible that there are two active write-tokens. Even if there is no actual partition, ill timed crashes and recovery can produce the same effect on replicas. Under these conditions, replicas can become badly inconsistent. This problem is thematic in file systems that support replication[WPE+83,SKK+89], and it is the *version splitting* problem. Deceit handles this problem with two mechanisms: *write quorums* [vRT88,P$\hat{8}$9,P$\hat{8}$6] and *history descriptors* (described in Section 3.4.2).

This use of write quorums is similar to the traditional use of read/write quorums to assure update atomicity, but they should not be confused. Deceit does not use a read quorum, and updates are always applied to all available replicas.

3.4.1 Write Quorums

Let a_f be the number of available replicas for segment f with file group G_f. Before broadcasting a file update, a server verifies that a_f is at least as large as the *write quorum size*, q; otherwise, `write_segment` and `truncate_segment` requests are rejected, and replica generation is disabled. Whenever there is a failure or recovery in G_f, ISIS reports the change to Deceit, and Deceit recomputes a_f[10]. Using a large value for q reduces the probability of version splitting at a cost of reducing the availability of f for writing.

In order to positively prevent inconsistent replicas, another file parameter, the *maximum replica level*, m, is used. Replica generation is inhibited if there are at least m replicas, including replicas on failed servers. If $m < 2q$, then it is impossible to have more than one effective write-token under any circumstance. Note that a small m limits the read performance and flexibility.

After a failure, there is a short period of time before ISIS detects it and reports it to Deceit. During this time, it is possible to have multiple active write-tokens despite a large value for q. This scenario is exceptionally unlikely since it requires two concurrent updates immediately after a network partition on either side of the partition. One completely safe solution is to use a two-phase commit with every update. We decided that this solution was too expensive as any rare inconsistency that results would be handled by the history descriptors, described below.

Traditionally, the version splitting problem is handled by blocking servers in the minority partition, since each server is either fully operational or fully blocked. The

[10] This computation actually occurs the next time there is an update for f.

Deceit solution has the advantage of much higher expected availability when files tend to be close to the clients that write them. Since the majority is computed on a per file basis, it is likely that a minority partition will have a majority of the replicas of a needed file. If a user is in the minority partition, then he will be able to access most of his important files.

3.4.2 Version Splitting

Two replicas are considered to be *comparable* if one has received a prefix of the updates that the other has received. Version splitting is easily handled if all replicas are comparable; since obsolete replicas can be discarded. It is a fact that it is impossible to produce incomparable replicas unless new write-tokens are generated[11]. Deceit uses a mechanism to track exactly which updates have been applied to which replicas, and it is closely tied to write-tokens. This information is then used to detect and resolve inconsistency during recovery.

History Descriptors

Associated implicitly with each replica of file f is its update history H. H of f is a sequence of all updates to f starting from an empty file. $H' \prec H$ if H' is a prefix of H. $\prec$ is a partial order and can be represented as a tree of versions rooted at the initial, empty f. We call this tree the *history tree* of f.

Deceit frequently has to determine the relationship between two replicas to determine if one replica should be discarded or separated from the others (e.g. during recovery). The history tree contains the information needed, but Deceit does not explicitly store the full history tree since it would consume an unbounded amount of storage. Instead, Deceit records the state of a replica using three values: the *version number*, the *token signature*, and the *update counter*. These values together are called the *history descriptor* for a replica, and they are stored with the replica.

[11]Reverting a write-token to the coordinator for a segment is a form of token generation.

- The *version number* of a replica is a high level description of the history tree branch of a replica. Replicas with different version numbers are logically separate segments each with its own file group. The version number for a segment is encoded in the handle for that segment, and the client specifies different segment versions by using different handles.
- The *token signature* of a replica denotes the owner of the write-token when the last update was sent to that replica. Whenever a write-token is passed, the new token holder chooses a random 32-bit value to use as the token signature. The signature is sent in each update, and replicas are marked accordingly.
- The *update counter* is a simple serial counter for updates. Whenever a server receives an update for a replica, then the update counter on the replica is incremented.

The history descriptor provides a very efficient way of determining if two replicas are different or comparable. Two replicas with the same update history will have the same history descriptors, and two replicas with different history descriptors must have diverged at some point. If two replicas have the same version number and token signature but different update counters, then they are comparable. However, the converse is not true, so it is possible to identify comparable replicas as incomparable.

A history descriptor is also stored with each write-token, and it reflects all updates that have been delivered through that token. These history descriptors are used to restore a segment to a consistent state during recovery. Assume that a recovering server s has a replica of segment f, and so rejoins the file group for f. During the join, s will learn the history descriptor of the write-token as part of the state transfer in the join protocol. If s has a replica with the same history descriptor as the write-token, then the replica is consistent with the other replicas in the file group; otherwise, the inconsistency must be resolved. If the token signature agree, but the update counter of s's copy of f is too small, then the replica is simply obsolete and can be deleted. Otherwise, a new version is generated (described below), which effectively turns s's replica into a separate file. After the appropriate recovery action, s joins the group as if s had no replica of f.

Some file systems (i.e. the Coda File System) use vector timestamps to perform the same function as history descriptors. Vector timestamps have the property that they will always exactly identify the relationship between two replicas. History descriptors will sometimes identify comparable replicas as incomparable, resulting in extra versions. However, virtual timestamps are larger than history descriptors and have variable length, so the there is an increase in storage and system complexity. For this reason, we elected to use history descriptors.

Version Control System

Generating a new version from a replica is very similar to creating a new segment. One difference is that the new version is initially non-empty. It uses the incomparable replica from the old version as the first replica of the new version. The other difference is that the segment handle for the new version is nearly identical to the segment handle for the old version. A globally unique version number is chosen for the new version and this number is embedded in the new segment handle in the place of the old version number.

It falls on the user to reduce multiple versions for a file to a single version. By allowing the user to resolve incomparable versions, the semantics of the file may be used for resolution[Bre83,WPE+83,HPR88]. The facility by which users resolve conflicts can also be accessed directly at the user level as a normal file versioning system, such as in a source code management system. Deceit uses a simple mechanism: file names can be qualified with version numbers using a special syntax. For example, version 3 of "foo" can be referred to as "foo;3." Deceit captures the file name extension and transforms segment handles appropriately. By using this form of file name, specific versions can be modified and deleted. By using an unqualified filename, the user automatically requests version 0, which is the initial version. The system behaves similarly to the VAX/VMS[12][Dig84] version control system, except that VMS produces a new version on every file update, while Deceit produces new versions only during partitions or when explicitly requested.

3.4.3 Recovery Scenarios

In order to clarify the usage of the crash resilience mechanisms, several detailed scenarios are presented.

Replica Recovery

Let s be a server with a replica of segment f. Assume that s crashes and recovers. Upon recovery s will attempt to join the file group for f. If no other replicas are available, then s becomes the coordinator for f, and s can produce a new write-token for f using the history descriptor of its replica.

If s is not the coordinator for f, then the file group and a write-token already exist for f. Server s sends a join request to the coordinator for f. The join request is forwarded within the file group until it reaches the current token holder, t. Server t generates an ISIS group containing the old membership plus s, and t broadcasts this group address to the new group along with the current history descriptor for f. When s receives this message, s can determine if the state of its replica matches the state of the rest of the replicas. If there is a match, s is recovered.

If s finds that its replica is inconsistent, then s can either delete the replica or turn it into a new version of f. The replica is deleted only if the update counter for s's replica is too small, but the rest of the history descriptor is correct. Otherwise, a new version is generated, and it is the responsibility of the user to reconcile the

[12]VAX/VMS is a Trademark of Digital Equipment Corporation

two versions. If this occurs, an entry in a special log file is created so that the user can detect the event. The user can use the version control system and normal file editing to reduce the segment back to a single version.

Orphaned File Groups

Let G_f be a file group for segment f. If all the servers in G_f that have replicas of f fail, then there may remain servers that have lost contact with f. When a server with a replica eventually recovers, it will be the coordinator for f, and it will start a new file group G'_f. Meanwhile, the members of G_f will not receive new updates for f and are therefore isolated.

This condition is corrected in two ways. First, if a member s of G_f attempts to read or write f, s will receive no replies. At this point s can unilaterally leave G_f and return an error to the client. The next time s tries to access f, it will rejoin the group for f and thus will eventually become a member of G'_f. Second, there is a time-out on group membership for servers that do not have a replica of f and are not the token holder. After 30 seconds, s will unilaterally leave G_f. A later access to f will cause s to rejoin G'_f as before. The time-out is also convenient for reducing the size of inactive file groups.

Partition

Now consider the scenario where there is a persistent network partition as shown in Figure 3.7[13]. Recall that the write-token reverts to the coordinator if a token holder crashes. A new write-token will be generated in partition B, since it will appear to B that all servers in A failed. Access continues normally in both partitions since it is difficult to distinguish between this scenario and the case where the other replicas simply crashed. When the partition terminates, the servers in one group must rejoin the other group. Assume that the servers in partition B are forced to rejoin the

[13] ISIS does not support recovery from partitions, and so this scenario is speculative.

file group in partition A. Since the replicas in B are possibly inconsistent with the replicas in A, new versions can be generated at this time.

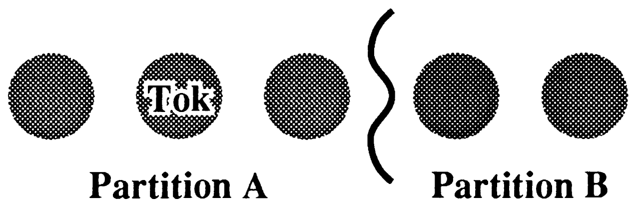

Figure 3.7: Network Partition

The write quorum size can be used to control updates during a partition. If the write quorum size is at least half of the maximum replica level, then reconciling versions after a partition is easy, since all versions will be an ancestor of the most recent version. It is common in network environments for a single server to be temporarily separated from the network due to congestion or failure in the network interface[Cri91]. Therefore, a write quorum size of 2 should handle nearly all practical situations.

3.5 File Parameter Summary

Several file parameters were mentioned in previous sections. These parameters are associated with each segment. Below we summarize the file parameters:

- Minimum Replica Level (r) - the minimum number of valid replicas that must be maintained. For example, a minimum replica level of 3 forces Deceit to maintain a valid replica on at least 3 separate servers. This parameter was discussed in Section 3.3.1. By default, the value is 1.
- Write Safety Level (s) - the number of replica servers that must reply to an update before a `write_segment` returns. A value of 0 produces asynchronous and unsafe writes; a value greater than or equal to the number of available

replicas produces slow or fully synchronous writes. This parameter was discussed in Section 3.3.2. By default, the value is 1.

- Stability Monitoring - specifies whether stability monitoring is to be used. It reduces the probability of non-one-copy serializable behavior, but there is a small performance cost. This parameter was discussed in Section 3.3.4. The default is to use stability monitoring.

- File Migration - denotes whether Deceit automatically attempts to create a replica of file f on a server that receives a request for f. For some applications, it may be bad to automatically generate local replicas. For example, a single reference to a very large data file, may unnecessarily consume excessive disk space. This parameter was discussed in Section 3.3.1. The default is that file migration is not used[14].

- Write Quorum Size (q) - the minimum number of available replicas required for updating a segment. A value of 1 produces no restriction of writing. This parameter was discussed in Section 3.4.1. By default, the value is 1.

- Maximum Replica Level (m) - the maximum number of replicas. This parameter is used in combination with the write quorum size to prevent multiple file versions. If $m < 2q$, then it is impossible to produce incomparable file versions. This parameter was discussed in Section 3.4.1. By default, the value is 5 since this will allow generous, but bounded read caching.

3.6 Optimizations

Several optimizations are used to improve the segment service performance. The first optimization reflects the fact that Deceit runs as a single server process, and

[14]Another approach would be to implement sophisticated heuristics to determine when to migrate a file. For example, some combination of file size and number of `read` operations from a particular client could be used as the determining factor. We have not implemented this idea.

normal file reads or writes block the entire process. SunOS 4.1.1 provides true asynchronous read and write capability. The segment server uses this facility in combination with the ISIS light weight tasks to provide concurrent file access for concurrent NFS requests. A read or write will block a single thread until it completes, but it will not block the entire process.

Deceit prefetches data after a read in order to take advantage of normal read sequentiality. For example, a read at position p of length l will cause another read at position $p + l$ of length l to be started asynchronously. If a request for this data arrives later, the request will block until the asynchronous read has completed, and the request will use that data. Under lightly loaded conditions this method significantly improves the observed performance of the server, as discussed in Section 3.8. This rendezvous requires a data structure to store temporary read results as well as garbage collection for unreferenced reads.

A large in-memory data structure is used to store meta-data about segments. Updates to this data structure needs to be stored on disk for safety. In order to simplify this process, this data structure is mapped into virtual memory directly from a backing store file. The operating system automatically propagates changes to the backing store, and no explicit kernel calls are required. On recovery, the backing store is mapped back into memory, and the data structure is checked for consistency. Unfortunately, there is a possibility of inconsistency between the files used to store segments and the descriptions of the segments in the meta-data backing file; we currently do not implement a solution to this problem.

3.7 NFS Envelope

The NFS envelope provides the full NFS service using the name service and the segment service. While the NFS envelope is not nearly as complex as the other components, it is a large part of the server program. In this section, we will summarize the details of the NFS envelope.

3.7.1 Usage of Other Components

The NFS envelope uses the name service in two ways. One name table is used to record the directory structure. Mappings in this table take the form <directory handle, "d", name>⇒<child handle>. The "d" signifies that the entry is a normal "down-link." Thus, listing a directory involves listing all mappings with the prefix <directory handle, "d">. For each down-link, there is a matching "up-link" of the form <child handle, "u", "..">⇒<directory handle>. Thus, counting the number of links to a file involves counting all mappings with the prefix <child handle, "u">. The up-links are used to quickly determine when all down-links to a file or directory have been deleted, and storage for the file can be freed.

The NFS envelope uses another name table to store owner and security information for each file or directory. Mappings in this table take the form
<handle>⇒<UNIX mode, user id, group id>. All file access checks are made using this information. The security information is changed by deleting the old mapping and creating a new one. In order to insure safety, if no security mapping can be found for a file, then all access is denied.

Often there will be several name service changes for a single NFS request. As an optimization, the NFS envelope groups all name service changes for a table into a single name service update by using a single `flush_name`. If there are updates to multiple tables, then parallel `flush_name` calls are issued. Performance for several updates is essentially the cost of a single update plus a small amount of CPU time.

The NFS envelope uses the segment service to store the contents of normal files and and the link data for soft links. Each normal file is associated with a segment using its handle. `READ` and `WRITE` requests for a file are translated into `read_segment` and `write_segment` calls for the associated segment. `READLINK` and

`SYMLINK` requests receive similar treatment.

3.7.2 Details of the NFS Envelope

In order to fully describe the NFS envelope, we will summarize the action required for each type of NFS request. This information will be used in Chapter 4 in the performance analysis of Deceit. Refer to Appendix A for a description of the meaning of each NFS request type.

`LOOKUP` Operation

The execute permission is checked on the directory using the security name table. The child file handle is determined using the directory name table. The result file attributes are computed.

`WRITE` Operation

If the file handle is for a control file (version number -1), the write is actually a special command. The ISIS message library has routines providing an architecture independent data format, which is used to parse the arguments of the special command. The only special command currently supported is setting the Deceit mode bits for a file. This command is executed by calling `setattr_segment`.

If the file handle is for a normal file, then the write permission is checked for the file using the security name table. The corresponding segment is determined from the file handle, and a `write_segment` call is issued.

`REMOVE` Operation

The write permission is checked for the directory using the security name table. The child file handle is determined using the directory name table. The down-link and the matching up-link are deleted. If the number of up-links is now zero, the security information is also deleted. A callback is used on the directory name table so that each server can monitor deletion of up-links on files. When the number of up-links becomes zero at a server, that server deletes all versions of the corresponding segment locally with a `delete_segment` call. In this way, disk storage is reliably freed at all servers when a file is deleted.

All name service operations are logged to disk for stability. Further, careful use of `flush_name` insures that a complex operation (e.g. `RENAME`) is executed atomically. If there is a crash and recovery, the operation will either have completed totally or have had no effect.

`RMDIR` Operation

This operation is nearly identical to the `REMOVE` operation, except no segment deletion is required.

`LINK` Operation

The write permission is checked for the target directory using the security name table. A new down-link and up-link are created for the file.

`RENAME` Operation

The write permission is checked for the target directory using the security name table. If a file already exists at the target location, it is deleted as in the `REMOVE` operation. Then, the old down-link is deleted for the file being moved, and the new down-link is created. Finally, the old up-link is deleted, and the new up-link is created. These name service changes are grouped into a single update by using a single `flush_name`.

A problem with directories is that it is possible to produce a closed loop by moving a directory into its own descendent. This possibility is checked using the directory name table before the directory is moved. This solution is not completely safe because two concurrent directory movements can jointly produce a loop.

STATFS Operation

The file system statistics are determine using a normal UNIX `statfs` call on the local segment storage directory.

GETATTR Operation

The file attributes are determined using the method described in Section 3.7.1. Note that the name service caches entries in memory, and the segment service caches segment meta-data in memory. Therefore, this operation usually will not require disk access.

SETATTR Operation

Owner user id, group id, and UNIX mode bits are changed by deleting the old security name mapping and creating a new security mapping. Time stamps are changed by using the `setattr_segment` call. The file size is changed using the `truncate_segment` call. Once the appropriate changes are made, the result file attributes are computed.

READ Operation

If the file handle is for a control file (version number -1), a description of all versions of the corresponding segment is computed. The result is passed by using the ISIS message library which provides an architecture independent data format. By reading the control file, the user can determine all Deceit specific information about a file.

If the file handle is for a normal file, then the read and execute permission of the file is checked. It is possible to read a file that has only execute permission, since the NFS server can not determine if a file is being used as an executable. The file data is read using a `read_segment` call. If the result of a previous `read_segment` corresponding to the same data is cached, that result is used instead. The result file attributes are computed. The RPC reply is then sent to the client, but the NFS envelope continues working. The next block of data is read and cached to allow better concurrency when a file is being read sequentially.

`READLINK` Operation

The link data is read using a `read_segment` call.

`READDIR` Operation

The read permission is checked for the directory. The directory entries are computed by calling `prefix_name` using <directory handle, "d"> as the key on the directory name table. The ".." entry is computed using <directory handle, "u", ".."> as the key.

`CREATE` Operation

The write permission is checked on the directory. If a file already exists at the specified name, it is truncated to zero length using `truncate_segment`. Otherwise, a new segment handle is generated. A new security name table mapping is created. A new down-link and up-link are created. A new segment is created using a call to `create_segment`. The result file attributes are computed.

`MKDIR` Operation

The write permission is checked on the parent directory. A new segment handle is generated (this handle does not refer to a real segment, but the same format is used for convenience). A new security name table mapping is created. A new down-link and up-link are created. The result directory attributes are computed.

`SYMLINK` Operation

The write permission is checked on the directory. A new segment handle is generated. A new security name table mapping is created. A new down-link and up-link are created. A new segment is created using a call to `create_segment`. The symbolic link data is written to the segment using a call to `write_segment`. The result file attributes are computed.

3.8 Timed Benchmarks

In this section, we present a detailed analysis of Deceit performance using different client and server configurations. The natural way to analyze a file system is to determine the cost associated with each type of operation. There are two aspects to this cost: the minimum time to complete the operation and the resources consumed while the operation is in progress, where resources can be CPU time, disk access, and network traffic. An operation that that executes very quickly but consumes all available resources will be unacceptable in an environment with heavy concurrent load. Similarly, a very slow operation will be unacceptable even if it consumes few resources.

The analysis of Deceit is made more complex by the ability to change file system behavior through the use of file parameters. The focus of the analysis will be on the relative performance of Deceit using different file parameters settings, instead of the absolute performance. Deceit was never heavily tuned (as opposed to NFS), so absolute performance is not very instructive.

3.8.1 Experimental Setup

The benchmarks were performed using the following environment. The NFS clients were Sparcstation IPCs, and the servers were Sparcstation 2's with 48 megabytes of physical memory. All machines were running version 4.1.1 of the Sun operating system. All hosts were connected with a single Ethernet network having a bandwidth of 1.25 megabytes/second. The disks were Maxtor model LXT213SY with an average seek time of 15 milliseconds and a maximum bandwidth of 3 megabytes/second.

The software was built using version 3.0.4 of ISIS which was last installed on November 11, 1991. Deceit was compiled with the GNU C compiler version 1.40 from the Free Software Foundation, and normal compiler optimization was used. Three versions of Deceit were used in the benchmarks. In addition to the normal version, a special version of Deceit was written which eliminated all disk access. This version was used to separate the network and CPU cost from the disk access. Also, a version was written which increased the cost of a disk operations by a random amount (chosen uniformly between 0 and 50 milliseconds). This version was used to explore the effect of random background load on Deceit performance.

The test machines were part of the general Cornell Computer Science department environment, but the benchmarks were performed at a time when there was very little activity. In addition, before each benchmark, the clients and servers were checked to insure that no extraneous processes were executing. Except for the Andrew Benchmark[15], each benchmark was executed many times in order to produce a statistically valid mean. The resulting values are given with a 95% confidence interval using the techniques described in [WW90].

3.8.2 Andrew Benchmark

The first benchmark is a measure of overall file system performance using the Andrew Benchmark[16]. The Andrew Benchmark contains a mix of file system operations that are meant to emulate normal usage. The benchmark builds a tree of source files, scans each directory and file, and then compiles all of the files. This benchmark executed the file systems operations by making user level UNIX file system calls on an NFS client. The client software added uncertainty to the times, particularly since a Sun NFS client caches data and delays operations.

[15]The Andrew Benchmark was very time consuming to execute. We decided that the value of collecting a statistically meaningful sample was not justified by the effort required.

[16]This benchmark is based on the distribution from June 14, 1985. It was provided by M. Satyanarayanan, the Information Technology Center, and Carnegie-Mellon University.

The times for the Andrew Benchmark are shown in Table 3.1. Deceit was tested three times using a default of one, two, and three replicas for a file. For example, Deceit(2) indicates that two replicas are created whenever a new file is created. The write quorum size and the write safety level were the same as the number of replicas, file migration was disabled, and stability monitoring was used. This choice of settings provided similar semantics between NFS and Deceit (except for availability). The timing resolution is measured only in seconds. For Phase I (creating directories), the times are very short and resolution is inadequate, so those results are not shown.

Table 3.1: Andrew Benchmark Times (seconds)

Phase	NFS	Deceit(1)	Deceit(2)	Deceit(3)
Phase II (Copying files)	18	29	57	71
Phase III (Directory stats)	19	17	18	19
Phase IV (Scanning each file)	25	26	25	28
Phase V (Compilation)	86	92	104	113
Total	151	166	210	243

For the total Andrew Benchmark, Deceit(1) is 9.9% slower than NFS. The largest performance difference is in Phase II which is dominated by heavy `WRITE` activity. Since Deceit does not run in the kernel, it is not capable of scheduling the disk as efficiently as NFS, hence the poorer performance. Also, a user level process requires more memory copying than a kernel process because data must move between virtual memory spaces. Phase V suffers from a similar effect to a lesser degree.

Replication adds a significant overhead: for the total benchmark, Deceit(2) is 26% slower than Deceit(1) and Deceit(3) is 46% slower. Phase II and Phase V are characterized by a large amount of writing, so the cost of replication is most prominent in those phases. Replication has the most effect in Phase II: Deceit(3) is 145% slows than Deceit(1). The reason is that Phase II involves many file creations, which in turn invoke a `CREATE` and a `SETATTR` NFS call. Table 3.2 in the next section shows that both of these calls are strongly effected by replication.

3.8.3 Overall Operation Cost

The results in Table 3.2 are a measure of straight NFS performance. A special test program executed each NFS RPC type repeatedly and averaged the execution times. The calls were made by directly sending network messages containing the RPC data rather than going through the normal client software. The results are much more consistent and precise than those in Table 3.1. A percentage is included in the table that indicates the ratio between the number of times each call was invoked to the total number of calls in the Andrew Benchmark. As with Table 3.1, Deceit was measured using one, two, and three replicas as the default for each file.

Table 3.3 shows the same benchmark, but the normal version of Deceit has been replaced by the version that does not actually access a disk. Therefore, the times in Table 3.3 indicate the CPU and network transmission time required for each operation.

The most prominent time differences between NFS and Deceit(1) are in `CREATE`, `REMOVE`, and `SETATTR`. In the case of `CREATE`, Deceit is not able to efficiently schedule disk access. Deceit must execute two disk operations: appending the name service log and creating a storage file. If both of these disk accesses were executed in parallel, then the time would be cut to nearly half, but Deceit does not do this. In the case of `REMOVE` and `SETATTR`, the NFS implementation violates its own safety properties and does not execute the operations synchronously. Consequently, NFS is much faster than Deceit for these RPC types.

Deceit is slightly faster than NFS for `LOOKUP`, `GETATTR`, `READ`, and `STATFS`. The

Table 3.2: NFS Benchmark Times (milliseconds per operation)

RPC Type	Usage	NFS	Deceit(1)	Deceit(2)	Deceit(3)
LOOKUP	37%	4.6 ± 0.2	4.3 ± 0.3	4.1 ± 0.2	4.1 ± 0.3
GETATTR	36%	3.7 ± 0.2	3.2 ± 0.2	3.2 ± 0.3	3.0 ± 0.1
READDIR	15%	13.6 ± 1.4	12.3 ± 1.3	12.8 ± 1.3	12.0 ± 1.2
WRITE (8 Kb)	8%	129.3 ± 2.2	138.3 ± 1.6	148.7 ± 2.9	162.0 ± 3.0
CREATE	3%	59.0 ± 5.4	124.6 ± 7.1	219.9 ± 7.6	226.7 ± 7.6
READ (8 Kb)	0.7%	20.6 ± 0.8	18.4 ± 0.6	18.8 ± 0.7	19.1 ± 0.8
MKDIR	0.5%	116.5 ± 5.8	78.9 ± 8.3	235.7 ± 9.3	278.0 ± 17.4
REMOVE	0.2%	14.7 ± 0.8	141.3 ± 8.0	422.4 ± 10.6	532.6 ± 17.7
RENAME	0.1%	64.2 ± 0.4	53.5 ± 8.9	164.3 ± 10.4	168.1 ± 6.0
STATFS	0.03%	3.7 ± 0.2	3.4 ± 0.4	3.3 ± 0.3	3.2 ± 0.3
NULL	0%	2.5 ± 0.2	2.5 ± 0.1	2.6 ± 0.1	2.4 ± 0.1
SETATTR	0%	3.6 ± 0.2	85.7 ± 5.4	471.3 ± 4.4	639.3 ± 13.9
RMDIR	0%	61.6 ± 2.9	65.9 ± 2.0	214.2 ± 15.0	218.4 ± 10.8

reason is that Deceit is more aggressive about keeping information in memory. The entire directory structure is in the name service which is held in memory, and this fact explains the fast `LOOKUP` and `GETATTR`. Deceit caches the result of a `STATFS` for several seconds so that repeated calls will be very quick. Deceit prefetches data in the `READ` operation, and that optimization provides slightly better performance than the simple caching the NFS uses.

Table 3.2 shows that a `WRITE` using three replicas is 17% slower than when using one replica. However, the disk access time is approximately 116 milliseconds in all three cases. This fact indicates that Deceit is doing a good job of performing the disk access in parallel on all servers. Since the disk access consumes 70% to 87%

Table 3.3: NFS Benchmark Times without Disk Access

RPC Type	Deceit(1)	Deceit(2)	Deceit(3)
LOOKUP	4.2 ± 0.2	4.2 ± 0.2	4.3 ± 0.3
GETATTR	3.2 ± 0.1	3.3 ± 0.1	3.2 ± 0.1
READDIR	12.6 ± 1.3	12.5 ± 1.5	12.8 ± 1.7
WRITE (8 Kb)	18.3 ± 0.5	38.1 ± 1.0	49.5 ± 1.8
CREATE	12.3 ± 2.9	38.2 ± 3.7	51.7 ± 4.8
READ (8 Kb)	17.9 ± 0.6	17.9 ± 0.5	17.7 ± 0.7
MKDIR	9.9 ± 0.7	30.9 ± 1.1	51.3 ± 8.6
REMOVE	10.4 ± 0.6	47.8 ± 2.9	79.2 ± 3.6
RENAME	9.4 ± 1.4	21.5 ± 3.3	26.1 ± 2.3
STATFS	3.2 ± 0.1	3.2 ± 0.1	3.4 ± 0.2
NULL	2.5 ± 0.1	2.5 ± 0.1	2.5 ± 0.3
SETATTR	8.5 ± 0.3	89.8 ± 4.7	169.3 ± 4.7
RMDIR	7.3 ± 0.2	24.8 ± 1.6	37.7 ± 2.5

of the time, it is valuable to accomplish them in parallel. When disk access is eliminated, two replicas are 108% slower than one, indicating that it is roughly as expensive to send the `WRITE` request from one server to another as it is to deliver it initially with NFS.

Replication has a high relative cost in operations that require name service access. In particular, `SETATTR`, `MKDIR`, and `RMDIR` are pure name service operations, and replication has a dramatic effect on their time. The biggest jump occurs between one replica and two replicas. If there is only one replica, then the name service can shortcut past all of the ISIS communications protocols, but two replicas trigger the full communications overhead.

SETATTR is unusual in that replication increases the time required for disk access. The reason is that several name service updates are performed. In the single server case, the server can synchronously flush all updates with one file operation, but when there is more than one server, each update is written and synchronized separately to disk. This behavior is an artifact of a poorly tuned implementation. Fortunately, SETATTR is a very uncommon operation: essentially, it is only used when a UNIX `chmod` command is used. Note that going from two to three replicas does not increase the disk access cost component.

3.8.4 Variation in Client/Server Configuration

For this set of benchmarks, we focus on the cost of using different client and server configurations. In particular, we vary the communication infrastructure and the location of the file replicas. This type of analysis helps to determine the cost of small parts of file system operations.

The following test measures the cost of remote file access by using four different server configurations. In the first server configuration (1 local), there is only one server, and the client directly mounts it. In the second server configuration (1 remote), there are two servers with one replica of the file, and the replica is on the server that the client does not mount. In the third configuration (1 local + 1 remote), there are two servers with two replicas of the file, and the client mounts a server with a replica. In the last configuration (2 remote), there are three servers and two replicas, and the clients mounts a server without a replica. We tested all four configurations using the normal Deceit server and using the Deceit server without disk access. Table 3.4 lists the time of each operation for 8 Kb WRITEs.

The difference in time between "1 local" and "1 remote" is about 18 milliseconds for both server types, which is the time to forward a WRITE from one server to another. This time is consistent with the time required to execute an NFS WRITE

Table 3.4: Local vs. Remote WRITE Time (milliseconds)

Server	1 local	1 remote	1 local + 1 remote	2 remote
Normal	130.5 ± 4.2	147.6 ± 3.9	153.5 ± 3.6	184.6 ± 2.8
No disk	17.5 ± 0.4	36.3 ± 0.7	38.5 ± 0.5	46.5 ± 1.0

without disk access (i.e. the time required for an RPC with 8 Kb of data). The difference between "1 local + 1 remote" and "2 remote" is due to the time required to send the message containing the WRITE to the seconds remote server. In the "no disk" case, the difference is 8 milliseconds, which is less than 18 because there is some concurrency in the message transmission. However, in the "normal" case, the difference is 39 milliseconds, which is too large and an artifact of a poor implementation[17].

The write token produces a transient bottleneck when there are concurrent WRITEs to one file. The next test measures the cost of contention for the write token. In this test, there is one file with r replicas written by r clients. The important variable is whether all clients mount the same server or different servers. When all clients mount the same server, there is no token contention since the token can stay at that server. When all clients mount different servers, the token must move around rapidly among the servers. Table 3.5 shows the total number of WRITE operations per second (of all clients) for several different configurations.

It is difficult to interpret the results of Table 3.5 because of the many variables involved. It is not clear how frequently a token acquisition was required, nor how the client configuration effected load balancing. Regardless, it is clear that token contention has a strong effect on performance.

[17]The broadcast protocol did not produce adequate concurrency, and the first WRITE became partially serialized with the second WRITE.

Table 3.5: `WRITE`s per Second with Token Contention

Server	Replication	Without Contention	With Contention
Normal	2	11.03 ± 0.26	7.96 ± 0.16 (28% less)
Normal	3	9.91 ± 0.20	6.64 ± 0.15 (33% less)
No disk	2	43.01 ± 0.85	28.29 ± 0.45 (34% less)
No disk	3	36.45 ± 0.43	20.45 ± 0.74 (44% less)

3.8.5 Variation in File Parameter Settings

In this set of benchmarks, we focus more closely on the cost of using different file parameter settings. The cost of varying the number of replicas is demonstrated in Table 3.2. The other relevant file parameters are the write safety level, the use of file migration, and stability monitoring.

The next test measures the performance gain achieved by varying the write safety level. Table 3.6 shows the time for 8 Kb `WRITE` operations to a file with three replicas. Three different versions of Deceit were tested: the normal version, the version without disk access, and the version with a randomly chosen extra delay in each `write`.

Table 3.6: Effects of Write Safety Level

Server	$s = 0$	$s = 1$	$s = 2$	$s = 3$
Normal	46.6 ± 23.2	158.9 ± 3.3	171.7 ± 4.1	174.4 ± 4.1
No disk	36.9 ± 1.1	37.8 ± 1.6	46.1 ± 1.7	46.5 ± 1.4
Random Extra	42.3 ± 15.3	175.5 ± 2.6	192.3 ± 4.3	206.5 ± 3.6

The biggest jump is time is between $s = 0$ and $s = 1$ because $s = 0$ implies that no synchronous disk access is necessary. In the "normal" case, there is little difference after that: waiting for three disks is only 9.8% more expensive than waiting for one. In the "random extra" case, we randomly added up to 50 milliseconds of time to each disk operations to simulate background load from other clients. Yet there is still only 17.6% growth is time between $s = 1$ and $s = 3$. The conclusion is that s should just be equal to the number of replicas, unless a particular server is known to be very slow.

The value of file migration was demonstrated in Table 3.4. In that table, we showed that an 8 Kb `WRITE` required 18 more milliseconds when it needed to be transferred between servers. File migration will eliminate this extra time in most cases for typical file system usage. The penalty for file migration is the expense of an occasional file transfer. We measured the time required to transfer a file for two different file sizes[18]. For a 1 Kb file, the time was 128 $\pm$ 10 milliseconds. For a 100 Kb file, the time was 1931 $\pm$ 100 milliseconds. Using a linear interpolation between the two values, we find that the transfer rate is approximately 55 Kb per second plus 148 milliseconds of overhead. Considering the bandwidth of the network (1.25 Mb/sec) and the disk (3 Mb/sec), this transfer rate seems very slow. The slow rate indicates that the Deceit file transfer protocol needs to be debugged and tuned.

Stability monitoring is difficult to evaluate since it is only important under very unusual conditions: one client must read a file at a server where there is a replica immediately after another client writes the same file at a different server. Under those conditions, a `READ` operation will be forwarded between servers instead of using a local replica. The difference in time between a local `READ` and a remote `READ` was measured as 14.0 $\pm$ 1.7 milliseconds, which can be relatively significant under some conditions. In general however, the performance impact of stability monitoring is negligible.

[18]The file transfer protocol is triggered asynchronously after a `READ` operation has completed and a reply has been sent to the client. The file transfer protocol is executed in a separate thread by Deceit, and no client is blocked by it.

3.8.6 Resource Consumption

The previous performance tests focused on operation execution time. In this section, we emphasize the resource consumption of a `WRITE` operation. In the next test, between one and three clients issued concurrent writes to separate files on one server. Individually, each client executed `WRITE` operations serially: it waited for one to finish before starting the next one. Up to two other servers held replicas of those files. Table 3.7 shows the average time required for each `WRITE` operation.

Table 3.7: `WRITE` Time with Multiple Clients

Server	Number of Clients	Deceit(1)	Deceit(2)	Deceit(3)
Normal	1	134.0 $\pm$ 3.5	152.2 $\pm$ 4.4	195.5 $\pm$ 4.1
Normal	2	172.9 $\pm$ 2.6	179.2 $\pm$ 5.0	243.6 $\pm$ 11.4
Normal	3	241.7 $\pm$ 3.6	261.9 $\pm$ 5.3	316.8 $\pm$ 5.4
No disk	1	24.1 $\pm$ 0.5	43.1 $\pm$ 0.9	58.6 $\pm$ 1.2
No disk	2	27.2 $\pm$ 1.0	52.7 $\pm$ 2.0	77.7 $\pm$ 5.0
No disk	3	31.7 $\pm$ 1.7	66.3 $\pm$ 2.6	97.6 $\pm$ 4.9

This data suggests that since concurrency causes a the normal server to slow down more than a diskless server, disk access is the main concurrency bottleneck for a normal server. Table 3.8 shows the percentage reduction in disk performance for a normal server. These results were computed by subtracting the non-disk component from the normal server times and then computing a ratio.

3.9 Deceit Summary

3.9.1 General Conclusions

Deceit demonstrates that it is possible to allow users to dynamically control key aspects of the replication and control protocols. As a result, a wide range of per-

Table 3.8: Reduction in Disk Performance for `WRITE`

Number of Clients	Deceit(1)	Deceit(2)	Deceit(3)
2	35%	25%	35%
3	98%	101%	88%

formance, safety, and availability properties are made available. Also, "exotic" features such as file migration and disk load balancing are feasible. At the same time Deceit provides a clean and intuitive interface to clients. Deceit simplifies the administration of a collection of NFS servers by effectively unifying them into one server.

Deceit, however, is far from an ideal file system. The largest problem is that the server program is too complex. Thorough debugging is a nearly impossible task due to the distributed, multi-threaded, and highly non-deterministic nature of the server. In particular, race conditions continue to be a nagging problem. Powerful software tools such as a distributed debugger and automatic tester would help.

Conformance to the NFS protocol has crippled Deceit performance and functionality. NFS had seemed an obvious choice for a client/server protocol due to its simplicity and widespread availability. Unfortunately, it lacks client fail-over, consistent client caching, bulk data transfer, and explicit update synchronization. Also, some important details of the semantics are undocumented; they appear only in the primary implementation from Sun Microsystems. These failings limit the effectiveness of any filesystem supporting NFS and have forced some unpleasant design decisions on Deceit.

3.9.2 Performance Results

Section 3.8.2 demonstrated that using two or three replicas instead of one reduced server performance by 26% or 46% respectively. Some part of that figure can be attributed to poor server optimization, particularly in disk synchronization. Still, it is not clear if 46% is a "lot" or a "little": that decision is application dependent. Section 3.8.3 refined this result by breaking down server performance into individual operations. We found that the name service scales poorly, so certain operations (e.g. `MKDIR`) become very slow when replication is used. The name service design needs to be reconsidered to correct this problem.

Section 3.8.4 demonstrated that Deceit is relatively insensitive to the actual location of replicas. An extra communication hop only adds 13% to the time required to accomplish a `WRITE`. Considering the slow rate of file transfer (55 Kb/sec plus 148 msecs of overhead), file migration is not recommended in general. Even if file migration were very fast (1 Mb/sec), it would still be difficult to justify because of the extra disk traffic required.

If clients spray `WRITE` operations to more than one server for the same file at the same time, then token contention is a result and Deceit performs poorly. In Table 3.5, we demonstrated a performance degradation of 49% for three replicas under these conditions. The conclusion is that it is much less important which server clients use then ensuring clients use the same server for all access to a particular file.

In all of the benchmarks, disk access time dominated performance. When there was only one replica, fully 87% of the time require to execute a `WRITE` was spent in disk access. Even when three replicas were used (with corresponding higher communications overhead), disk access still consumed 69% of the time. These values are generally consistent with the other operations types. Table 3.7 (Section 3.8.6) showed that disk access is the primary bottleneck when there is concurrent server access. It is possible to saturate the disk drive with only two concurrent `WRITE` operations.

3.9.3 ISIS

ISIS greatly accelerated the implementation of Deceit. ISIS provides many primitives which are well suited to a distributed service. It was possible to explore many protocols and configurations in a relatively short period of time. On the other hand, ISIS performance and complexity are problematic. Some operations (e.g. group creation) are very expensive and, it was necessary to use awkward mechanisms to provide sufficient efficiency.

3.10 Future Deceit Enhancements

Many useful enhancements could be made to Deceit to make it a more useful system. A few possible enhancements are presented here. Other, more broadly defined, future work is presented at the end of this thesis.

3.10.1 Client Enhancement

The standard NFS client software does not effectively utilize Deceit since NFS can not switch between servers on a per-file basis. Unfortunately, any client enhancement would require a fundamental change to the NFS protocol. Assuming that an improved client/server protocol is available, a better client would have the following features. The client will use server call-backs to cache more effectively and efficiently. If a server fails, then the client would fail-over to another equivalent server[19]. Also, the client would switch between servers depending on replica locations to avoid server indirection.

[19]This can be implemented by using network impersonation without modifying client software.

The Andrew File System[SHN+85,HKM+88] and the Sprite File System[NWO88] are examples of file systems that have very carefully designed clients and client/server protocols. The Coda File System[SKK+89] and the Ficus File System[GHM+90, PGJH90] go further and have clients that can act as temporary servers. None of these client protocols are incompatible with the design of Deceit. It would be a straightforward (but time consuming) exercise to implement the necessary interfaces.

3.10.2 Cells

In the above discussion, it was assumed that all clients could directly access any Deceit server, but from a security perspective this property is not necessarily desirable. Sets of Deceit servers would be subdivided into *cells* to prevent Deceit from being non-secure (and inefficient) in a very large implementation. Each cell would be an independent instance of Deceit with distinct files and processes. Each cell would maintain its own name space, and replication will be contained within a cell. A cell would provide security and administrative boundaries.

Access between cells would be provided through a logical directory. called the *global root directory*. This scheme is similar to the ideas presented in [Neu89] and [TCW89]. It would be impossible to list, as it implicitly would contain the full host names of every accessible Deceit server. Instead, it would be used indirectly as a subdirectory of a normal directory. For example, if a user is in the Cornell computer science cell and wants to access files in the MIT computer science cell, he picks a machine "foo.cs.mit.edu" at MIT where a Deceit server is running. By executing the command "cd /priv/global/foo.cs.mit.edu," a user can access the MIT cell with normal file operations. In this case, the global root directory is "/priv/global." Mount and access restrictions would be applied to the Cornell cell as with any client.

3.10.3 Fast Local Storage

Deceit performance is badly encumbered by the fact that all local storage is done in normal UNIX files. The problem is that UNIX provides directories, time stamps, and hard link reference counts, which is redundant with service that Deceit implements. Creating a Deceit file will modify the segment meta-data file, the name service log file, the segment storage file, and the UNIX directory where the storage file is placed. Including i-node and data writes, up to eight disk writes can be required to create a Deceit file. Also, Deceit is not able to efficiently schedule disk access. In a future version, Deceit would directly access the disk device. For additional performance gains, a log-structured file system[RO90,OD89,DO89] would be used.

An alternative to rebuilding the base file system from the bottom up is to use stackable layers[GHM+90,Ros90] in the file server. Splitting a file system into several layers greatly increases the reusability of the code and can improve overall efficiency. Each layer is optimized for a specific function, and layers can be shuffled around to provide different file system properties.

3.10.4 New File Parameters

The existing file parameters are not ideal. Perhaps with different parameters, the design could be simplified, the range of behavior could be increased, and the efficiency could be improved. For example, one new parameter could be the *Read Quorum Size*. When a file is read, this number of replicas is read. If there are not enough replicas, then the operation is aborted. Otherwise, the result with the latest timestamp is used. A read quorum can be used to provide consistency and stability. In fact, with read quorums the entire stability monitoring protocol would become obsolete, although a `read` operation would have to be executed on several servers instead of one.

Another new parameter could be the binary parameter Immutable. If Immutable is *true*, then the file contents are fixed. If the file system knows that a file is immutable, then caching and replication become much simpler. This optimization was used in LOCUS[WPE+83] as the normal mode of operation. Most of the protocols in the segment service could be circumvented yielding better performance and availability.

Chapter 4

Comparative Analysis

4.1 Deceit Analysis

In order to complete the file system survey begun in Chapter 2, the Deceit file system presented in Chapter 3 will be analyzed. This analysis will be expressed using the file parameters presented in Section 3.5 on page 76. The following is a brief summary of those parameters:

- Minimum Replica Level (r) - the minimum number of valid replicas that must be maintained.
- Write Safety Level (s) - the number of replica servers that must reply to an update before a `write` returns to the client.
- Write Quorum Size (q) - the minimum number of available replicas required for updating a file.
- Maximum Replica Level (m) - the maximum number of replicas. This parameter is used in combination with the write quorum size to prevent multiple file versions.

The analysis will result in a time computation for each operation as a function of SDAs, ADAs, SNPs, and ANPs. This allows us to directly compare Deceit with the earlier file systems described in Chapter 2. However, such a comparison is simplistic and potentially misleading. We will illustrate this fact by computing an expected operation time from our model and comparing the results with our benchmark times.

4.1.1 Performance Comparison

For the performance comparison, assume that there exists n servers and one file f with exactly $r \leq n$ replicas. Let S be a file server.

Using the results from Table 3.2 on page 88, it is possible to compute the actual time required for a typical SNP or SDA by computing the time differences between operations. We will start by computing the time for a SNP. A `NULL` operation requires 2.5 milliseconds, and it consists of a simple exchange of short messages. Therefore, a SNP for a short message is $2.5/2 = 1.2$ milliseconds. A `READ` operation (without disk) requires 17.9 milliseconds, and it consists of a short messages and an 8 Kb message. Therefore, a SNP for an 8 Kb message is $17.9 - 1.2 = 16.7$ milliseconds. Interpolating the two SNP times give an approximate formula of $1.2 + 1.9l$ milliseconds for a SNP where l is the message length in Kb. When an SNP consists of a broadcast to multiple destinations, it is longer because ISIS uses a sequence of messages to individual destinations to emulate a broadcast. The difference between 2 and 3 servers for an 8 Kb `WRITE` is $138.7 - 138.3 = 10.4$ milliseconds. Subtracting the reply yields $10.4 - 1.2 = 9.2$ milliseconds per additional destination for an 8 Kb message.

Next, we will compute the time required for a SDA. An 8 Kb `WRITE` operation without disk requires 18.3 milliseconds, and with disk it requires 129.3 milliseconds. Therefore, a SDA for an 8 Kb write is $129.3 - 18.3 = 111$ milliseconds. A `RENAME` operation involves a name service update which has a small synchronous disk write. Therefore, a SDA for a small write is $53.5 - 9.4 = 44.1$ milliseconds. The two SDA times give an approximate formula of $44.1 + 8.4l$ milliseconds for a write where l is the data length.

We can also approximate the time for a read. A synchronous write requires that data blocks and the i-node block are written. For a small write, there is only one block of each type, so the time to write an isolated block is $44.1/2 = 22.0$ milliseconds (this value is consistent with the 15 msecs seek time for the disk). For 8 Kb of related data blocks, the time is $111 - 22 = 89$ msecs. Assuming that reading data off a disk is as expensive as writing data on a disk, we assert that a read requires 22 msecs for a single disk block and 89 msecs for 8 Kb of related blocks. The two times give an approximate formula of $22 + 8.4l$ msecs for a read where l is the data length.

Common Components

Three sub-operations are used repeatedly in Deceit. They are file name service update (Section 3.2.2), write-token acquisition (Section 3.3.2), and group join (Section 3.3.3). We will analyze them first as separate components, and then we will analyze entire file system operations in Section 4.1.2.

A normal name service update requires a synchronous broadcast with replies at all servers followed by another asynchronous broadcast. An entry to the name service log file is written at all sites upon receipt of both broadcasts. Therefore, a name service update requires 2 SNPs, 1 ANP, 1 SDA, and 1 ADA for a total of $3n - 3$ messages. If an operation requires multiple service updates to be performed, they can be combined into a single update by using a single `flush_name` call. For example, a `RENAME` operations requires two name service updates: the old hard link must be deleted, and the new hard link must be created. By combining both updates into a single message and log file update, the two updates are essentially as expensive as one.

Group join occurs whenever a server accesses a file which has no local replica and has not been accessed for 30 seconds. Let S be a server that needs to join file group G_f of segment f. To do so, server S sends a join request to the coordinator for f, and the coordinator forwards this request to the token holder. The token holder then broadcasts the new group membership to all members of G_f including S. Assuming that a join request is forwarded k times, this operation costs $k + 2$ SNPs for a total of $r + k + 1$ messages. It is very unusual to for k to be larger than 1 because the coordinator is a member of G_f and will have very accurate knowledge about the location of the token holder.

Write-token acquisition occurs whenever a server attempts to write to a file that was written previously by another server. Assuming stability monitoring, this requires one broadcast to request the token and one broadcast to pass the token. The total cost is 2 SNPs for a total of $2r - 2$ messages. The write-token request can be combined with the group join request as shown in Figure 4.1 (as mentioned in Section 3.3.3). The total cost of joining G_f and acquiring the write token is $k + 3$ SNPs for a total of $2r + k + 2$ messages. This is a savings of 1 SNP and $r - 1$ messages on performing the operations separately.

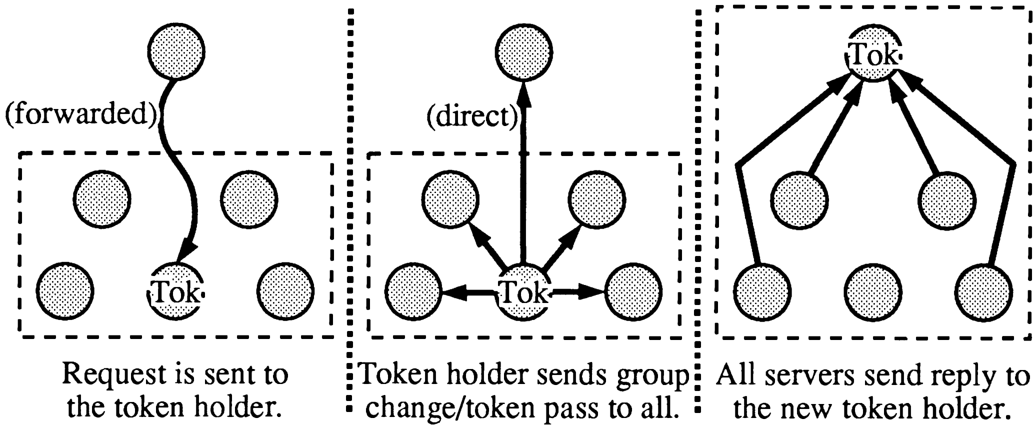

Figure 4.1: Combined Write-token Request and Group Join Request

If stability monitoring is not in use, then it is not necessary for all servers to reply to a token request. In this case, write-token acquisition still requires 2 SNPs but only for a total of r messages. When combined with the group join, the total cost is $k + 2$ SNPs for a total of $r + k + 2$ messages; a savings of 1 SNP.

4.1.2 Individual Operation Performance

In this section, we analyze the cost of each complete file system operation. Each operation can involve several stages, where some stages are optional. To clarify the analysis, the cost of each stage will presented in a table. The best, expected, and worst case cost for the whole operation will be given with some supplementary analysis. We will also compute the operation time from the cost model assuming that there are three servers with a replica at each server. This time will be compared with the experimental results (Section 3.8).

Read Performance

Table 4.1 shows the costs of each component in a `read` operation. The experimental time for each component in milliseconds is also provided.

Table 4.1: `Read` Operation Component Cost

Operation Component	SNPs	ANPs	SDAs	ADAs	messages	time
Client RPC	2	0	0	0	2	2.5–17.9
Group join	$k+2$	0	0	0	$r+k+1$	≥ 2.5
Relay to replica	2	0	0	0	2	2.5–17.9
Read from disk	0	0	1	0	0	22–89

In this table, "Client RPC" refers to the NFS RPC between the client and its preferred server S, which is always necessary. "Group join" refers to S joining G_f which is necessary if S lacks a replica of f and has not joined G_f within the last 30 seconds. "Relay to replica" refers to forwarding a `READ` request from S to another server S' that has a usable replica. As well as being necessary when S has no local replica, this component occurs when stability monitoring is in use, S is not the coordinator for the file, and an update has been sent within the last 10 seconds. Note that the reply to a forwarded `READ` request must go through S to reach the client. "Read from disk" refers to the actual disk transfer which is always necessary.

The worst `read` performance is achieved when S does not have a replica and has not accessed the file recently. Under these conditions, the cost is $k + 6$ SNPs, 1 SDA, and $r + k + 5$ messages. The best (and typical) `read` performance is achieved when S has a replica, and the file has not been updated recently. This case is typical because Deceit will create or cache a replica of the file on a server where it is being accessed, and client read caching tends to eliminate reads immediately after writes. In this case, the cost is 2 SNPs, 1 SDA, and 2 messages. File migration increases the probability that S will have a replica, at a cost of increased disk usage and background network traffic. If stability monitoring is disabled, then S can always read from its own replica, but one-copy serializability may be violated more frequently.

Table 3.2 provided a benchmark time of 19.1 milliseconds for a `READ` when $r = 3$. In this benchmark, there was always a local copy of the file, so only 2 SNPs and 1 SDA were used (i.e. the "typical" case). The first SNP is a small data request and should take 1.2 milliseconds. The second SNP is a large reply and should take 16.7 msecs. Finally, the SDA is a large data read and should take 89 msecs. The total is $1.2 + 16.7 + 89 = 106.9$ msecs. Since this number is much larger than 18 msecs, we can deduce the server caching (in the kernel) captured nearly every read request. Since the benchmark only required a few minutes, and the servers had a large amount of physical memory, it is reasonable that the server kept the entire working set of data in memory.

Write Performance

We will now consider the performance of the `write` operation. Table 4.2 shows the cost of the different operation components in a `write` operation. The experimental time in milliseconds for each operation component is provided. Here, we assume that there are no competing `write` operations from other clients (Section 3.8.4 demonstrated that such interference is costly). Let S be the server that receives the `write` request from a NFS client, and let f be the file that is being written with a write safety level of s.

Table 4.2: `Write` Operation Component Cost

Operation Component	SNPs	ANPs	SDAs	ADAs	messages	time
Client RPC	2	0	0	0	2	2.5–17.9
Group join	$k+2$	0	0	0	$r+k+1$	≥ 2.5
Token acquisition	2	0	0	0	$2r-2$	2.5
Write to disk ($s>0$)	0	0	1	0	0	44.1–111
Write to disk ($s=0$)	0	0	0	1	0	0
(Assuming S has no replica of f)						
Update Delivery ($s>0$)	2	0	0	0	$2r$	2.5–17.9
Update Delivery ($s=0$)	0	2	0	0	$2r$	0
(Assuming S has a replica of f and $r>1$)						
Update Delivery ($s>0$)	2	0	0	0	$2r-2$	2.5–17.9
Update Delivery ($s=0$)	0	2	0	0	$2r-2$	0

The worst `write` performance is achieved if S does not have a replica of f, S has not accessed f in a long time, $s \geq r$, and stability monitoring is specified. Under these conditions the total cost is $k+7$ SNPs, 1 SDA, and $4r+k+4$ messages (recall that group join and token acquisition can be combined.). The worst case cost is $k+6$ SNPs, 1 SDA, and $3r+k+4$ messages if stability monitoring is not specified. The typical performance occurs if S has a replica of f (due to file migration), S has written to f recently (since files are usually entirely rewritten), and $s=r$ (for fully synchronous writes). The total cost is 4 SNPs, 1 SDA, and $2r$ messages. The best performance occurs if S has the only replica of f, S has written to f recently, and $s=0$. in which case the total cost is 2 SNPs, 1 ADA, and 2 messages. `Write` performance can be optimized by setting $r=1$, by using file migration, and by disabling stability monitoring.

Deceit with $r=3$ required 162.0 ± 3.0 milliseconds for a `WRITE` operation in the "typical" case. The model predicts a similar time. Of the 4 SNPs, 3 are small messages requiring 1.2 msecs each, and 1 is a large message requiring 16.7 msecs. The SDA is a large write and required 111 msecs. Therefore, the model predicts a total time of $(3 \times 1.2) + 16.7 + 111 = 131.3$ msecs. ISIS uses a sequence of messages to emulate a broadcast, and each additional message requires 9.2 msecs. Given that extra cost, the expected time is $131.3 + (2 \times 9.2) = 149.7$ msecs. We do not understand where the remaining $162.0 - 149.7 = 12.3$ msecs is spent.

Write performance can also be optimized by setting s to 0, but doing so is dangerous since most errors will not be reported to the user. We evaluated Deceit performance when $s=0$ in Table 3.6, Section 3.8.5. The NFS protocol does not allow truly asynchronous operation, so setting $s=0$ in Deceit does not eliminate the 2 SNPs from the operation cost. Also, in the benchmark Deceit became overwhelmed with requests, so the server was not able to respond to new requests rapidly. The result was that a `write` to three replicas required 46.6 milliseconds (rather than the 18 milliseconds required for pure communication).

Create Performance

Table 4.3 shows the cost of different operation components in a `create` operation. The experimental time in milliseconds for each operation component is also shown.

Table 4.3: `Create` Operation Component Cost

Operation Component	SNPs	ANPs	SDAs	ADAs	messages	time
Client RPC	2	0	0	0	2	2.5
Name service update	2	1	1	1	$3n-3$	51–165.6
Replica creation	2	0	1	0	$2r-2$	70.7

"Replica creation" refers to the broadcast required to create $r-1$ blank replicas on servers other than the client's preferred server. This component does not require communication if $r=1$. The total cost of a `create` is 6 SNPs (4 if $r=1$), 1 ANP, 2 SDAs, 1 ADA, and $2r+3n$ messages.

Deceit with three servers required 226.7 ± 7.6 milliseconds for a `CREATE` operation, and $226.7 - 51.7 = 175$ msecs were spent on disk access. The predicted time is much lower. The 4 SNPs corresponded to 6 small messages and should require $(6 \times 1.2) = 7.2$ msecs. One SDA is a small write and should require 44.1 msecs. The other SDA is a file create and should require 68.2 milliseconds. Therefore, the expected time for a `CREATE` on three servers is $7.2 + 44.1 + 68.2 = 119.5$ msecs.

The discrepancy between 226.7 and 119.5 was due to several reasons. First, the SDA required three synchronous disk accesses instead of one due to a performance bug in the name service. Second, the name server broadcast was emulated by using three messages to individual destinations instead of a real broadcast, although one of those messages required trivial delivery. With these two considerations, the expected time rises to $(6 \times 2 \times 1.2) + (3 \times 44.1) + 68.2 = 214.9$ msecs. This value is close to 226.7, and the difference can be justified by CPU usage.

Delete Performance

Deceit implements the `delete` operation by deleting the directory entry in the name service. A replica at server S' is deleted when S' receives the deletion of the name service entry for the last hard link to f. Table 4.4 shows the cost of each operation component. The experimental time in milliseconds for each component is also shown.

Table 4.4: `Delete` Operation Component Cost

Operation Component	SNPs	ANPs	SDAs	ADAs	messages	time
Client RPC	2	0	0	0	2	2.5
Name service update	2	1	1	1	$3n-3$	51–165.6
Replica deletion	0	0	1	0	$2r-2$	75.4

The total cost is 4 SNPs, 1 ANP, 2 SDAs, 1 ADA, and $3n-1$ messages. It is very similar to the `create` operation.

Deceit with three servers required 532.6 ± 17.7 msecs for a `REMOVE` operation, and $532.6 - 79.2 = 453.4$ msecs were spent on disk access. The 4 SNPs corresponded to 4 small messages and should require $4 \times 1.2 = 4.8$ msecs. One SDA is a small write and should require 44.1 msecs. The other SDA is a file remove and should require 75.4 milliseconds. Therefore, the expected time for a `REMOVE` on three servers is $4.8 + 44.1 + 75.4 = 124.3$ msecs. This value is low for the same reasons that the `CREATE` operation prediction was low, but 532.6 is still very high even with this consideration. Clearly, a `REMOVE` operation must include a performance bug that has not been detected yet.

Creatdir and Deletedir Performance

These operations are accomplished with a client RPC and a single name service update. The cost is 4 SNPs, 1 ANP, 1 SDA, 1 ADA, and $3n - 1$ messages.

Deceit with three servers required 278.0 $\pm$ 17.4 msecs for a `MKDIR` and 218.4 $\pm$ 10.8 msecs for a `RMDIR`. `MKDIR` was slower than `RMDIR` because of the associated memory allocation and sanity checking. Also, the `RMDIR` code was slightly better at managing updates to the name service.

The operation time predicted by the model for `RMDIR` was smaller than the observed time. The 4 SNPs corresponded to 4 small messages and should require $4 \times 1.2 = 4.8$ msecs. The SDA is a small write and should take 44.1 msecs. Therefore, the expected time for a `RMDIR` on three servers is $4.8 + 44.1 = 48.9$ msecs. The discrepancy between 218.4 and 48.8 was the same as the discrepancy in the `CREATE` operation. Using the same correction, the expected time rises to $(4 \times 2 \times 1.2) + (3 \times 44.1) = 146.7$. We must conclude that the `RMDIR` operation must suffer from the same performance bug that `REMOVE` does since there is still a significant difference between 146.7 and 218.4. This conclusion is sensible since both operations use nearly the same code.

Readdir and Open Performance

It is difficult to analyze the `open` operation on Deceit since NFS does not directly implement `open`. In the SunOS implementation of `open`, an `open` is usually translated to a NFS `LOOKUP` RPC, and we will assume this case. A `LOOKUP` and `READDIR` RPC require a name service lookup and no interserver communication. The name service fully replicates all data and keeps it in memory. The cost for either operation is 2 SNPs and 2 messages.

The `LOOKUP` operation required 4.3 $\pm$ 0.3 msecs in Deceit. Both SNPs in a `LOOKUP` are short messages and should require 1.2 msecs each. Therefore, the model predicts the time for `LOOKUP` as $(2 \times 1.2) = 2.4$ msecs. The remaining $4.3 - 2.4 = 1.9$ msecs can be explained as the time required for the server to search the name service data structure. The `READDIR` operations required 12.3 ± 1.3 msecs in Deceit. The first SNP was a short request messages. The second SNP was a reply containing the directory contents, and its length was variable. The model predicts the minimum total as $1.2 + 1.2 = 2.4$ msecs and the maximum total as $1.2 + 16.7 = 18.9$ msecs. The actual time falls within this range and indicates a reply size of 4.2 Kb.

4.1.3 Availability Comparison

Deceit has maximum read availability. This availability is summarized in Table 4.5, which is a duplicate of Table 1.2 on page 9. The biggest jump in availability is achieved by increasing n from 1 to 2. The amount of time that a server is not available is reduced by a factor of 21 and reliability is increased by a factor of 11. As n tends to infinity, availability reaches an asymptote due to network failures: distant servers produce little increase in availability since they must communicate to the client through an unreliable network.

Write availability is governed by the write quorum size (q) and the minimum number of replicas (r). Computed availability and reliability are shown in Tables 4.6 and 4.7 for different values of r and q. The analysis methods are described in [BP75]. The failure times from Table 1.1 on page 3 are used. The value corresponding to $q > \frac{r}{2}$ (providing data consistency) are shown in boldface.

As expected, q can have a profound impact on availability and reliability. Using $q = 2$ instead of $q = 1$ increases the time that the server is not available by a factor of 23 to 54. Using $q = 2$ also decreases reliability by a factor of 18 to 24. In order to prevent multiple file versions, it is necessary that $q > \frac{r}{2}$. For $r > 2$, $q > \frac{r}{2}$ increases the time that the server is not available by a factor of 54 to 68. Reliability is decreased by a factor of 35 to 51. On the other hand, using $q > \frac{r}{2}$ for $r \geq 3$ is

Table 4.5: Deceit Read Availability

r	Availability	Reliability
1	95.42%	393.5 hours (16 days)
2	99.786%	4,335 hours (6 months)
3	99.986%	41,508 hours (4.7 years)
4	99.9956%	101,895 hours (11.6 years)
5	99.9960%	111,873 hours (12.8 years)
6	99.9960%	112,505 hours (12.8 years)
7	99.9960%	112,540 hours (12.8 years)
8	99.9960%	112,542 hours (12.8 years)

more reliable that $r = 1$.

4.1.4 Safety Comparison

The safety properties exhibited by a Deceit file are dependent on the file parameter settings. They are summarized in the following list:

- client cache consistency = *timeout*

  Client cache data is maintained with a 30 second timeout according to the NFS protocol.

- if $q > \frac{m}{2}$, then data consistency = *eventual*
  else data consistency = *non-directory*
  If the quorum level is greater then half of the maximum number of replicas, then one file version will always dominate after a partition or crash recovers. Otherwise, the user is responsible for reconciling the contents of normal files. The name service automatically reconciles directories.

Table 4.6: Deceit Write Availability (Percentage)

r	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$
1	**95.42**	-	-	-	-	-
2	99.786	**90.97**	-	-	-	-
3	99.986	**99.299**	86.72	-	-	-
4	99.9956	99.872	**98.641**	82.68	-	-
5	99.9960	99.907	**99.732**	97.83	78.82	-
6	99.9960	99.909	99.815	**99.562**	96.88	75.15

Table 4.7: Deceit Write Reliability (Hours)

r	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$
1	**393.5**	-	-	-	-	-
2	4,335	**191.5**	-	-	-	-
3	41,508	**1,259**	126.6	-	-	-
4	101,895	4,657	**654.5**	94.51	-	-
5	111,873	6,207	**2,257**	414.5	75.41	-
6	112,505	6,371	3,153	**1,395**	291.5	62.73

- if stability monitoring is used, then one-copy serializability = *true*
  else one-copy serializability = *false*
  Stability monitoring insures that only one replica is visible while there are active updates.
- update stability = *false*
  If all replica servers fail, and an obsolete replica recovers, then updates will disappear. However, if the write quorum is equal to the maximum number of

replicas, then it is very unlikely that any replica will become obsolete.

- atomicity = *true*

  Deceit is careful to execute operations in a resilient way. Recall that the name service logs updates atomically.

4.2 Formal Model

The remainder of the chapter consists of a discussion about file system safety and performance. Our goal is to understand the efficiency of Deceit and the utility of flexible semantics. This discussion is motivated by the survey in Chapter 2 and by experience and experimentation with Deceit (Chapter 3). As part of this effort, we try to uncover unspecified safety properties about file systems. File system designers usually have a very intuitive understanding about the properties of a "correct" file system and frequently make choices concerning application compatibility, client interface simplicity, and file system performance. In this chapter, we present several properties that by observation seem to be paramount to file system designers.

Unfortunately, the file system literature contains very little material on the formal properties of each file system and its environment. This fact is particularly true for large, complex file systems, and it reflects the sharp split between theory and practice in Compute Science. Therefore, it is difficult to verify these results without extensive (and occasionally heated) discussion with the primary authors of each file system.

Assume that there is an oracular service called the *failure detector* that notifies all machines of every process crash and recovery[SS83,Sch84]. Notifications can be delayed, but all machines are eventually notified, if they do not fail first. Failure is immediately detected during synchronous communication: if A sends a message to B and then waits for a reply, and B had previously crashed, then the reply will return immediately with the information that B has failed.

Communication is reliable and FIFO per channel. Partitions are not allowed, however, *virtual partitions*[ASC85] are possible, as discussed in Section 3.1.1. In a virtual partition, two disjoint sets of machines are alternately alive; there is no time in which the two sets are both alive and can communicate. For example, server A runs for a few minutes after server B has crashed, then A crashes, then B recovers and runs while A is down. A virtual partition is similar to a real partition since two hosts can not communicate, yet they must independently preserve file system semantics. A virtual partition recovers when both servers are running simultaneously and can effectively communicate with each other.

Some of the safety properties in this chapter use the term *available*. For example, "a client can read a file if any replica is *available*." Our definition of available is that A is available to B if A can perform synchronous communication with B (i.e. request and reply). For example, if client C wishes to send a file update u to server S, then S is available to C if S can receive u, process u, and then reply to C. C learns that S was available when C receives a reply from S without being notified that S had failed. According to this definition, it is not possible to know which machines are available for an operation until after it has completed.

Some of the results in this chapter are lower bounds of operation cost. Since the focus of this thesis is replication and fault tolerance, we will restrict our attention to file systems that maintain at least three replicas of every file with each replica on a different server (it will also be shown that maintaining two replicas is disportionately faster than maintaining three replicas). The following 2 properties specifies what we consider to be minimally correct behavior for such a system.

1. If all replicas are identical and available to a client C, then any updates issued by C will be applied to all replicas.

2. A final reply must be sent to the client after an update has been written to the disk for every available replica (i.e. the file system is fully synchronous).

Property 2 implies that an update requires at least 2 SNPs and 1 SDA: the update request, the disk write, and the update reply.

4.3 Maximum Availability

Maximum availability is provided by Locus, Coda, Ficus, and Deceit (with certain file parameter settings). Table 4.8 shows the safety properties for these four systems.

Table 4.8: Maximum Availability Safety Properties

	Locus	Coda	Ficus	Deceit
client cache consistency	*true*	*on close*	*timeout*	*timeout*
data consistency	*user*	*non-directory*	*non-directory*	*non-directory*
one-copy serializability	*true*	*true*	*true*	*true*
update stability	*false*	*false*	*false*	*false*

The important observation is that none of the systems provide data consistency or update stability. In fact, it can be shown that these properties cannot be provided when maximum availability is provided.

We first show that maximum availability precludes update stability. Let two servers S_1 and S_2 each have a replica of a file f. Consider the following sequence of events:

1. S_2 crashes.
2. A client C issues a `write` for file f. Since the servers provide maximum write availability, the update will be applied to the replica at S_1.
3. S_1 crashes, and then S_2 recovers.
4. C issues a `read` for f. Due to maximum read availability, S_2 will respond to the request.

The result of the `read` will correspond to the contents of f before the `write` was applied. In effect, the `write` was lost, and update stability is violated. S_1 was never able to communicate with S_2 since there was not a time when both were up. Note that one-copy serializability implies update stability, and so maximum availability also precludes one-copy serializability.

Since update stability (or 1CS) can be more important than maximum availability for some applications, it is useful to provide this kind of flexibility. For example, if servers always recover using the last process to fail[Ske85], then update stability can be provided. Another solution would be to require that a majority of the servers respond to every operation.

4.3.1 Data Consistency

None of the maximum availability file systems provide data consistency for reasons more subtle than those for update stability. Consider the following protocols that *do* supply data consistency.

Protocol 1 (First Data Consistency Protocol) *Let a file f be replicated at servers S_1, S_2, ... S_n. All available servers apply all updates, and periodic state exchange is used to insure that all updates are applied to all replicas eventually despite failure. Version vectors are used to check for inconsistency, and if inconsistency is detected, then one version is chosen to replace all others.*

This protocol has a serious problem. Assume that f has replicas at S_1 and S_2, and a partition (either real or virtual) separates S_1 and S_2. Let update u_1 be issued to S_1 yielding file f^{u_1}, and update u_2 be issued to S_2 yielding file f^{u_2}. This scenarios is illustrated in Figure 4.2. When the partition disappears, is it correct to apply u_1 to f^{u_2} or to apply u_2 to f^{u_1}? Which result should be chosen as the final one? Update u_1 was produced using f, not f^{u_2}, and it may not be correct to apply it to f^{u_2}. This condition can be expressed more formally as a safety property.

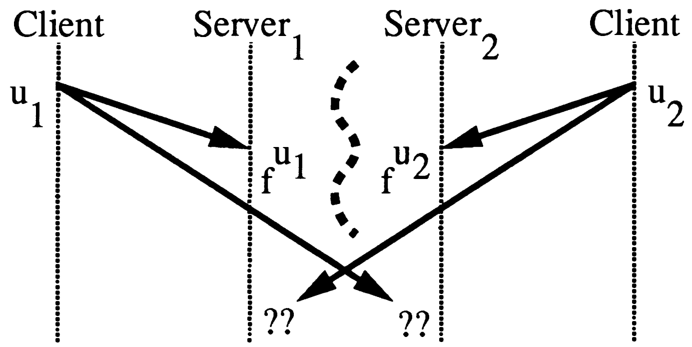

Figure 4.2: Data Consistency Scenario

Definition: Let $\prec$ be a partial order on some finite set of elements. y is an *immediate predecessor* of x iff $y \prec x$ and there does not exist z such that $y \prec z \prec x$. The relation $\prec$ is a *tree order* iff every element except $\perp$ has a unique immediate predecessor. A *path* is a list of elements $x_0, x_1, \ldots, x_n$ such that $\perp$ is the immediate predecessor of x_0, and x_i is the immediate predecessor of $x_i + 1$. It can be shown that for every element x in a tree order, there is a unique path starting at $\perp$ and ending in x. Also, every y such that $y \prec x$ is on the path ending in x.

Definition: Let f be a replicated file in a file system. The file system satisfies *update monotonicity* if a tree order can be formed on the set of `write` operations to each file. This order must respect the order in which clients issue `writes`. Let "$\prec$" be this order relation. The result of each `read` must be consistent with some path in the tree of `writes`.

The definition of update monotonicity is motivated by the need for a file system to be as close to one-copy serializable as possible while still having maximum availability. One-copy serializability is considered to be an important property in most data systems, and it can not be ignored. Update monotonicity is a weakening of 1CS that can be provided simultaneously with maximum availability. 1CS implies update monotonicity because 1CS forces a linear order which is trivially a tree order. The intuition is that update monotonicity allows a file to diverge into multiple versions, but each version respects 1CS. Since the file system can allow replicas to diverge is a controlled way, updates can still continue during a partition.

Returning to the scenario from Figure 4.2, applying u_1 to f^{u_2} or applying u_2 to f^{u_1} is a violation of update monotonicity. Assume that clients read f^{u_1} and f^{u_2} before recovery from the partition. If $u_1 \prec u_2$, then the act of reading f^{u_2} violates update monotonicity. Similarly, if $u_2 \prec u_1$, then the act of reading f^{u_1} violates update monotonicity. Finally, the act of applying either update on top of the other forces one of the orders. To summarize, the first data consistency protocol violates update monotonicity regardless of the strategy for converging on a single version. Here is another protocol that provides data consistency.

Protocol 2 (Second Data Consistency Protocol) *Let a file f be replicated at servers S_1, S_2, ... S_n. All updates are applied to all available replicas, but updates are not stored and applied later to recovering replicas. If a file version conflict arises during recovery from a crash or partition, then one particular version is chosen and all other versions are discarded.*

The problem with this protocol is more obvious. For the explanation the following definition is necessary:

Definition: A file system satisfies *update persistence* if an update that is accepted by at least one server is eventually visible to all clients.

Update persistence captures a fundamental property of file systems: data is not discarded without authorization from a client. Every file system mentioned in this paper satisfies update persistence. The file system community seems to have come to the consensus that update persistence is more valuable that data consistency. The second data consistency protocol violates update persistence by discarding versions, and the updates that produced them, before every client can access them.

Finally, we can show that the combination of maximum availability, update monotonicity, and update persistence preclude data consistency. Assume that f has replicas at S_1 and S_2, and a partition separates S_1 and S_2. Let update u_1 be issued to S_1 and update u_2 be issued to S_2. Due to maximum availability, u_1 will be applied at S_1 yielding file f^{u_1}, and u_2 will be applied at S_2 yielding file f^{u_2}. This scenario is illustrated in Figure 4.2. From the above argument, u_1 can not be applied to f^{u_2}, and u_2 can not be applied to f^{u_1} without violating update monotonicity. Further, both f^{u_1} and f^{u_2} must be eventually available to the clients to satisfy update persistence. Therefore, data consistency is violated.

Several methods are available for providing data consistency at some cost in availability. Deceit can be made to provide data consistency by setting the write quorum size to be more than half of the maximum replica level.

4.3.2 Safety Gaps Revisited

In the context of the previous section, the safety property gaps in Table 4.8 are reasonable. However, the values for data consistency are not *false*, but instead are *user* and *non-directory*. First, consider the value of *user*. Users reconcile file versions by applying updates to some versions and deleting others until a single version is left. Since the updates are freshly produced and applied to specific versions, update monotonicity is not violated by the process of manual reconciliation. Additionally, the user specifically requests the version deletions, after presumably examining all versions, and so update persistence is not violated. By forcing the user to explicitly request unsafe operations, the core safety properties are satisfied.

Using automatic directory reconciliation (ADR) is a more questionable design choice. ADR is used because concurrent updates to a directory are far more common than concurrent updates to a file, and so most inconsistency is expected to occur in directories. Also the semantics of directories are well understood, making an

ADR protocol feasible. Unfortunately, some applications depend on strict one-copy serializability in directories, and it is inevitable that an ADR implementation is "not quite right." An example follows.

Consider a program that uses two files to store data. Assume that this program must satisfy the safety condition that files `a` and `b` are mutually exclusive; if one exists, then the other must not exist. In order to insure this, the program executes the following algorithm whenever it wants to create `a` or `b`:

1. Create file `x`. If this fails, try again.
2. If file `a` or `b` already exist, then abort.
3. Create `a` or `b`.
4. Read the directory to verify the creation of `a` or `b`.
5. Delete `x`.

File `x` is used to provide mutual exclusion in the critical section. In UNIX file systems, file creation can be used as a mutex since a file can not be created on top of another file of the same name.

Now consider a scenario where client c_1 tries to create `a`, and c_2 tries to create `b`, but they are in different network partitions. Assuming maximum availability, both clients will be able to create `x` and hence both enter the critical section. Afterwords, `a` will be created in one partition, and `b` will be created in the other.

The Coda, Ficus, and Deceit ADR protocols will resolve the above situation similarly. The creations and deletions of `x` do not conflict since they referred to two different instances of `x`. The creations of `a` and `b` do not conflict since the files have different names. Hence, the reconciled directory will contain `a` *and* `b`.

4.4 Performance Bounds

In this section, we analyze the cost of `read` and `write` operations in order to understand the efficiency of these operations in Deceit.

4.4.1 `Write` Lower Bounds

Consider the comparison among several file servers in Table 4.9.

Table 4.9: Message Cost Comparison

File System	Worst Case `Write` SNPs
Locus	4
Ficus	> 4 (amortized from `close`)
Deceit	> 4
Coda	2 (amortized from `close`)

While this is a small data sample, is does appear that Coda has an unusually efficient `write` operation. Coda pays for this performance with a loss of update monotonicity. This trade-off will be demonstrated below.

First, consider the case of a file server S that is not permitted to communicate with other servers, and let S receive update u. Since there is no communication, S can not determine a global ordering for u or the status of other replicas. In Section 4.2 we argued that S must apply u under these conditions, otherwise S would be degenerate. If there is a collection of non-communicating servers, then each must independently apply its updates in the order that they are received.

Now, we will show that update monotonicity implies that more than 2 SNPs are necessary in the worst case. Assume that all `writes` use only 2 SNPs in all cases. Also assume that all servers are initially up, and all replicas are identical. By the assumptions in Section 4.2, an update must be delivered to all servers. The update delivery and reply consume both SNPs for each update, and server will not be able to communicate among themselves. Consider the following scenario:

1. Client C_1 issues update u_1, and client C_2 issues update u_2 for file f.
2. Server S_1 receives u_1, and server S_2 receives u_2. S_1 and S_2 must apply these updates by the above argument.
3. Server S_1 receives u_2, and server S_2 receives u_1. Now S_1 and S_2 have applied u_1 and u_2 in different orders.
4. Client C_1 issues update u_3, and client C_2 issues update u_4.
5. Server S_1 receives u_3, and server S_2 receives u_4.
6. All servers crash and recover. Updates u_3 and u_4 were not delivered to any other server.

By update persistence, u_3 and u_4 must become visible to all clients. Exposing u_3 implies that $u_1 \prec u_2$. Exposing u_4 implies that $u_2 \prec u_1$. Therefore, update monotonicity is violated. Note that it is only necessary for a `write` to require more than 2 SNPs in the worst case.

The Coda file system does indeed violate update monotonicity: it is possible to apply the same set of updates in a different order at different servers. Coda detects this condition, and handles it with the same mechanism that it uses for normal version splitting. A result is that Coda can execute a `write` with 2 SNPs in all cases. On the other handle, the user will be asked to resolve inconsistency more frequently.

It is possible to optimize a file system so that only 2 SNPs are necessary in most cases. Bursts of operations are typical, so we would like to optimize for bursts from a single client. This type of efficiency is expressed in the following definition:

Definition: An *update stream* is a sequences of file updates from a single client for a single file that does not contend with concurrent updates from other clients for the same file. A file system is *stream efficient* if all but the first update in an update stream require only 2 SNPs.

Given this definition, we can derive the following conclusion: if a stream efficient file system uses 3 SNPs in the worst case for a `write` operation, then $O(r^2)$ messages are required. To see this fact, consider the following scenario: client C has been streaming updates to a file system for a long time. At some point C ceases updating, and another client C' issues an update u. By the definition of stream efficiency, servers must apply updates from C and reply to C immediately after receiving an update. Since message delivery can be delayed, servers can become arbitrarily inconsistent during a stream. Consider a server S that has received update u. Before S can apply u, S must determine a consistent order for u relative to updates from C. The constraint is that update u must be after all of the updates that have been completed at any server, since updates can not be retroactively removed. Therefore, S needs knowledge from all servers before S can apply u. This knowledge exchange requires 1 SNP and $r^2 - 2$ messages. If the client request and reply is added, then the total cost is 3 SNPs and $O(r^2)$ messages.

If it possible to reduce the number of messages by allowing 4 SNPs in the worst case. The knowledge exchange can be accomplished by sending all information to a single server, then broadcasting that information to all servers. In this case, 2 SNPs and $2r - 2$ messages are used for this exchange. The total cost is 4 SNPs and $O(r)$ messages.

In many file systems, including Deceit, the client can communicate with only one server. This restriction increases the cost of a `write` by 2 SNPs because the primary server must disperse the request and collect the replies on behalf of the client. Figure 4.3 illustrates the optimal communication pattern for a worst case `write` which is stream efficient, satisfies update monotonicity, and is sent to only one server. It can be shown by using a simple knowledge based argument that

Figure 4.3 is minimal. A total 5 SNPs and $O(r^2)$ messages, or 6 SNPs and $O(r)$ messages are required.

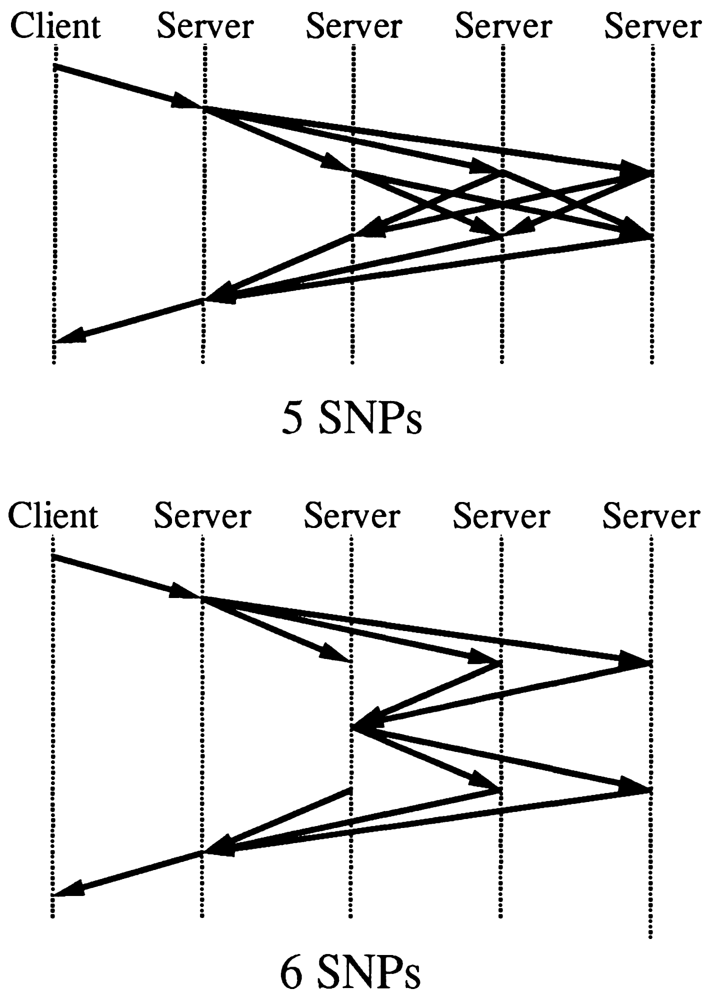

Figure 4.3: Optimal `Write` Protocols

In the special case of only two replicas, it is possible to use fewer SNPs because knowledge does not have to travel as far. Let C send u to S_1, and let S_1 and S_2 be the servers that store a replica of the file. After S_1 forwards u to S_2, S_2 can independently determine the global order for u. In this way, exactly 4 SNPs and 4 messages are used.

4.4.2 Write Upper Bounds

Section 4.4.1 provides lower bounds for `write` cost. This section provides complementary upper bounds. To start, Protocol 3 demonstrates that it is sufficient to use 3 SNPs and $2r$ messages for a `write` in all cases.

Protocol 3 . *Let client C issue update u.*

1. *C sends u to a previously designated server S.*
2. *S broadcasts u to all servers. S provides a global ordering an all updates.*
3. *All servers apply u when it is received from S.*
4. *All servers reply to C after u has been applied.*

To summarize, Protocol 3 showed that 3 SNPs are sufficient in all cases, but Section 4.4.1 showed that 2 SNPs are not sufficient for the worst case. There remains a range of protocols that use 2 SNPs for some operations and 3 SNPs for others. Protocol 6 is an example of such a protocol, and it is more efficient than Protocol 3. Protocol 6 is difficult to describe, so Protocols 4 and 5 are given as intermediate steps. In the process of developing Protocol 6, we prove that it is correct (a fact that may not be obvious at first glance).

Protocol 4 is the first step in developing Protocol 6. It is similar to an ABCAST developed by Dale Skeen[BJ87b]. To understand Protocol 4, the following definition is necessary:

Definition: Let client C_u broadcast update u, and client C_v broadcast v. These updates are *concurrent* if u is received before v at some servers, and u is received after v at others. We will write concurrent updates u and v as $u \| v$. If u and v are not concurrent, then there is a well defined order between them: the reception order at all servers. If u was received before v at all servers, then we will write $u \leadsto v$.

Protocol 4 *Let a client C generate an update u. Consider the following protocol:*

1. *C broadcasts u to all servers (FIFO ordering).*
2. *Immediately after receiving u, each server broadcasts u again to all servers (a descriptor of u can be used instead of u itself).*
3. *Each server waits for all secondary broadcasts of u.*
4. *Each server applies u in a consistent order.*
5. *Each server replies to C.*

The global order in step 4 remains to be described. This ordering is as follows:

1. *If $u \rightsquigarrow v$, then u is applied before v.*
2. *If $u \| v$, then any deterministic ordering between u and v will be correct. For example, the client addresses can be compared.*

Our next Lemma shows that rule 2 in step 4 produces a consistent order.

Lemma 1 *In Protocol 4, if server S has progressed to step 4, then S will know about all updates that are concurrent with u.*

Proof: Let v be an update, and $u \| v$. By the definition of concurrency, v will be received from the client at some server S' before u. Step 2 will be executed at S' for v before it is executed for u. Since message delivery preserves sender order, S will receive v from S' before S receives u from S'. Since S will wait in step 3 for all secondary broadcasts of u, S will receive v before progressing to step 4. □

Protocol 4 uses 3 SNPs and r^2+2r messages, so it is less efficient than Protocol 3, but the flexibility in the second ordering rule will be very useful for Protocol 6.

Protocol 5 *We now present a modification of Protocol 4 that reduces contention. A local variable* `last_client` *is initialized to* `null`.

1. *C broadcasts u to all servers.*
2. *At each server, if* `last_client` *equals C, then mark u with* `fast_flag`*; else set* `last_client` *to* `null`.

3. *Each server broadcasts u again.*
4. *Each server waits for all secondary broadcasts of u.*
5. *Each server applies u in a consistent order.*
6. *At each server, if there is no pending update from a client other than C, then* `last_client` *is set to C.*
7. *Each server replies to C.*

The ordering is step 5 is defined as follows:

1. *If* $u \rightsquigarrow v$*, then u is before v.*
2. *If* $u \| v$*, u was marked with* `fast_flag` *by any server, and v was not marked, then u is before v.*
3. *If the previous two rules do not apply, than any deterministic ordering can be used.*

Our next Lemma shows that rule 2 in step 5 produces a consistent order.

Lemma 2 *In Protocol 5, let* $u \| v$ *be concurrent updates, and let u be marked with* `fast_flag` *at* S'*. A server S will know that u was marked with* `fast_flag` *before S reaches step 5 for v.*

Proof: Since $u \| v$ are concurrent, they must be from different clients. First, assume that v was received before u at S'. In step 2, `last_client` would be set to `null` when v was received at S'. Since the condition in step 6 would not be satisfied until after v was applied, `last_client` would not change until after u was received at S'. It would be impossible for u to be marked with `fast_flag`. Therefore, v was received after u at S'.

Message delivery preserves sender order. Therefore, v will be received after u at S from S' in step 4. Since u was marked with `fast_flag` at S', then S will know it before S can finish step 4 for v. □

This protocol has another important property:

Lemma 3 *In Protocol 5, if two updates are concurrent, then at most one update can be marked with* `fast_flag`.

Proof: (by contradiction) Let u_2 and v_2 be concurrent updates. Assume that both updates were marked with `fast_flag` at servers S_u and S_v respectively in step 2. There must have been a previous update u_1 such that u_1 was issued from the same client as u_2, and u_1 caused `fast_flag` to be set in step 6. A similar relation holds between v_2 and a previous update v_1. Assume without loss of generality that u_1 was applied before v_1.

Case 1: ($u_1 \| v_1$ **or** $u_2 \| v_1$) Due to Lemma 1, S_v knew about u_1 or u_2 when S_v applied v_1. The condition in step 6 could not satisfied for S_v. This fact contradicts the assumption that `last_client` was not `null` when S_v received v_2. Therefore, the complement of the case assumption must be true. This value is $\neg((u_1 \| v_1) \vee (u_2 \| v_1)) = (v_1 \leadsto u_1 \vee u_1 \leadsto v_1) \wedge (v_1 \leadsto u_2 \vee u_2 \leadsto v_1)$.

Case 2: ($v_1 \leadsto u_1$) This contradicts the assumption that u_1 was applied before v_1.

Case 3: ($u_2 \leadsto v_1$) Since v_1 and v_2 were issued from the same client, $v_1 \leadsto v_2$. By transitivity, $u_2 \leadsto v_2$ which contradicts the assumption that $u_2 \| v_2$.

Case 4: ($u_1 \leadsto v_1$ **and** $v_1 \leadsto u_2$) In this case, v_1 would have been received at S_u after u_1 but before u_2. If v_1 was received at S_u after u_1 was applied, then `last_client` would be set to `null` in step 2 for v_1. If v_1 was received at S_u before u_1 was applied, then the condition in step 6 could not satisfied for S_u. These implications contradict the assumption that `last_client` was not `null` when S_u received u_2. □

Protocol 5 can be optimized using Lemma 3. When a server marks an update with `fast_flag`, then that update can be executed immediately since it is guaranteed to be before any other concurrent updates. This optimization is expressed in Protocol 6.

Protocol 6 *A local variable* `last_client` *is initialized to* `null`.

1. *C broadcasts u to all servers.*
2. *At each server, if* `last_client` *is equal to C, then execute the following:*
   (a) *If there are no other pending updates, then apply u immediately, and reply to C.*
   (b) *Mark u with* `fast_flag`.
3. *Else, set* `last_client` *to* `null`.
4. *Each server broadcasts u again.*
5. *Each server waits for all secondary broadcasts of u.*
6. *At each server, if u was not applied in step 2a, then apply u in a consistent order. The ordering is the same as in Protocol 5.*
7. *At each server, if there is no pending update from a client other than C, then* `last_client` *is set to C.*
8. *At each server, reply to C if this was not done in step 2a.*

This protocol is essentially identical to Protocol 5, except for the shortcut in step 2a. This shortcut allows most of the work to be accomplished asynchronously. The broadcast in step 4 can be delayed without blocking the client until the condition in step 2a is violated. At that point descriptors for all of the previous updates are sent with the next broadcast. Protocol 6 is stream efficient because a stream of updates from client C will cause `last_client` to be set to C for the duration of the stream. The shortcut in step 2a will be used for all updates but the first in a stream.

It is possible to reduce the total number of messages in Protocol 6 by adding 1 SNP. The change is that a designated server collects and distributes the secondary broadcasts in step 4. Thus, 1 SNP and n^2 messages are traded for 2 SNPs and

$2r-2$ messages. A total of 4 SNPs and $4r-2$ messages are necessary in the worst case.

Deceit uses 4 SNPs and $4r-2$ messages to acquire a write-token and broadcast an update from a server. We demonstrated in Section 4.4.1 that this is the optimal cost. However, Deceit does not use Protocol 6. The reason is that Protocol 6 is too complex. A token protocol is easier to understand and debug. It serves to prove a point, but it is impractical for implementation.

Since Deceit does not use Protocol 6, it is reasonable to suspect that there is a cost in performance. This cost appears as a race condition during token passing in Deceit. If server S_1 is passing the write-token to server S_2, and server S_3 requests the token at the same time, it is possible for S_3 to "miss" the token at both servers. The S_3's request could be received at S_1 after it has released the token, and the request could be received at S_2 before it has acquired the token. Afterwords, S_3 must try again with a new token request. Therefore, the worst case cost in Deceit is larger than 4 SNPs, but this case is very rare.

4.4.3 Total `Write` Cost in Deceit

Deceit takes 7 SNPs in the worst case `write` operation: 2 SNPs for the NFS RPC, 3 SNPs for write-token acquisition/file group join (Section 3.3.2), and 2 SNPs to synchronously broadcast the update. From the result in Section 4.4.1, it would seem that only 6 SNPs are necessary. However, we will argue that the extra SNP is required.

The extra SNP is in the file group join[1]. Theoretically, it is not necessary to join the file group in order to accomplish a `write` operation. The server could simply broadcast the update to *all* servers. However, a server must know the group membership to be able to broadcast a `write` to only the group members. In ISIS,

[1]This is the difference between the cost of a group join with token request and a plain token request.

there is no reliable way to determine the group membership without actually being a member of the group. Therefore, the group join is necessary in Deceit.

4.5 Design Trade-offs

There are a variety of conclusions that can be drawn from the results in this chapter. These conclusions take the form of design trade-offs. In this section, we discuss the trade-offs associated with file availability, stream efficiency, and special disk controller hardware.

File Availability

File availability is a clear design trade-off. It was illustrated in Table 4.6 on page 114 that there is more than an order a magnitude loss of availability by requiring a quorum of two servers instead of one. In other words, it is a very big step from maximum availability to the next level. If availability or reliability are very important to an application, then it is necessary to provide maximum availability, or to have a large number of replicas (i.e. at least 4).

Unfortunately, maximum availability conflicts with important safety properties. For example, maximum availability prevents update stability. Update stability captures the idea that the file system does not forget updates. Maximum availability also prevents one-copy serializability (1CS). 1CS captures the idea that replication is hidden from the client. Finally, maximum availability prevents prevents data consistency. Data consistency captures the idea that all replicas converge to a single value.

Automatic directory reconciliation (ADR) is an example of an attempt to provide data consistency with maximum availability. Unfortunately, ADR protocols tend to underestimate the semantic requirements on directories. All of the existing ADR implementations reconstruct directories by simply merging the list of directory updates. Only obvious conflicts, such as two files with the same name, are brought to the attention of the user. Some applications use directories as a form a structured file, and they are easily disrupted by ADR protocols.

Stream Efficiency

Another design trade-off is stream efficiency. A stream efficient file system has poorer worst-case performance, but better typical-case performance, than a general file system. In the worst case, an extra SNP or two are required. This cost intuitively corresponds to setting up the servers for a stream from a client. It is similar to passing a token in a token based protocol. Once the token is passed, a client has a fast, exclusive channel for broadcasting updates. In a general file system, there is no cost associated with setting up a fast channel.

Protocol 6 is an example of a stream efficient protocol. It is similar to a token-based protocol, except the knowledge of the token location is distributed over all of the servers. In effect, client C gets the "token" when any server sets the value of its variable `last_client` to the address of C. The protocol insures that at most one client will have the token at any time. By distributing the token location knowledge, token acquisition can be overlapped with update propagation. Also, the problem of starvation due to token contention is eliminated. If there are competing updates, then no client gets the token, and all operations require more than 2 SNPs. Unfortunately, Protocol 6 is difficult to implement, particularly in an environment where crashes are common. Therefore, it is inappropriate for most applications.

Disk Controller Hardware

There is a design trade-off in the use of special disk controller hardware. The problems of consistency and update propagation vanish if multi-ported disk controllers are used. If all server hosts can communicate with all disks, and this communication channel can not be broken, then replication becomes almost trivial. A single server can accomplish any operation by directly accessing all of the disks. The software is vastly simplified, and data consistency is easy. The disadvantages of this approach are that unusual hardware is required, and replication is limited by physical constraints. All replicas will need to be within a short physical distance from each other. Therefore, the file system will still be vulnerable to site disasters.

Chapter 5

Conclusion

This thesis had three main goals. The first goal was to characterize the semantics and performance of a wide range of file systems. We accomplished this goal with the file system survey in Chapter 2. A simplified model was used so that the selected file systems could be compared on the basis of safety and a simple model of performance.

The second goal was to build a file system that allowed many possible combinations of safety and performance. The product of this goal was the Deceit File System described in Chapter 3. This file system was used to explore exotic combinations of safety and performance thereby broadening the survey from Chapter 2. The flexible design allowed us to isolate particular safety properties for a detailed analysis. A second reason for building a flexible file system was that we believed such a file system could produce general performance benefits by allowing the user to tune the file system separately for each file.

The third goal was to explore the relationship between semantics and performance. The bulk of our work is found in Chapter 4. The survey from Chapter 2 provided examples of performance trade-offs from which general conclusions were suggested. Also, the development of Deceit led to an intuitive understanding of these trade-offs. The implicit assumption was that if it seemed impossible to meet a particular performance level, then some safety property could be shown to be the culprit. In the process of determining property relationships, several new properties were defined.

5.1 File System Properties

5.1.1 Safety Properties

There are two categories of file system properties that were presented in this thesis: traditional properties and new properties. The traditional properties are as follows:

1. *Client cache consistency* indicates that the file data caches on the clients are changed synchronously whenever the underlying file data is changed.

2. *Data consistency* indicates that all file replicas eventually converge on a single value.

3. *One-copy serializability* indicates that file system behavior is insulated from any underlying replication.

4. *Update stability* indicates that completed operations are stable with respect to failure.

5. *Atomicity* indicates that failure will not cause the file system to enter an inconsistent state.

6. *Maximum availability* indicates that a `read` or `write` to a file will always succeed if any replica of the file is available.

The new properties are as follows:

1. *Update monotonicity* indicates that files are allowed to diverge into separate versions, but this divergence is well behaved.

2. *Update persistence* indicates that all file data must become available to all clients eventually.

3. *Stream efficient* indicates that the file system is fully optimized for long streams of updates for a particular file from a single source.

Most surveyed file systems that supported replication also provided maximum availability[1]. We found that maximum availability prevented several other desirable safety properties. First, we determined that maximum availability precluded update stability. Put simply, a file system that provides maximum availability may be forced to switch to a server with stale data and thereby temporarily lose updates. It followed that one-copy serializability (1CS) was also incompatible with maximum availability since update stability is a consequence of 1CS.

One-copy serializability (1CS) is a very useful property since it insures that software which is written for a traditional file system will continue to run correctly on a file system with replication. Since 1CS was not compatible with maximum availability, it was necessary to determine a property that was near 1CS and was also compatible with maximum availability. In effect, we attempted to formalize the statement "file system X provides 1CS most of the time." Update monotonicity satisfied that goal, and it seemed to be intuitively correct since nearly all file systems actually provided update monotonicity. Similarly, update stability had to be weakened, and the result was update persistence. Update stability states that the result of an update will be *immediately* available, and update persistence says only that it will be *eventually* available. Together, update monotonicity and update stability formed a powerful combination for proving performance and safety properties.

Using these two new properties, we showed that maximum availability precluded data consistency. This result was strongly indicated because none of the file systems

[1]These file systems were RNFS, HA-NFS, Locus, Ficus, Coda, and Deceit.

with replication provide data consistency except for the one that does not provide maximum availability (i.e. Echo). Some file systems with maximum availability attempted to provide data consistency in the directory structure by using automatic directory reconciliation (ADR). Directories tend to be a target of inconsistency, and their semantics are understood, so they are a natural candidate for automatic reconciliation. We demonstrated the ADR is a fundamentally flawed concept for general purpose file systems because ADR violates directory semantics that some applications depend on.

5.1.2 Performance

Following the discussion of safety properties, some performance bounds for the `write` operation were presented. Update monotonicity and update stability were used for all of these results. The basic lower bound was that 2 SNPs are insufficient in the worst case. Coda helped to demonstrate this point since it violated update monotonicity and only used 2 SNPs in every `write`. To complement this conclusion, Protocol 3 on page 127 showed that 3 SNPs and $2r$ messages are sufficient in all cases. Between these two bounds, there was room for a protocol that used 2 SNPs in most cases but 3 SNPs occasionally.

The desired protocol should use 2 SNPs in the common case, and we formalized this property with the definition of stream efficient. A typical file system experiences updates from a single source for any one file, and stream efficiency states that only 2 SNPs are used under these conditions. We demonstrated that a stream efficient file system would required more messages in the worse case to offset its good performance in the typical case. In the worst case, a stream efficient file system needs 3 SNPs and $O(r^2)$ messages. Finally, we presented Protocol 6 on page 131, a stream efficient update protocol that meets this bound.

5.2 File Parameters and Flexibility

The file parameters in Deceit can have a dramatic effect on performance and safety. For example, Table 5.1 describes the cost of different degrees of replication. Writing to a file with three replicas costs a factor of 5 more messages more than writing to a file with one replica (Section 4.1.1). Also, three rounds of communication can be required rather than one round. In the Deceit implementation, using three replicas is 46% slower than one replica for the Andrew Benchmark (Section 3.8.2).

Table 5.1: Worst-case `Write` Operation Cost by Degree of Replication

	Theoretical Minimum		Deceit	
r	SNPs	messages	SNPs	messages
1	2	2	2	2
2	4	4	6	6
3	5	10	6	10
4	5/6	17/12	6	14
5	5/6	26/16	6	18
6	5/6	37/20	6	22

If the file parameters had a large granularity (e.g. a set of parameters per directory), then many files would be unnecessarily safe. For example, it is reasonable to have 2 or 3 replicas of an important program source file, but it is not reasonable to replicate a core dump because it happens to be in the same directory. The ability to tune the replication level of each file allows the user to maximize safety for important files and to maximize performance for others. A result is that overall efficiency is improved.

This form of flexibility has several costs. There is the administrative cost of setting the file parameters for each file. It is almost impossible for an unsophisticated user to understand, much less set, an appropriate value for "stability monitoring." Heuristics based on file name matching or directories could be used to reduce this problem. In addition to the administrative cost, there is a cost in server size and complexity. The server program must be prepared for every possible combination of parameters settings. In effect, the server must be as complex as necessary for the most demanding possible setting, instead of the typical setting. Each file can be manipulated independently, so every file represents an instance of the entire set of control protocols, which can greatly increase the storage requirements for each file.

The Deceit file system is an attempt to provide this form of flexibility. The options that Deceit provides are listed in Section 3.5 on page 76. Below is a summary of the effects of each file parameter on safety and performance. We analyze each file parameter using the results from Section 3.8 and Section 4.1.

5.2.1 Minimum Replica Level

The most pervasive control option is the minimum replica level (r). The value of r effects the total amount of storage for the file, the availability of the file, and the cost of most file operations. The file availability is effected in a complex way depending on the policy for reading and writing during a failure. Maximum and minimum availability (Tables 1.2 and 1.3 on page 9) mark the extremes.

`Read` performance can be improved by using a large value for r. The reason is that a large value raises the probability that a `read` operation will not have to be forwarded. If the client picks a server without a replica, then 3 SNPs are necessary. Deceit will use 4 SNPs and 4 messages in this case. Otherwise Deceit uses 2 SNPs and 2 messages, which is minimal. In the actual implementation, we measured the time required to forward a `read` operations at 18 milliseconds which is small in comparison to disk access times.

`Write` performance is also effected strongly by the replication level. If there is no replication, then the minimum cost is 2 SNPs and 2 messages. If there is replication, then the results in Sections 4.4.1 and 4.4.2 apply. A detailed description of Deceit performance in this case is provided in Section 4.1.2. Table 5.1 summarizes the results for worst case cost of a `write` operation. This table assumes that the client communicates with only one server, the server has a replica of the file, and the file system is stream efficient. Table 5.1 illustrates that Deceit performs optimally for $r = 1$ and nearly optimally $r > 3$. For $r = 2$ or $r = 3$, Deceit is less efficient.

The Andrew Benchmark (Section 3.8.2) illustrated the cost of replication in the implementation. For the overall benchmark, Deceit with $r = 2$ was 26% slower than $r = 1$, and Deceit with $r = 3$ was 46% slower. Further analysis revealed the `WRITE` operation was only 17% slower for $r = 3$, but the `CREATE` operation was fully 81% slower. A `CREATE` operation involves several name service operations, and the name service scaled poorly as replication was increased. A redesign of the name service would partially alleviate the problem.

5.2.2 Write Safety Level

The write safety level (s) is a subtle control option. There are three interesting value ranges: $s = 0$, $0 < s < m$, and $s = m$ (where m is the maximum replica level).

If $s = 0$, then `write` operations are asynchronous. No SNPs are required; all communication can be in the form of ANPs. Update stability and update persistence are impossible: if a server fails before completing a `write`, then the client would not know that the `write` failed. In general, Deceit does not provide update stability, but it is much less likely to be violated if $s > 0$.

If $0 < s < m$, then there are stronger guarantees about the success of the `write`. When the RPC returns to the client, the client knows that the `write` has been recorded on at least one disk. Therefore update persistence is provided, although update stability is still impossible. With this setting, 2 SNPs and 1 SDA are required for each `write`.

If $s = m$, then the safety guarantees depend on other file parameters. When the RPC returns to the client, the client knows that the `write` has been recorded on the disks at all available replicas, but necessarily at *all* replicas. Deceit still does not provide update stability in this case, but the scenario that violates it is extremely unlikely.

As s is increased from 1 to m, the expected time to completion also increases since a `write` completes as quickly as the s fastest disks. Table 5.2 illustrates this rise in expected completion time using a normal distributed for disk access times. In Table 5.2, it is assumed that a disk drive completes a `write` in an average of 1 unit of time with a standard deviation of 0.2. The table shows the expected time for the fastest s out of r drives to finish.

Table 5.2: Expected Time for the Fastest s out of r Disks

	s					
r	1	2	3	4	5	6
2	0.889	1.114	-	-	-	-
3	0.832	1.000	1.170	-	-	-
4	0.793	0.942	1.059	1.205	-	-
5	0.767	0.901	1.000	1.100	1.232	-
6	0.747	0.872	0.961	1.039	1.129	1.254

We measured the `write` times for different values for s in the Deceit implementation. These results are shown in Section 3.8.5. The main conclusion was that server performance was not strongly effected by s for values $s \geq 1$. When s was changed from 1 to 3, the performance degradation was only 9.8%. Even when large random delays (uniformly chosen between 0 and 50 milliseconds) were added to disk access times, the performance degradation was only 17.6%. The conclusion is that s is not a well chosen file parameter. It would be better to simply be able to switch the server between "asynchronous mode" and "synchronous mode."

The measured times for $s = 0$ were discouraging. The NFS protocol does not allow truly asynchronous operation, so setting $s = 0$ in Deceit does not eliminate the 2 SNPs from the operation cost. Also, in the benchmark Deceit became overwhelmed with requests, so the server was not able to respond to new requests rapidly. The result was that a `write` to three replicas required 46.6 milliseconds (rather than the 18 milliseconds required for pure communication). The conclusion is that setting $s = 0$ is a bad choice unless the server is lightly loaded, and the user is very confident that the `write` will succeed (or the data is unimportant).

5.2.3 Stability Monitoring

Stability monitoring has relatively little cost and effect. The cost is that on occasion `reads` cost 4 SNPs instead of 2 SNPs. In the implementation of Deceit, the extra 2 SNPs would cost 18 milliseconds for an 8 Kb `READ`.

The benefit of stability monitoring is to help provide one-copy serializability. Stability monitoring is significant only when one client is writing to a replicated file, and another client is concurrently reading the file at a different server. In any other case, the other protocols and ISIS message ordering provide one-copy serializability without assistance. Regardless of stability monitoring, one-copy serializability can be violated if a server crashes.

Stability monitoring added substantially to the complexity of the server, and it was aesthetically unpleasant. Each server had to maintain extra status information about each segment, and it was difficult to recover to a consistent state after a crash. The conclusion is that the stability monitoring protocol is unsatisfactory. A better approach would be to use a real distributed locking protocol. Perhaps this protocol also could be used to replace write-tokens.

5.2.4 File Migration

File migration is difficult to analyze due to its heuristic nature. The benefit is that a `read` costs less if the file is logically closer to the client that issues the `read`. Also availability will be increased if the data is physically closer to the client. The cost of file migration is that the data needs to be read, transmitted, and rewritten. Migration is most effective on small files that tend to be accessed by a single client for long periods (e.g. private source files in a user directory). In this case, there is a low overhead with a high payoff.

The success of file migration depends critically on the policy describing when to migrate files. In Deceit the policy comes in two parts. First, users must enable migration on files using a file parameter. Second, a file is migrated asynchronously after it is read at a server that does not have a replica of the file. In this way, a server collects the working set of files that it needs. This policy is a form of caching.

We measured two key values in the implementation: the file transfer rate and the cost of forwarding a `read`. The file transfer rate was 55 Kb per second plus 148 milliseconds of overhead. This rate is very slow compared to the raw network bandwidth (1.25 Mb/sec) and disk bandwidth (3 Mb/sec). The main reason for the slow rate was that Deceit was not able to efficiently schedule access to the disk. A large collection of `write` could not be adequately processed in parallel. The cost of forwarding a 8 Kb `read` was 18 milliseconds, which is small in comparison to disk access time. The conclusion is that file migration is not valuable in general. A user will want to migrate a file only under unusual conditions, and a user level utility command would solve the problem more cleanly than a file parameter.

5.2.5 Write Quorum Size

The write quorum size (q) controls the trade-off between write availability and data consistency. A low value for q increases write availability and the risk of inconsistency between replicas. Tables 4.6 and 4.7 on page 114 provide a numerical description of availability for different values of r and q. Figure 5.1 summarizes the same results in graphical form. It is clear that there is a large availability penalty incurred by increasing q. This jump is most dramatic between $q = 1$ and $q = 2$; it is more than a factor of 10.

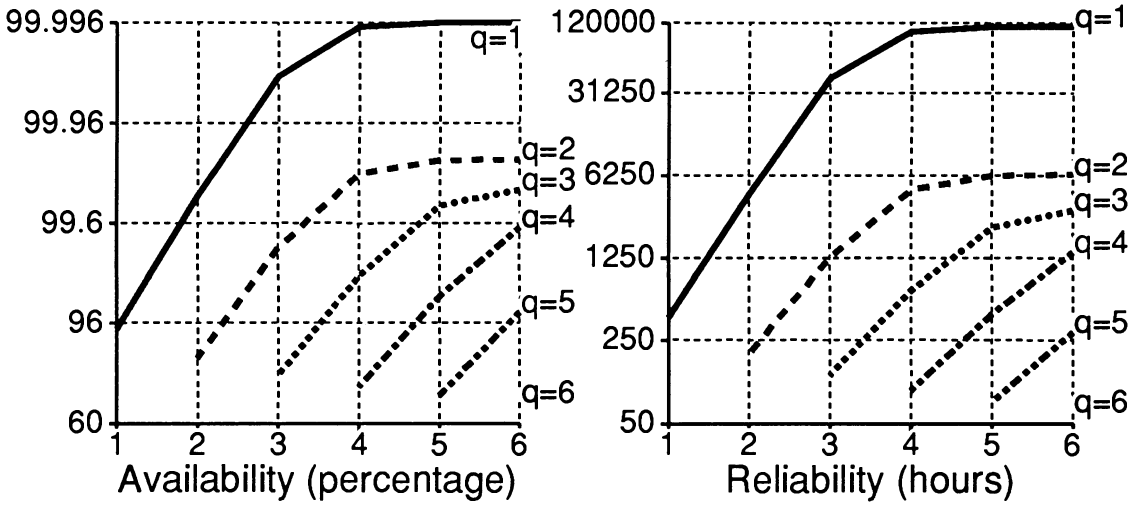

Figure 5.1: Availability and Reliability by Quorum

It is more difficult to quantify the risk of inconsistency between replicas. For this to occur, it is necessary that two clients concurrently write to a file on either side of a partition. Alternatively, crashes and recoveries can cause a client to switch from one set of replicas to another during a stream of updates. These possibilities are shown in Figure 5.2, and they are highly dependent on the expected usage and environment. In typical environments, it is very unusual for two clients to concurrently write to the same file. It is also unusual for network partition to occur (except for very brief ones).

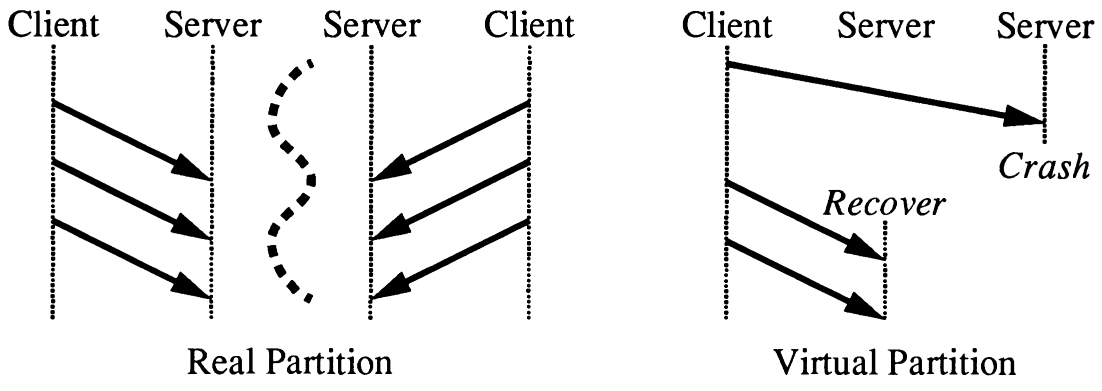

Figure 5.2: Scenarios for Replica Inconsistency

5.2.6 Maximum Replica Level

The last file parameter is the maximum replica level (m). This parameter limits the amount of replication. A low value for m imposes a cost in availability and expected `read` performance. A high value for m allows heavy consumption of disk space and increases the risk of data inconsistency.

The primary use for m is to prevent data inconsistency. If $q > \frac{m}{2}$, then it is theoretically impossible to create incomparable versions of the file. However, in Deceit there is a very improbable race condition that can produce data inconsistency despite using a large q. The essential problem is that Deceit may not know whether the write quorum is satisfied until *after* the `write` has completed. In conclusion, maximum replica level is a very valuable file parameter in Deceit since it is the only way to constrain resource consumption and to provide data consistency.

5.3 Future Direction in File Systems

Most file systems that support replication also violate basic file system properties such as one-copy serializability and update stability, which is not surprising since traditional semantics were defined on single processor time-shared systems. In order to fully realize the value of replication, the basic file system paradigm must be updated. A key problem with replication is maintaining the ordering of updates since one update can overwrite a previous one. If the semantics of a file could be changed so that updates were commutative, then a distributed file system could be simpler and more efficient. Servers could broadcast updates immediately without establishing a global order. Resolving inconsistent replicas would be simple and automatic: it only would be a matter of copying updates from one replica to the other.

One limited step in updating file semantics is the use of versions. Deceit, Locus, Ficus, and Coda allow replicas to become inconsistent during a partition, but after recovery there are multiple versions of the same data. Having multiple versions can create havoc with traditional applications, and the user must intervene to correct the situation. Also, one-copy serializability and update stability are impossible. Update persistence and update monotonicity were an effort to define safety in such an environment.

Traditional directory semantics hinder replication for the same reasons that traditional file semantics do. Deceit attempts to provide traditional directory semantics by replicating the entire image of the file system at all servers. This decision will inevitably create a scaling bottleneck: either all servers must retrieve directory data from some small set of directory servers, or the directory data must be replicated everywhere. Other systems have solved this problem by using *user centered naming*[CW88]. In this approach, each user has an encapsulated view of the file system that can be sent from server to server as the user moves around. Most directory changes take place in the user's private space and only affect the servers that store the user's view. In effect, user centered naming distributes the load by breaking the directory structure up into small, natural units.

New file systems that support replication and distribution are much more complex the older file systems. As a result, modularity is becoming vital in the implementation of these new systems. A distributed system is inherently very difficult to understand, so it is essential that it be built and tested in small pieces. Modularity serves to clarify the design, to isolate bugs (e.g. race conditions and memory leaks), and to provide a convenient entry point for regression testing. For example, Deceit is divided into 3 major pieces, with several subdivisions. This level of modularity was not enough: Deceit could be improved by further refinement into very simple, layered protocols.

Finally, it should be noted that specialized hardware can reduce server complexity. For example, HA-NFS uses dual-ported disks that eliminate nearly all communication directly between the servers. One CPU can write both disks in parallel without synchronization with the other CPU. Some researchers have used the RAID approach[SS90b] which is an elaboration of the some concept. The main limitation of specialized hardware is that servers usually must be physically close together. Also, it is not likely that such hardware will be generally available as the use of distributed file systems becomes more widespread.

Appendix A

NFS Protocol

The following material was taken from [Sun86b] and heavily edited. It is not original to this author. As such, it is not perfectly accurate or consistent. In particular, NFS servers are not truly stateless, and all the RPC types are not idempotent.

The Sun Network Filesystem (NFS) protocol provides transparent remote access to shared filesystems over local area networks. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call[Sun86c,Sun86d] (RPC) primitives built on top of an External Data Representation[Sun86a] (XDR). RPC prevides a clean, procedure-oriented interface to remote services. XDR provides a common way of representing data types over a network.

The NFS protocol is stateless. That is, a server does not need to maintain state about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a crash. With stateless servers, a client need only retry a request until the server responds. The client of a stateful server needs to detect a server crash and rebuild the server's state when it comes back up.

All file operations are done using file handles to refer to a file or directory.

The file handle can contain whatever information the server needs to distinguish an individual file. In the current implementation, the file handle also acts as a capability. A random 32-bit value is encoded into the handle, and this value is checked against a value in the i-node when the file is accessed.

Each RPC type will be described using the following notation:

• `TYPE` (arguments) returns (results)
description

where "`TYPE`" is the type of RPC, "arguments" is the list of argument values, "results" is the list of result values, and "description" is a textual description of the RPC. The NFS RPC types are the following:

• `NULL` () returns ()
This procedure does no work. It is made available to allow server response testing and timing.

• `GETATTR` (file handle) returns (file attributes)
This procedure reads the attributes associated with a file or directory. These attributes include size, owner, security, and timestamps.

• `SETATTR` (file handle, file attributes) returns ()
This procedure sets the attributes associated with a file or directory.

• `LOOKUP` (directory handle, name) returns (child handle, attributes)
This procedure determines the handle for an entry in a directory. When traversing a path, each path component requires a separate `LOOKUP` call. The attributes for the child are returned as in the `GETATTR` operation.

• `READLINK` (file handle) returns (string)

This procedure reads the contents of a symbolic link. A symbolic link is created using `SYMLINK`.

• `READ` (file handle, offset, size) returns (data, attributes)

This procedure reads a contiguous block of data from a normal file. A normal file is created using `CREATE`. NFS limits size to 8 Kbytes. The attributes for the file are returned as in the `GETATTR` operation.

• `WRITE` (file handle, offset, size, data) returns ()

This procedure writes a contiguous block of data to a normal file. NFS limits size to 8 Kbytes.

• `CREATE` (directory handle, name, attributes) returns (file handle, attributes)

This procedure creates a normal file in the specified directory with the specified name and attributes. If a normal file is already there, then it is truncated to zero length.

• `REMOVE` (directory handle, name) returns ()

This procedure deletes one hard link to a file or symbolic link. If this link is the last, then the disk space for the file is deallocated.

• `RENAME` (original directory handle, original name, new directory handle, new name) returns ()

This procedure replaces a hard link for a file, directory, or symbolic link. This operation is atomic on the server, it cannot be interrupted in the middle.

• `LINK` (file handle, new directory handle, new name) returns ()

This procedure creates a new hard link to a normal file. A hard link has the property that if a change is made through either of the links, then the change is reflected through both links.

• `SYMLINK` (directory handle, name, string, attributes) returns ()

This procedure creates a symbolic link. A symbolic link is a pointer to another file using "string" as a relative path name. The client interprets "string" when the symbolic link is traversed. There is no guarantee that the target of the link will exist or be consistent between clients.

• `MKDIR` (directory handle, name, attributes) returns (new directory handle, attributes)

This procedure creates a directory in the specified location. The creation will fail if another file or directory already occupies the location.

• `RMDIR` (directory handle, name) returns ()

This procedure deletes the specified directory. The deletion will fail if the directory is not empty.

• `READDIR` (directory handle, offset, size) returns ([handle, name, offset] list)

This procedure reads the contents of a directory. The contents are returned as a list of tuples where each tuple contains the file handle, file name, and current directory offset. The offset can be used in later `READDIR` operations to skip initial directory entries. At most "size" bytes of data are returned. NFS limits size to 8 Kbytes.

• `STATFS` (file handle) returns (file system attributes)

This procedures returned the attributes associated with a file system. The file handle specifies which file system to check. These attributes include total disk space, free space, and disk block size.

NFS has very limited security. A client must include the user id number, the group id number, and the client name with each RPC. This information is included in clear text with each RPC, and it is trivial to forge. Any process on a valid client has unlimited access to any exported files. The only substantial security feature is that most NFS servers verify the name of the client against the internet address included in the RPC. This address is important because it is used to send back the reply to the client. In order for an invalid client to access an NFS server, it is necessary to temporarily modify the network routing so that an RPC call with an authenticated return address can get back to the invalid client.

Bibliography

[ASC85] A. El Abbadi, Dale Skeen, and F. Cristian. An Efficient Fault-Tolerant Algorithm for Replicated Data Management. In *The Fourth ACM Symposium on Principles of Database Systems*, Portland, OR, March 1985. ACM.

[BEM90] Anupam Bhide, Elmootazbellah N. Elnozahy, and Stephen P. Morgan. Implicit Replication in a Network File Server. In *The Workshop on Management of Replicated Data*, pages 83–90, Houston, TX, November 1990. IEEE. Also in *Technical Committee on Operating Systems and Application Environments* (Newsletter), 4(3), Fall 1990.

[BEM91] Anupam Bhide, Elmootazbellah Elnozahy, and Stephen Morgan. A Highly Available Network File Server. In *Winter 1991 USENIX Conference*. USENIX Association, January 1991.

[BEMS91] Anupam Bhide, Elmootazbellah N. Elnozahy, Stephen P. Morgan, and Alex Siegel. A Comparison of Two Approaches to Build Reliable Distributed File Servers. In *The 11th International Conference on Distributed Computing Systems*, Arlington, TX, May 1991. IEEE.

[BJ87a] Kenneth Birman and Thomas Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *The Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138. ACM, November 1987.

[BJ87b] Kenneth Birman and Thomas Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions of Computer Systems*, 5(1), February 1987.

[BJKS] Kenneth Birman, Thomas Joseph, Kenneth Kane, and Frank Schmuck. *ISIS - A Distributed Programming Environment – User's Guide and Reference Manual.* Cornell University Computer Science Department, Ithaca, New York.

[BP75] Richard E. Barlow and Frank Proschan. *Statistical Theory of Reliability and Life Testing (Probability Models).* International Series in Decision Processes, and Series in Quantitative Methods for Decision Making. Holt, Rinehart and Winston, Inc., 1975.

[Bre83] Pearl Brereton. Detection and Resolution of Inconsistencies Among Distributed Replication of Files. *Operating Systems Review*, 17(1):10–15, January 1983.

[BSS90] Kenneth Birman, Andre Schiper, and Pat Stephenson. Fast Causal Multicast. Technical Report TR90-1105, Cornell University Computer Science Department, Ithaca, NY, April 1990. Submitted to ACM Transactions on Computer Systems.

[Cri91] Flaviu Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[CW87] Luis Felipe Cabrera and Jim Wyllie. QuickSilver Distributed File Services: An Architecture for Horizontal Growth. Research Report RJ 5578 (56697), IBM Almaden Research Center, San Jose, CA (USA), April 1987.

[CW88] Luis Felipe Cabrera and Jim Wyllie. QuickSilver Distributed File Services: An Architecture for Horizontal Growth. In *1988 IEEE 2nd Conference On Computer Workstations*, pages 23–37. IEEE, March 1988.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *The Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12. ACM, August 1987.

[Dig84] Digital Equipment Corportion, Maynard, Massachusetts. *VAX/VMS - Introduction to VAX/VMS*, VAX/VMS Version 4.0 edition, September 1984.

[Dio80] Jeremy Dion. The Cambridge File Server. *Operating Systems Review*, 14(4):26–35, October 1980.

[DO89] Fred Douglis and John Ousterhout. Log-Structured File Systems. In *COMPCON Spring '89, Digest of Papers*, pages 124–129. IEEE, February 1989.

[Flo86a] Rick Floyd. Directory Reference Patterns in a UNIX Environment. Technical Report 179, University of Rochester, August 1986.

[Flo86b] Rick Floyd. Short-Term File Reference Patterns in a UNIX Environment. Technical Report 177, University of Rochester, March 1986.

[GHM+90] Richard Guy, John Heidemann, Wai Mak, Thomas Page Jr., and Gerald Popek. Implementation of the Ficus Replicated File System. In *Summer 1990 USENIX Conference*, pages 63–71, Anaheim, CA, June 1990. USENIX Association.

[GNS88] David Gifford, Roger Needham, and Michael Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, March 1988.

[GP90] Richard Guy and Gerald Popek. Reconciling Partially Replicated Name Spaces. Technical Report CSD-900010, Department of Computer Science at the University of California Los Angeles, April 1990.

[Gra85] Jim Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, Tandem Computers Inc., Cupertino, CA, June 1985.

[Gra90] Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. Technical Report 90.1 (Part no. 33579), Tandem Computers Inc., Cupertino, CA, January 1990.

[Hac85] Anna Hac. Distributed File Systems - A Survey. *Operating Systems Review*, 10(1):15–18, January 1985.

[Hag87] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *The Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162. ACM, November 1987.

[HBJ+90] Andy Hisgen, Andrew Birrell, Chuck Jerian, Timothy Mann, Michael Schroeder, and Garret Swart. Granularity and Semantic Level of Replication in the Echo Distributed File System. In *The Workshop on Management of Replicated Data*, pages 2–4, Houston, TX, November 1990. IEEE. Also in *Technical Committee on Operating Systems and Application Environments* (Newsletter), 4(3), Fall 1990.

[HBM+89] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart. Availability and Consistency Tradeoffs in the Echo Distributed File System. In *The 2nd Workshop of Workstation Operating Systems*, pages 49–54, Pacific Grove, CA, September 1989. IEEE.

[Her85] Maurice Herlihy. Comparing How Atomicity Mechanisms Support Replication. In *The Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 102–110. ACM, August 1985.

[Her87] Maurice Herlihy. Concurrency Versus Availability: Atomicity Mechanisms for Replicated Data. *ACM Transactions on Computer Systems*, 5(3):249–274, August 1987.

[HKM+88] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and Performance in a Distributed File System. *ACM Transactions of Computer Systems*, 6(1):51–81, February 1988.

[HMSC88] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 133–145. ACM, January 1988.

[JB86] Thomas Joseph and Kenneth Birman. Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. *ACM Transactions of Computer Systems*, 4(1), February 1986.

[JPGH90] Thomas Page Jr., Gerald Popek, Richard Guy, and John Heidemann. The Ficus Distributed File System: Replication via Stackable Layers. Technical Report CSD-900009, Department of Computer Science at the University of California Los Angeles, April 1990.

[LCP90] Darrell Long, John Carroll, and C. J. Park. A Study of the Reliability of Internet Sites. Technical Report UCSC-CRL-90-46, Department of Computer and Information Sciences at University of California Santa Cruz, Santa Cruz, CA, 1990.

[LeL78] Gerard LeLann. Algorithms for Distributed Data Sharing Systems which use Tickets. In *Third Berkeley Workshop*, pages 259–272, August 1978.

[LeL81] Gerard LeLann. *Distributed Systems - Architecture and Implementation, An Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, chapter 12, pages 278–282. Springer-Verlag, 1981.

[LZCZ86] Edward Lazowska, John Zahorjan, David Cheriton, and Willy Zwaenepoel. File Access Performance of Diskless Workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[MHS89] Timothy Mann, Andy Hisgen, and Garret Swart. An Algorithm for Data Replication. Technical Report 46, Digital Equipment Corporation Systems Research Center, June 1989.

[MS87] Keith Marzullo and Frank Schmuck. Supplying High Availability with a Standard Network File System. Technical Report 87-888, Department of Computer Science at Cornell University, December 1987.

[MSC+86] James Morris, Mahadev Satyanarayanan, Michael Conner, John Howard, David Rosenthal, and F. Donelson Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[MT84] Sape Mullender and Andrew Tanenbaum. Protection and Resource Control in Distributed Operating Systems. *Computer Networks*, 8(5,6):421–432, October 1984.

[Mul89] Sape Mullender, editor. *Distributed Systems*, chapter 13–14. Addison-Wesley, 1989.

[Neu89] B. Clifford Neuman. Workstations and the Virtual System Model. Technical Report 89-10-10, Department of Computer Science and Engineering at University of Washington, October 1989.

[NO88] Michael Nelson and John Ousterhout. Copy-on-Write for Sprite. In *Summer 1988 USENIX Conference*, pages 187–201, San Francisco, CA, June 1988. USENIX Association.

[NWO88] Michael Nelson, Brent Welch, and John Ousterhout. Caching in the Sprite Network File System. *ACM Transactions of Computer Systems*, 6(1), February 1988.

[OCD+88] John Ousterhout, A. Cherenson, Fred Douglis, Michael Nelson, and Brent Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[OCH+85] John Ousterhout, Herve Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *The Tenth ACM Symposium on Operating Systems Principles*, pages 15–24. ACM, December 1985.

[OD89] John Ousterhout and Fred Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[P8̂6] Jehan-Francois Pâris. Voting with Witnesses: A Consistency Scheme for Replicated Files. In *The 6th International Conference on Distributed Computer Systems*, pages 606–612, Cambridge, MA, May 1986. IEEE.

[P8̂9] Jehan-Francois Pâris. Voting with Bystanders. In *The 9th International Conference on Distributed Computing Systems*, pages 394–401, Newport Beach, CA, June 1989. IEEE.

[PGJH90] Gerald Popek, Richard Guy, Thomas Page Jr., and John Heidemann. Replication in Ficus Distributed File Systems. In *The Workshop on Management of Replicated Data*, pages 5–10, Houston, TX, November 1990. IEEE. Also in *Technical Committee on Operating Systems and Application Environments* (Newsletter), 4(3), Fall 1990.

[Pug90] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[RO90] Mendel Rosenblum and John Ousterhout. The LFS storage manager. In *Summer 1990 USENIX Conference*, pages 315–324, Anaheim, CA, June 1990. USENIX Association.

[Ros90] David Rosenthal. Evolving the Vnode Interface. In *Summer 1990 USENIX Conference*, pages 107–117, Anaheim, CA, June 1990. USENIX Association.

[Sat89a] Mahadev Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[Sat89b] Mahadev Satyanarayanan. Scalable, Secure and Highly Available File Access in a Distributed Workstation Environment. Technical report, Department of Computer Science at Carnegie Mellon University, October 1989.

[Sat90a] Mahadev Satyanarayanan. A Survey of Distributed File Systems. *Annu. Rev. Comput. Sci.*, 4:73–104, 1990.

[Sat90b] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *Computer*, 23(5):9–21, May 1990.

[SB89] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. In *The Twelfth ACM Symposium on Operating Systems Principles*, pages 83–90. ACM, December 1989.

[SBM89] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A Flexible Distributed File System. Technical Report TR 89-1042, Department of Computer Science at Cornell University, Ithaca, NY, November 1989.

[SBM90a] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A Flexible Distributed File System. In *Summer 1990 USENIX Conference*, pages 51–61, Anaheim, CA, June 1990. USENIX Association.

[SBM90b] Alex Siegel, Kenneth Birman, and Keith Marzullo. Position Paper for Deceit: A Flexible Distributed File System. In *The Workshop on Management of Replicated Data*, pages 15–17, Houston, TX, November 1990. IEEE. Also in *Technical Committee on Operating Systems and Application Environments* (Newsletter), 4(3), Fall 1990.

[Sch84] Fred Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Sch88] Frank Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems.* Ph.D. dissertation, Cornell University, August 1988.

[SGN85] Michael Schroeder, David Gifford, and Roger Needham. A Caching File System For a Programmer's Workstation. In *The Tenth ACM Symposium on Operating Systems Principles*, pages 25–34. ACM, December 1985.

[SHN+85] Mahadev Satyanarayanan, John Howard, David Nicols, Robert Sidebotham, Alfred Spector, and Michael West. The ITC Distributed File System: Principles and Design. In *The Tenth ACM Symposium on Operating Systems Principles*, pages 35–50. ACM, December 1985.

[Ske85] Dale Skeen. Determining the Last Process to Fail. *ACM Transactions on Computer Systems*, 3(1):15–30, February 1985.

[SKK+89] Mahadev Satyanarayanan, James Kistler, Puneet Kumar, Maria Okasaki, Ellen Siegel, and David Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. Technical Report CMU-CS-89-165, Department of Computer Science at Carnegie Mellon University, November 1989.

[SKK+90] Mahadev Satyanarayanan, James Kistler, Puneet Kumar, Maria Okasaki, Ellen Siegel, and David Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[SKS90] David Steere, James Kistler, and Mahadev Satyanarayanan. Efficient User-Level File Cache Management on the Sun Vnode Interface. In *Summer 1990 USENIX Conference*, pages 325–331, Anaheim, CA, June 1990. USENIX Association.

[SM89] V. Srinivasan and Jeffrey Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *The Twelfth ACM Symposium on Operating Systems Principles*, pages 45–57. ACM, December 1989.

[SMI80] Howard Sturgis, James Mitchell, and J. Israel. Issues in the Design and Use of a Distributed File System. *Operating Systems Review*, 14(3):55–69, July 1980.

[SNS88] J. G. Steiner, B. Clifford Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Winter 1988 USENIX Conference*, Dallas, TX, 1988. USENIX Association.

[SS83] Richard Schlichting and Fred Schneider. Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[SS90a] Mahadev Satyanarayanan and Ellen Siegel. Parallel Communication in a Large Distributed Environment. *IEEE Transactions of Computers*, 39:328–348, March 1990.

[SS90b] Michael Stonebraker and Gerhard Schloss. Distributed RAID – A New Multiple Copy Algorithm. In *Proceedings of the Sixth Conference on Data Engineering*. IEEE, February 1990.

[Sta88] Carl Staelin. File Access Patterns. Technical Report CS-TR-179-88, Department of Computer Science at Princeton University, September 1988.

[Sun86a] Sun Microsystems, Inc., Mountain View, CA. *External Data Representation Protocol Specification*, February 1986.

[Sun86b] Sun Microsystems, Inc., Mountain View, CA. *Network File System Protocol Specification*, February 1986.

[Sun86c] Sun Microsystems, Inc., Mountain View, CA. *Remote Procedure Call Programming Guide*, February 1986.

[Sun86d] Sun Microsystems, Inc., Mountain View, CA. *Remote Procedure Call Protocol Specification*, February 1986.

[Sun90] Sun Microsystems, Inc., Mountain View, CA. *Network Programming Guide*, March 1990. Part No. 800-3850-10.

[Svo80] Liba Svobodova. Management of Object Histories in the Swallow Repository. Technical Report MIT/LCS/TR-243, Laboratory for Computer Science at Massachusetts Institute of Technology, July 1980.

[Svo81] Liba Svobodova. A Reliable Object-Oriented Data Repository for a Distributed Computer System. In *The Eighth Symposium on Operating Systems Principles*, pages 47–58. ACM, December 1981.

[TCW89] Marvin Theimer, Luis-Felipe Cabrera, and Jim Wyllie. QuickSilver Support for Access to Data in Large, Geographically Dispersed Systems. In *The 9th International Conference on Distributed Computing Systems*, pages 28–35, Newport Beach, CA, June 1989. IEEE.

[TM81] Andrew Tanenbaum and Sape Mullender. An Overview of the Amoeba Distributed Operating System. *Operating Systems Review*, 15(3):51–64, July 1981.

[TMvR86] Andrew Tanenbaum, Sape Mullender, and Robbert van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *The 6th International Conference on Distributed Computing Systems*, pages 558–563. IEEE, May 1986.

[TvRvS+90] Andrew Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory Sharp, Sape Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[vRT88] Robbert van Renesse and Andrew Tanenbaum. Voting with Ghosts. In *The 8th International Conference on Distributed Computing Systems*, pages 456–462. IEEE, June 1988.

[Wel90] Brent Welch. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System.* Ph.D. dissertation, University of California, Berkeley, CA 94720, February 1990. Technical Report UCB/CSD 90/567.

[WO86] Brent Welch and John Ousterhout. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System. In *The 6th International Conference on Distributed Computing Systems*, pages 184–189. IEEE, May 1986.

[WO88] Brent Welch and John Ousterhout. Pseudo Devices: User-Level Extensions to the Sprite File System. In *Summer 1988 USENIX Conference*, pages 37–49, San Francisco, CA, June 1988. USENIX Association.

[WO89] Brent Welch and John Ousterhout. Pseudo-File-Systems. Technical Report UCB/CSD 89/499, Computer Science Division (EECS), University of California, Berkeley, CA 94720, April 1989.

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. In *The*

Ninth ACM Symposium on Operating Systems Principles, pages 49–70. ACM, October 1983.

[WW90] Thomas H. Wonnacott and Ronald J. Wonnacott. *Introductory Statistics*. John Wiley and Sons, fifth edition, 1990.