Deceit: A Flexible Distributed File System*

Alex Siegel
Kenneth Birman
Keith Marzullo

TR 89-1042
November 1989

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

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N0014-87-C-8904, and under NASA-Ames grant no. NAG 2-593. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Deceit: A Flexible Distributed File System *

Alex Siegel,
Kenneth Birman, and Keith Marzullo
{siegel,ken,marzullo}.cs.cornell.edu

Cornell University
Ithaca, NY

December 7, 1989

Abstract

Deceit, a distributed file system being developed at Cornell, focuses on flexible file semantics in relation to efficiency, scalability, and reliability. Deceit servers are interchangeable and collectively provide the illusion of a single, large server machine to any clients of the Deceit service. Non-volatile replicas of each file are stored on a subset of the file servers. The user is able to set parameters on a file to achieve different levels of availability, performance, and one-copy serializability. Deceit also supports a file version control mechanism. In contrast with many recent DFS efforts, Deceit can behave like a plain Sun Network File System server and can be used by any NFS client without modifying any client software. The current Deceit prototype uses the ISIS Distributed Programming Environment for all communication and process group management, an approach that reduces system complexity and increases system robustness.

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N0014-87-C-8904, and under NASA-Ames Grant No. NAG 2-593. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

1 Introduction

This paper discusses the Deceit Distributed File System which is being developed at Cornell University. The research emphasis of the Deceit system is a more flexible file semantics to address the problems of scalability[1], efficiency, and reliability. Our premise is that it is valuable for the user to be able to adjust system semantics on a per file basis. Needed features may be employed without paying a penalty for unused features. The default behavior is equivalent to NFS.

Users select atypical features by overriding the default selections established by Deceit. Here are some of the choices that a user or system administrator might make:

- Availability—there are many techniques known for handling machine crashes and communication loss. Different solutions offer varying levels and types of resilience and cost. For example, data replication reduces the probability that the file will become unavailable for reading, but file updates become more expensive.
- Update propagation—there is a delay between when a client issues an update and when that update is apparent to other clients. Some applications may have constraints on this delay.
- Causality—constraints may exist between files to provide file consistency. By having the file system enforce such constraints, the user may augment or reduce the concurrency of updates and the success of file caching. For example, a run-time debugger may require that an executable file and its source file are consistent.
- Update stability—it may not be necessary for an update to be written to non-volatile storage or sent to all replicas immediately. Asynchronous update propagation can produce dramatic improvements in performance. Note that an update can be visible to all clients before it is has been delivered to all file replicas. The difference between update *propagation* and *stability* is clarified below.

The user can dynamically set *file parameters* to select the method that Deceit uses to provide the above properties. This paper describes the components of the Deceit file system which provide these file parameters and also gives a less detailed description of the

[1]In this discussion scalability will refer to the total number of file servers rather than the number of clients.

entire architecture. Some additional features that we plan to add in the near future are discussed also.

There are many existing distributed file systems including RNFS, Andrew, Sprite, Locus, Amoeba, SWALLOW, DFS, and Cedar (see bibliography for an extensive list of references). We are deeply indebted to these other distributed file systems for many of Deceit's design features. A full presentation of other distributed file systems is beyond the scope of this paper, and a partial presentation would inevitably be unfair, so we only compare Deceit to NFS in this paper.

This organization of this paper is as follows. Section 2 gives a general description of the architecture with the major assumptions underlying the Deceit design. Section 3 describes the file replication management protocols. Section 4 describes the parameters that a user can set on a file. Section 5 is a detailed description of the logical structure of Deceit. Section 6 describes some example scenarios where Deceit might be used, and Section 7 is the conclusion.

2 General Architecture

2.1 Contrast between NFS and Deceit

One way to understand the Deceit architecture is to contrast it to the NFS architecture[47, 48,46]. 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[2] is built by linking together the directory trees provided by the servers into a single tree. This linking is done separately at each client. Refer to Figure 1 for an example directory tree. 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.

Deceit and NFS use the same client/server communication protocol (i.e. the same transport and RPC interface), so a Deceit service appears to be a NFS file service to a client. As a result, Deceit provides a normal NFS name tree. All NFS operations are supported

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

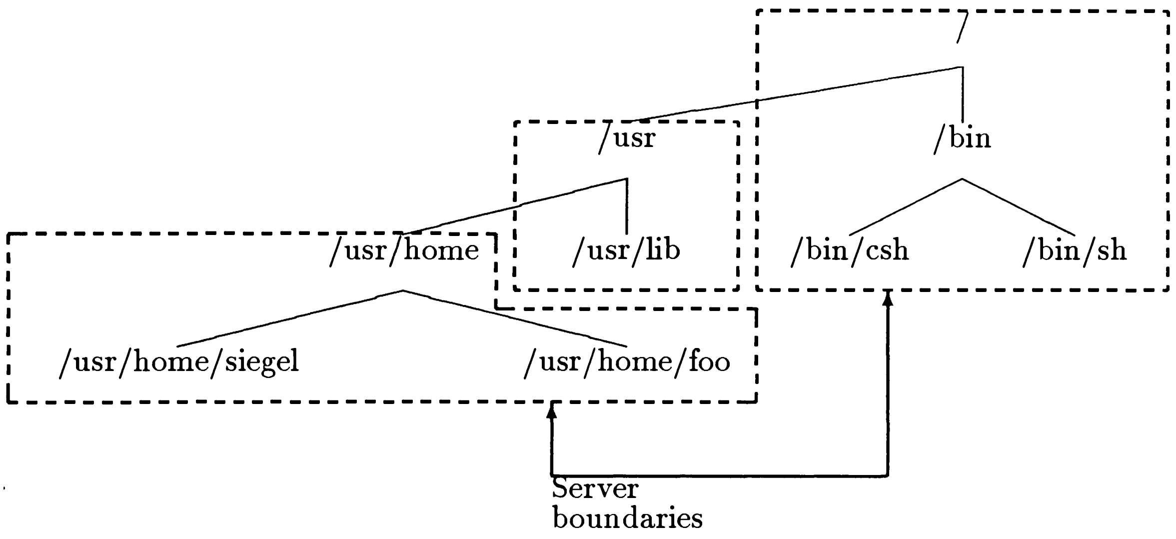
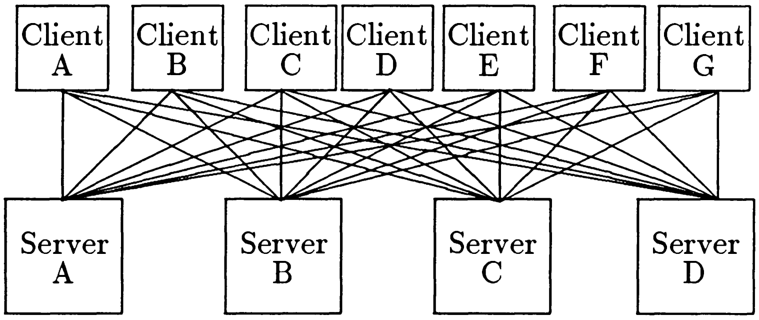


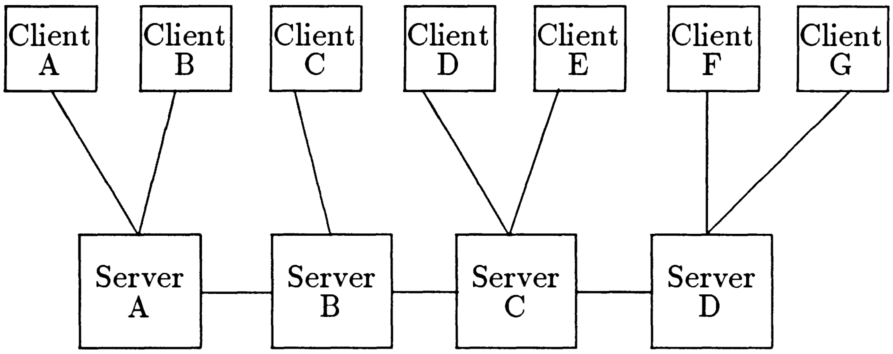
Figure 1: Example NFS directory tree

with no change to any client software. The *file handle* is an important component of the NFS protocol. A file handle is associated with each file or directory, and clients usually refer to files or directories by file handle. This type of handle for files is common in the file system literature[26,34,36]. Since Deceit uses the NFS protocol, Deceit provides file handles. These file handles are guaranteed to be unique and usable as long as a replica of the file exists.

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, the request is automatically forwarded to a server that has 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. A comparison between Deceit and normal NFS communications paths is provided in Figure 2. When one machine fails, Deceit clients can connect to another machine and continue operation; standard NFS client software does not provide this capability. Users may think of Deceit as a single, highly reliable and responsive server.



NFS Communication Paths



Deceit Communication Paths

Figure 2: communication paths

A second difference is that Deceit allows replication of files. 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 can specify a desired replication level and can provide explicit control over the placement of file replicas, if desired. Directories are handled similarly to files; they are stored on a subset of the available servers.

A third difference is that Deceit supports a file version control mechanism. A user may explicitly produce, manipulate, and delete specific versions of a file. Also, a user can inquire about the relationships between versions and ask Deceit to delete obsolete versions. This version control mechanism is "blended" into NFS semantics so that its use is optional.

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 directory versions, and provide other functions. These commands are discussed below.

2.2 Cells

In the above discussion, it was assumed that all clients could directly access any Deceit server, but this property is not necessarily true. Deceit servers can be subdivided into *cells* to prevent Deceit from being non-secure (and inefficient) in a very large implementation. Each cell is an independent instantiation of Deceit with distinct files and processes. Each cell maintains its own name space, and replication must be contained within a cell. A cell provides security and administrative boundaries. In our present implementation, cells correspond to ISIS site clusters. An example of Deceit cells is shown in Figure 3.

Access between cells is provided through a logical directory. There is a logical directory called the *global root directory.* It cannot be listed, as it implicitly contains the full machine names of every accessible Deceit server. Instead, it is 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. The global root directory is a subdirectory of "/priv." The Cornell cell acts as a client to the MIT cell. Mount and access restrictions are applied as with any client.

2.3 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, and typical operational behavior.

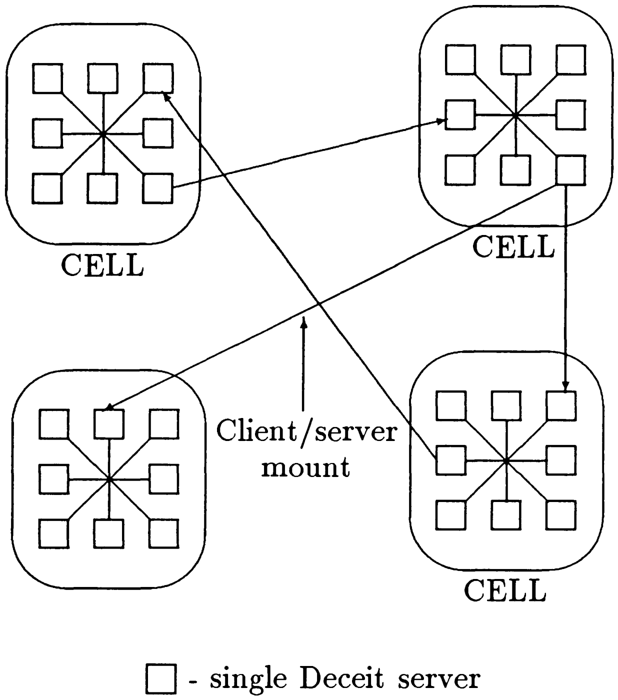


Figure 3: Example configuration of Deceit servers with cells

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. The servers are grouped into administrative subsets called *cells*, such that each cell is managed by a single centralized administration. Cells are assumed to be a small number of local area networks (e.g. 10-100 machines). Network communication is secure: messages are sent to the correct destination with a correct sources address, and messages can not be examined by machines which are not the intended receiver. Since all communication between servers is through the ISIS distributed system[2,3], all of the ISIS communication assumptions are present in Deceit.

Failure Assumptions

We assume that machines may crash without notification[3]; messages may be lost during transmission; and the network may experience long term communication partition.[4] All machines have roughly independent failure probabilities. Network partitions may be frequent. Within a cell, servers trust each other, but between cells there is minimal trust.

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[39,14,13,44].

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 *get attribute* (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.

2.4 Related Topics

The ISIS distributed system is used for crash and partition detection, communication primitives, and process group management[33,22]. 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. As a detailed discussion of ISIS would be a digression, the reader is referred to [4] for more information.

[3]For a more detailed discussion of machine and communication failure models, please refer to [3].

[4]Some readers may be aware that early versions of ISIS blocked during network partitions. As part of our work on Deceit, and other ISIS applications, this issue was reexamined. We expect a version of ISIS capable of surviving network partion to be available shortly.

Throughout this paper, the term "user" will be used to refer to the person or process who is initiating file system operations. In practice, some file system operations will need to be restricted beyond normal file security. For example, a system administrator may mandate that users can generate at most three replicas of a file. The separation of powers between users and system administrators is a fruitful area of research. In this paper, these distinctions will not be discussed.

3 Replication Management

Replication is one of the most important mechanisms in Deceit. If critical system components are distributed over several machines, then it is more likely that one of them will be unavailable at any time. Replicating components is an obvious solution. There are only a few file systems that allow files to be replicated among a set of servers[10,9,19,30,8,37].

Note, however, that caching is an important form of replication. Since caching is used in all file systems, then they all must support some form of replication. The Andrew File System[21] is a good example. Andrew supports caching on a client disk and in client memory. Deceit also supports client memory caching.

A desirable property of any replicated data system is *one-copy serializability*[5]. A file f exhibits one-copy serializability if the results of *reads* and *writes* to f are indistinguishable from the outcome of performing the same operations in a setting where there is only one replica of f. One-copy serializability is a useful property since it implies that the existence of multiple replicas is hidden at the user level.

3.1 Replica Generation

Associated with each Deceit file is a *minimum replica level* that can be defined and changed through a special command. If file f has a minimum replica level of r, then Deceit will insure that there are at least r non-volatile replicas of f as long as enough servers are available. To do so, new replicas may need to be generated. Associated with f is a server, t, called the *token holder* of f. The token holder is responsible for generating and deleting file replicas. Tokens will be discussed in more detail in Section 3.3. There are four ways that a replica can be generated:

[5] One-copy serializability is defined assuming all interprocess communication is through files.

1. The token holder t may lose contact with a replica. t counts the number of correct replies to an update broadcast for f. If the number of replies drops below r, then t will create new replicas. If there are no updates, replicas may become unavailable and later available without causing a new replica to be generated.

2. If the minimum replica level is increased, t will create new replicas.

3. A user may request the token holder t to create or delete a replica on a specific server with a special command. Users may inquire about the current location of all replicas for a file with another special command.

4. A server may request that a replica be generated in order to improve read performance.

Method 4 occurs as follows. If a client accesses file f through a server s which does not have a replica of f, then the operation is forwarded to a server which has a replica of f. As a background activity, a local non-volatile replica is generated on s to speed future reads and help ensure availability. In this manner, file migration is achieved with the replication mechanism. Each client slowly gathers its working set of files to the server to which it has connected. In some cases, the user may prefer that a replica is not automatically generated; this parameter may be set by the user.

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 TCP connection. Non-blocking I/O and careful buffer management allow the connection to run at high efficiency. The token holder delays updates during replica generation to prevent inconsistency.

Eventually, there may exist several unneeded replicas of a file. The token holder t will delete these extra replicas when an update occurs instead of updating them. They are deleted in least-recently-used order. The user may ask t to delete a replica with a special command.

Some existing DFSs allow files to be divided into segments for caching or replication. This option allows finer grain control over data movement and more efficient access to very large files. Unfortunately, it does not work well with the NFS protocol, and it greatly increases the complexity of the system. We have decided not to provide this option at the present time.

3.2 File Groups

For any file, f, there is an explicit *process group* of servers that need current information about f, which we will call the *file group* of f. A process group is a set of machines or processes; there must be a mechanism for broadcasting messages to all members and sending messages to individual members. Deceit represents each file group with an ISIS process group.

The file group for f contains all servers that have a replica of the file or have cached information about the file. This set is a superset of the replica holders, and it includes those servers which cache only timestamps or mode bits. The fundamental operation within a file group is *update distribution.* An update to f originates from a client and is given to its server. That server then broadcasts the update to all members of f's file group; no other servers receive this update for f. Refer to Figure 4 for a schematic description of update distribution. The concept of a file group is fundamental to the scalability of the entire system, since only the size of f's file group affects the speed of updates to f.

In Deceit, a server needs to join a file group before it is allowed to broadcast an update to, or have a replica of, that file. Joining a file group is an expensive operation and may require a global search to find a member of the group. This operation is one of the main obstacles to scaling Deceit to an arbitrary size. Deceit limits global search to within a Deceit cell to ameliorate this problem. As a result, file groups must stay within a single cell.

We believe that much of the complexity of distributed file systems arise in problems analogous to those found in group management. For example, a read/write quorum protocol can be viewed as a protocol for atomically broadcasting data updates to a group of replicas. The problem of locating a file replica by file handle is similar to the problem of locating a group member by group name. It would be interesting, but beyond the scope of our present discussion, to compare file systems in these terms.

3.3 Write-tokens

To coordinate access to replicated data, we use a *write-token protocol.* This protocol is based on one presented in [33,42]. A *write-token*[27,28] is associated with each file group. Only a server that holds the token is allowed to distribute updates to the corresponding

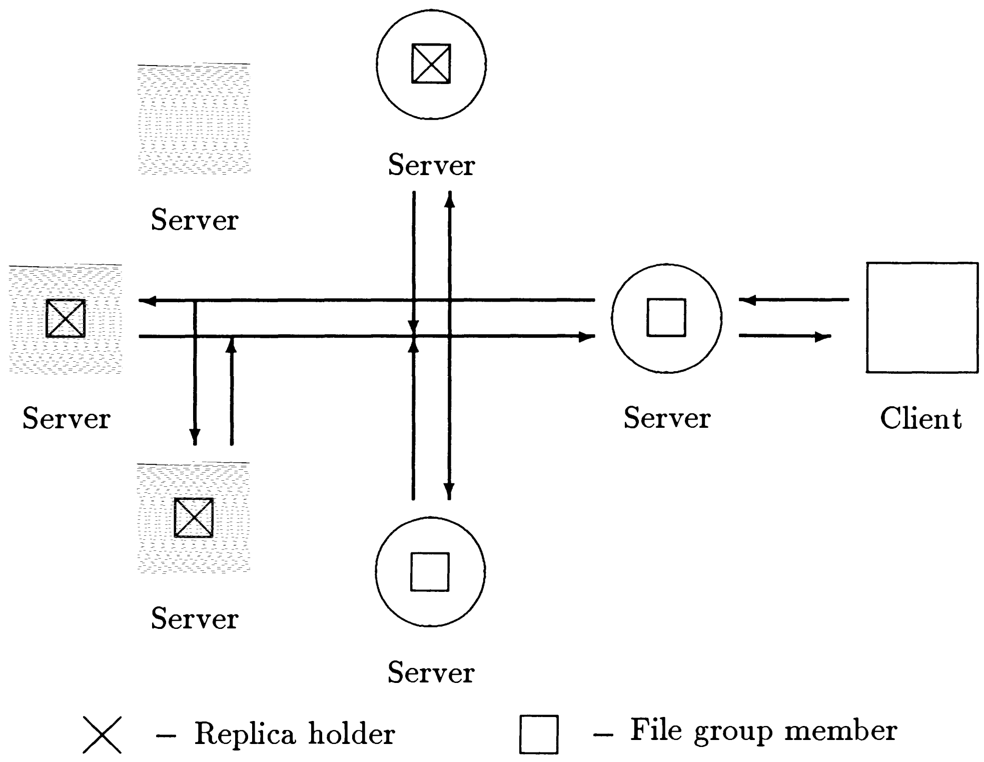


Figure 4: Update Distribution

file group. An update requires only one communication round[6] if the token is held. 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 those conditions most updates will require only one broadcast.

The token holder 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, the original client RPC that requested the update will return.

A server that lacks a token must acquire it before distributing an update for that file. Token acquisition requires one round, but it is only done for the first in a series of updates.

[6] A *communication round* is the distribution of a message to a set of processes. The collection of synchronous replies is included in the round.

To acquire a token, a server broadcasts a *token request* to that file group. The server that holds the token broadcasts a *token pass* in response. It is necessary for correctness that the updates arrive in identical order at all servers regardless of token movement.

There are several optimizations to this protocol. One optimization is to broadcast an *update* in the same message with a *token request*, replica holders execute those updates upon receiving the corresponding *token pass*. Another optimization is to pass an update to the current token holder instead of requesting the token if it is likely that there will be only one update; for example, a small file that is overwritten in a single update will probably not be updated again soon. Deceit currently uses neither of these optimizations.

3.4 Global One-copy Serializability

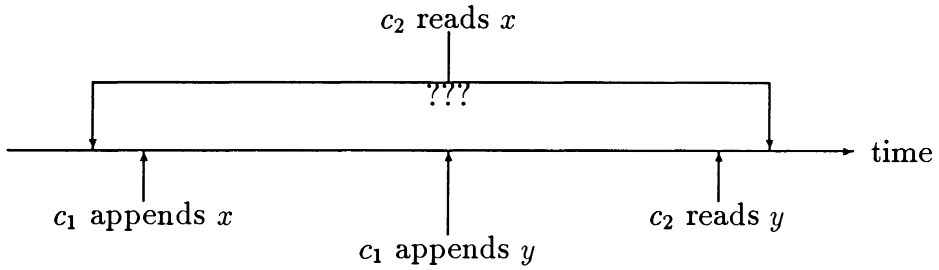
The *write-token* scheme described in Section 3.3 is sufficient to achieve one-copy serializability in a file system containing only one file, but a real file system will require a stronger mechanism. *Global one-copy serializability* is defined as the property that clients should observe one-copy serializability on the whole file system rather than simply on individual files. A related property is *real-time consistency*; if one user writes a file and calls a friend on the phone, the friend should be able to observe the update within a bounded delay.

Global one-copy serializability is stronger than simple one-copy serializability as is shown in Figure 5. In this example, files x and y are initially empty. Client c_1 appends to x and then appends to y. Concurrently, client c_2 successfully reads from y and then observes that x is empty. This result is impossible if there is only one replica of x and y. Yet x and y *separately* exhibit one-copy serializability.

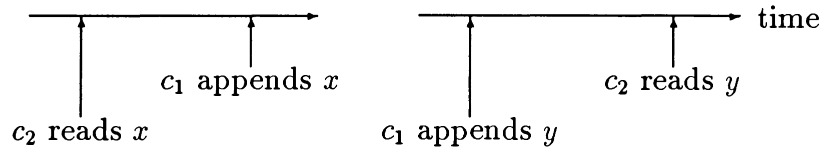
An obvious solution is to wait for the update to complete at every member of the file group before allowing a *write* call to return to a client. Unfortunately, this can lead to bad performance, particularly in the case where a replicated file is being written with a stream of small updates. A more efficient mechanism that allow updates to complete concurrently is called for.

Deceit provides global one-copy serializability with a *stability notification* mechanism. Before a file can be modified, all members of the file group are notified that the file is *unstable*. All available[7] replicas must be so notified before any updates can occur (the failure of the

[7] A replica at server b is *available* to a if a can communicate with b. ISIS provides a clean notion of availability since failure detection is coordinated with communication.



Time History of Operations with Files x and y



Separate Time Histories of x and y

Figure 5: Illustration of One-copy Serializability Example

token holder during stability notification is discussed in Section 3.5). After stability notification, all file reads and inquiries are forwarded to the token holder. Only the token holder's replica needs to be updated before a write can return to a client. One-copy serializability is guaranteed because, in effect, the token holder's replica is now the "primary" replica. After a short period of no write activity, the token holder notifies all other members of the group that the file is *stable* again. Table 1 gives a short summary of the sequence of events required in a normal update. Stability notification is normally invisible to applications, and its main effect is on performance and update visibility to clients.

The main benefit of stability notification is that updates become visible to all clients simultaneously.[8] On the other hand, an overhead is incurred at the beginning and end

[8]In a distributed system, simultaneous events may not appear to happen at the same physical time since communication delay introduces uncertainty.

	Precondition	Action
1	token is not held	acquire token
2	replicas are not marked as *unstable*	mark replicas as *unstable*
3	true	distributed update
4	failure detected	count update replies
5	insufficient replicas	generate new replicas
6	period of no write activity	mark replicas as *stable*

Table 1: Typical Sequence of Events in an Update

of a stream of updates. This overhead can be expensive if updates are short and rare. Also, reads that are concurrent with updates are more expensive. By default, Deceit uses stability notification, but the client can specify that stability notification is not used.

3.5 Crash and Partition Failures

The algorithm presented in Sections 3.1 to 3.4 must be resilient to failure. The mechanisms that Deceit uses for this purpose are presented below.

Histories and Version Pairs

Associated implicitly with each replica of file f is an update history $f.h$. An update history is a list of all updates to the file and which server issued these updates. History $f.h$ is an *ancestor* of history $f.h'$ if $f.h$ is a prefix of $f.h'$. The histories that all replicas of a file pass through form a tree under the ancestor relation; this tree is called the *history tree.* Two histories are incomparable if neither is an ancestor of the other.

Deceit does not explicitly store the full history of a replica. Instead, Deceit maintains a one-to-one mapping from histories to integer pairs (v_1, v_2) where v_1 is the *major version number*, and v_2 is the *subversion number*[9]. v_2 is incremented on every update, and v_1 is changed to a new unique number every time there is a potential branch in the history tree. These branch points are recorded with a replica so that version number pairs can be

[9]In the literature, this value is often called an *update counter.*

compared as if the histories that they represent were available. For example, the relation $(v_1 = v_1' \wedge v_2 < v_2') \Rightarrow (f.h$ is an ancestor of $f.h')$ always holds.

A version pair is stored with each write token. The token version pair can be compared to a replica version pair to quickly decide if a replica has received every update through that token. This version pair is available to the user through a special command so that the user can determine if a file has been modified.

Token Generation

If a client wishes to update a file, and no write token is available for the specified version, a new token will be generated. Assume that the file being updated has major version number v_1, and a replica is available with version pair (v_1, v_2). Server s can generate a new token by picking a globally unique major version number v_1' and building a token with version pair (v_1', v_2), then s stores v_1 with the new token. Replicas corresponding to the new token are generated by copying the original replica.

Generating a new token is more than simply generating a new version pair. Every file replica is associated with only one token. The new token represents a distinct new file with a distinct set of replicas. After the new token is generated, enough replicas are generated to satisfy the *minimum replica level* constraint. File data is drawn from the existing available replica.

It may be necessary to constrain when a token can be generated. Deceit provides file parameters settings that provide this capability. There are three options. The first option is to totally inhibit the generation of new write-tokens. This option has the advantage that a server can always write to a file after it has acquired the write-token, but it is easy to suffer long term loss of file availability. The second option is to allow a server to generate or use a token only if the majority of the replicas are available. A token becomes disabled if the majority of the replicas becomes unavailable. This option provides relatively high availability, and multiple versions can be generated only during transitional periods. On the other hand, it is more difficult to implement, and write availability may be lost in the middle of a stream of updates. The third option is to not restrict token generation at all. Deceit uses this second option as the default.

Restricting updates to the majority partition requires a mechanism for counting the number of available replicas. Replicas are normally counted by counting the correct replies to an update broadcast. All replica generation must be accomplished through the token holder, so that the token holder always has an upper bound on the total number of replicas. For purposes of computing a majority, the total number of replicas is taken to be the maximum of the *minimum replica level* and the upper bound on the number of replicas. For a server without access to the token, the total number of replicas is assumed to be the minimum replica level; the number of available replicas is determined by broadcasting an inquiry to the file group.

Version Control System

During a partition event, multiple file versions can be generated. It follows that Deceit must be capable of maintaining distinct versions which are distinguished only by different values for the major version number, v_1. The facility by which this is accomplished may 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, major version 3 of "foo" can be referred to as "foo;3." By using this form of file name, specific versions can be created[10], modified, and deleted. By using an unqualified filename, the user automatically requests the most recent available version. A directory entry actually uses the unqualified filename, so creating a new file version does require an update to a directory. The system behaves similarly to the VAX/VMS[11][11] 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.

Local Non-volatile Storage

Several types of information must be kept in non-volatile storage to allow recovery from a crash. Each server stores all file data for its replicas. This data includes: the actual data of the file, the replica state, and the version pair. Additionally, each server stores all state information relating to each token that is held. Also, each server stores a non-volatile copy of the map between file handles and local file names. Some of a server's non-volatile storage is updated immediately when values change, and some of it is written asynchronously, depending on safety.

[10]Deceit selects major version numbers carefully to insure global uniqueness. Users must be careful when creating new versions during a partition to preserve uniqueness.

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

3.6 Crash Scenarios

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

Non-token Replica Crash

When a server s recovers from a crash, it contacts the token holder for each file f such that s has a replica but no token for f. Each token carries the version pair that replicas should have if they are up to date. If s finds that it has an obsolete replica of f, s destroys it. Since the history for the replica of s is a prefix of the history associated with the token, no update will be lost.

Token Crash

Token loss is detected when a server attempts to contact a token holder during the course of normal read or update operations. Let us assume that server s needs to distribute an update for a file, but it can not contact the current token holder. Subject to token generation constraints, s can generate a new token. Since s now holds the token for a version of the file, s can complete the original operation.

Assume that s could not contact the old token holder s' because s' had crashed. When s' recovers, it will be notified about the creation of the new version during its recovery protocol. s' will note that the new version is a direct descendent of the old version and destroy the old version and all of its replicas.

Partition

Now consider the scenario where there was a network partition, but no updates were issued to the file in the partition with the token. *Read* access on the token holder side continues normally, since it is difficult distinguish between this scenario and the case where the other replicas simply crashed. *Write* access in the partition which does not contain the old token may cause a new token to be generated. When the partition is resolved, the old token holder will be notified. It will appear to the clients as if the token had actually been moved, and the updates were propagated very slowly to some servers.

The hard case is when a partition occurs and updates are issued to the file on both sides concurrently. In this case both of the incomparable versions of the file are kept, and a notification is logged into a well known file. It is the responsibility of the user to resolve such conflicts. By allowing the user to resolve incomparable versions, the semantics of the file may be used for resolution [5,52,20]. Both versions are made available to the user and may be edited, modified, or deleted independently. Since concurrent updates are assumed to be rare, this case should occur very rarely.

Stability Notification in the Presence of Failure

If the token holder t for a file f loses contact with some of f's replicas during an update distribution, those replicas might be left in an inconsistent state. Stability notification is used to detect this case. Before an update is distributed, all available replicas are marked as *unstable*. Therefore, if replica states are inconsistent, then all inconsistent replicas will be marked as *unstable*.

Inconsistency is detected when a *read* is given to a server s which has an *unstable* replica of f, and s is unable to contact t. In order to respond to a *read*, s must locate a *stable* replica. s produces a *stable* replica by broadcasting to f's file group to determine the state of all available replicas. If there is a *stable* replica at server s', the operation is forwarded to s'. If no replica is marked as *stable*, s forces the most up to date replica to be *stable*, and all obsolete replicas are destroyed.

Disastrous Failure

Despite all of these precautions, with a suitably pathological sequence of crashes and recoveries, it is still possible to produce non-one-copy serializability. For example, if an obsolete file replica recovers and all other replicas simultaneously crash, the file will appear to go back in time. We could solve this problem by inhibiting token generation and by consulting the token holder during every operation, but this solution would destroy most of the benefit of replication.

4 File Semantics

Deceit associates the following semantic parameters with each file:

1. Minimum Replica Level - the minimum number of valid replicas that must be maintained. For example, a minimum replica level of 3 would force Deceit to maintain a valid replica on at least 3 separate servers. By default, the value is 1.

2. Write Safety Level - the number of replica servers that must reply to an update before a *write* RPC returns to a client. A value of 0 produces asynchronous unsafe writes; a value greater than or equal to the number of available replicas produces slow and fully synchronous writes. By default, the value is 1.

3. Stability notification - specifies whether stability notification is to be used. Stability notification guarantees global one-copy serializability and real-time update propagation, but there is a performance cost. The default is to use stability notification.

4. File migration - should Deceit automatically attempt to create a non-volatile replica of file f on a server that receives requests from a client for f. For some applications, it may be bad to automatically generate local replicas. For example, for a very large data file, generating a local replica may consume too much disk space. The default is that file migration not be used.

5. Write Availability Level - determine when Deceit can generate a new write-token if a token has been lost. If this flag is set to "high", then a token may be generated whenever one is needed. A high availability means it is likely that multiple file versions will result due to a partition. A value of "medium" allows a new token to be generated by server s only when s can contact a majority of the replicas, and a token is disabled if fewer than the majority is available. As a result, some replicas may occasionally be "read only," but multiple file versions will occur less frequently. A value of "low" prevents the production of additional tokens. Loss of file write access may be frequent and long term, but there is no chance of generation of multiple versions. The default value is "medium."

5 System Components

The Deceit server consists of two components as shown in Figure 6. The first component is a *distributed reliable segment server.* The segment server provides a simple, flat, reliable distributed file service with no user level security or user specified names. There is no notion of directories or links in the segment server. The segment server implements all of the update, replication, and versioning protocols, and it is the layer where file parameters exist. On top of the segment server is a full NFS file service which uses the segment server for storage and communication, called the *NFS file service envelope.*

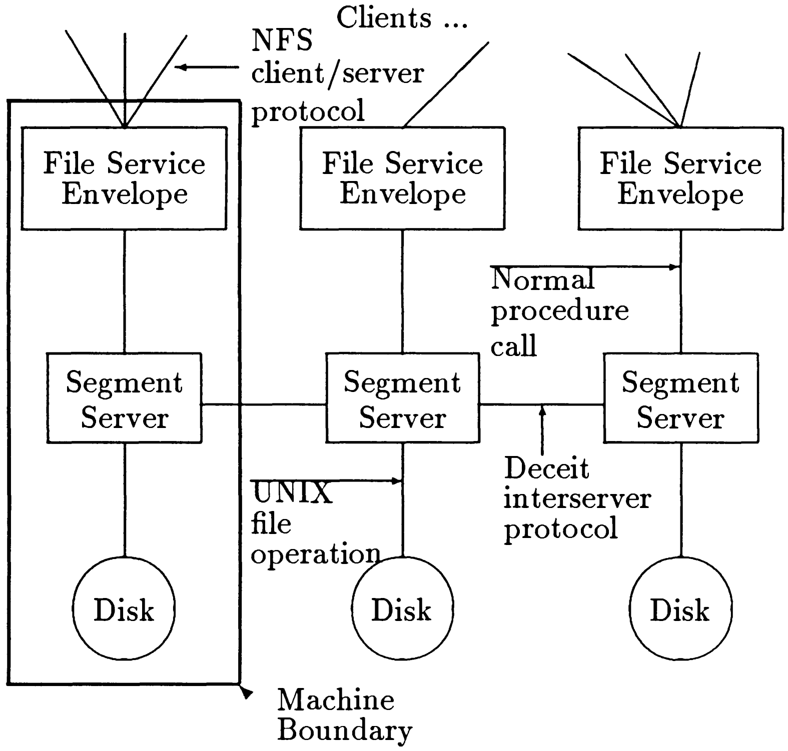


Figure 6: Expanded view of Deceit architecture

Deceit does not directly address most security issues. It is assumed that communication between instances of the segment server is secure (e.g. encrypted or physically secure). Also, the local files used for storage by the segment server are inaccessible to unauthorized users. Client/server communication is secured, and client authentication is provided using DES encryption in the NFS interface. It is beyond the scope of this discussion to provide a detailed description of these mechanisms.

5.1 Segment Server

A *segment* contains an array of bytes that can be indexed by an offset. Associated with a segment is the data in the segment, the values for each of the semantic parameters, a version number pair, a process group, and read and write timestamps. The interface to the segment server consists of five normal procedure calls: *create*, *delete*, *read*, *write*, and *setparam*. *Create* has no arguments and simply returns a handle for a new segment of zero length. *Delete* takes a segment handle and deletes all storage allocated for it. *Read* reads a portion of a segment. *Write* modifies a segment by replacing, appending, or truncating data in the segment. *Setparam* is used to specify semantic parameters on a segment.

It is valuable to have a form of *serial transaction*[12] when accessing a segment. Many distributed files systems, particularly earlier ones, had strong mechanisms for executing an atomic transaction. For example, DFS[45] had an extensive transaction mechanism which involved locking and intention lists. SWALLOW[50,49] used a system of virtual time stamps to provide serializability. The Alpine File System[6] used log files to provide recoverability. Later distributed files systems had weaker atomicity guarantees in order to provide better performance.

To help provide a transactional capability, Deceit employs the version number pair. A *read* call not only returns data, but it also returns the version pair associated with that data. A *write* also returns the version pair of the segment after the write has completed. A *write* call can also have a version pair as a parameter; in this case the *write* will succeed only if the version pair of the segment matches the version pair in the call when the data is actually updated, otherwise an error will be returned. Using this feature, a limited type of serial transaction can be achieved.

This notion of version pairs can be used to implement an optimistic concurrency control mechanism. A *read* implicitly begins a transaction with the establishment of a specific version pair for the segment. Writes are then issued with that version pair. A transaction

[12]An *atomic* transaction two properties: *recoverability* and *serializability.* Recoverability means that the transaction completes or fails entirely. Serializability means that the transactions exhibit behavior consistent with some total ordering. A serial transaction is a transaction that only provides serializability.

is implicitly completed when the last write completes successfully. A write which returns an error due to the use of the wrong version pair is similar to a transaction which has been aborted due to the detection of non-serial behavior. Unfortunately, there is no way to back out of previous updates. The application which attempted the write can restart with the original read or abandon the attempt entirely. In effect, transactions have been transformed into light weight tasks which run at the level above the segment server.

A good example of this behavior is the addition of an entry to a directory in the Deceit file system. The directory is read, and a position is selected in the directory for adding the entry; this may overwrite an existing dead entry or append a new entry. Then, an update is given to the segment server with the version pair returned by the original read. If a version pair conflict occurs, the whole operation is restarted.

5.2 NFS File Service Envelope

The full file service is built on top of the reliable segment server. The principle is that every file, directory, or soft link is mapped into a unique segment. All NFS operations are mapped into creates, deletes, reads, and writes on segments. The UNIX kernel does a similar transformation when it transforms user file operations into disk operations. The full file service inherits the distributed nature of the segment server to provide a full distributed file system. Although the NFS envelope implementation is a large piece of software, it is totally independent of the underlying implementation of the segment service. In principle, it will never need to be changed despite radical changes in the segment server protocols.

Links and Garbage Collection

Since the segment server does not have a notion of links, it is the responsibility of the NFS envelope to decide when a file is no longer accessible, so the storage for it can be deallocated. Since both directories and files may have multiple versions, and since creation of new links may be hidden behind partitions, it can be very difficult to decide when to delete the segment for a file [29]. A normal NFS system has the advantage of centralized control and single file versions, so a simple link count suffices (however the link counts can be corrupted by an ill timed crash.)

We see two solutions to the problem. The first solution is to extend the concept of a link count. The link count would correspond to the total number of link copies, where every replica of every version of a directory referring to the file is counted once. Refer to Figure 7 for an example of this type of link count. The link count is stored in the segment as normal data. To maintain safety, it would be impossible to add a link to a file unless the file was write available. Unfortunately, it would also be unsafe to create new replicas or versions of directories unless each file in the directory was also write available. Another severe disadvantage is that when the link count becomes corrupted, it is extremely expensive (or impossible) to recalculate.

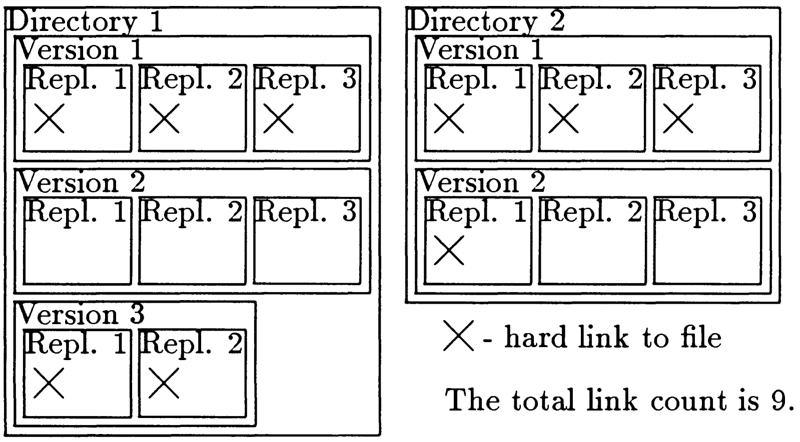


Figure 7: Example of Link Count Computation

The solution we chose is more complex. An *uplink list* of directory file handles is stored with each file. The NFS envelope attempts to maintain the property that if file f is in directory d, then d is in the uplink list of some version of f. When a hard link is made to f in directory d, d is added to the uplink list of all versions of f which can be updated *at that time.* Deceit also keeps a standard hard link count with f, but it is only considered to be a hint. When the link count goes to zero, the NFS envelope checks every available version of every directory in the uplink list. If none have a link to the file, the segment is deallocated; otherwise, the link count is corrected.

Our solution has several drawbacks. A file f may neither be moved nor additional links made unless the uplink list of some version of f can be safely modified. Also, a server s may add an uplink to f, but another server s' may never see that uplink if s' can only contact a disjoint set of versions. As a result, s' may prematurely attempt to deallocate f.

Another drawback is that if the link count of f is corrupted so that it is too large, f may never be garbage collected. Finally, when a file is moved, two directories, a link count, and an uplink list must be modified in some safe order. Garbage collection is discussed again briefly in Section 7.

5.3 Client Agents

The *agent* is a simple but important component of Deceit. The agent is the client software which interfaces between the user process and the NFS protocol. Currently, the agent runs in the kernel, but the agent can be in several possible locations. Refer to Figure 8. These different configurations provide widely differing performance.

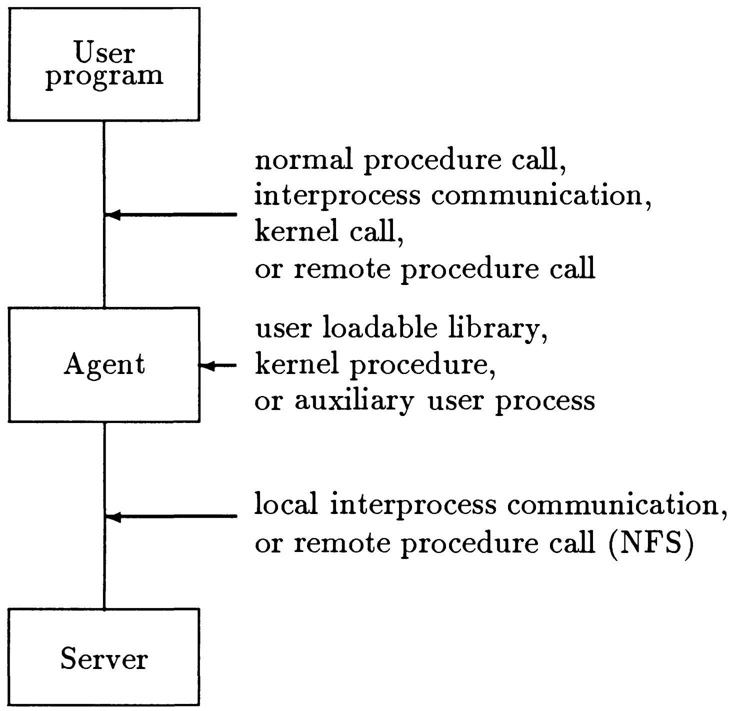


Figure 8: Example Agent/Server Configurations

The agent satisfies two primary functions. First, the agent provides caching. The agent caches file and directory data as well as information specific to the client/server protocol such as NFS file handles and server information. Another agent function in Deceit is *failover*. When one server fails, the agent must select another to continue operation. This second capability requires an extension to the NFS protocol.

A third optional agent function is using an *access shortcut*. Normally, a server forwards a requests for which the server does not have a replica. It is more efficient for the agent to cache file locations and directly communicate with the correct servers. This capability requires an extension to the NFS protocol.

Deceit currently uses the standard NFS client software provided in the Sun operating system. This software does not provide failover or shortcuts. A new agent is being written which will run as an auxiliary user process, and it will provide full functionality. An agent which can be loaded as a user library and directly issues NFS RPCs is planned, and this agent should greatly improve file performance.

A user library that acts as an agent is not easily implemented for several reasons. First, since the NFS protocol requires file handles and there is no way to easily extract file handles from the SunOS kernel, each user process will have to go through the full mount protocol to get file handles. Also, since files will be cached in the user process virtual memory space, processes will have to use some type of coordination protocol to share cached files.

5.4 ISIS

We made the decision to use ISIS after some consideration; an early version of Deceit did not use ISIS at all. When we used ISIS, several issues arose:

- ISIS requires a separate configuration and installation phase in addition to the one required for Deceit. This requirement was an inconvenience during development, and it will continue to be a problem in the future. A stand-alone package requiring little or no additional configuration information would be preferable.
- Deceit exposed several performance and development problems in ISIS. Group membership change, high volume state transfer, as well as other operations were too expensive. Some ISIS features, such as partition tolerance, were underdeveloped when Deceit began development. These issues are being addressed. Future versions of ISIS should allow Deceit to have satisfactory performance.

- One particular problem was the huge number of process groups that could be generated by Deceit. In the current implementation of ISIS, process groups are an expensive resource. More specifically, ISIS does not efficiently support more than 100–1000 process groups. Future versions of ISIS should support larger numbers of process groups. Future versions of Deceit will be more careful with generating and deleting process groups.

- ISIS saved a large amount of development time. We started to use ISIS after realizing that we were reimplementing much of the ISIS functionality in Deceit. ISIS also provides useful debugging primitives. We estimate that at least 6 months of development was saved.

6 Scenarios

The previous sections provided a great deal of detail about Deceit without much discussion about applications. To show how Deceit could be used to solve real problems, two important application scenarios are listed below. For each scenario, there is a short description of the scenario, followed by a description of how Deceit could be used efficiently in this application.

6.1 Academic Public Workstation Environment

This environment is characterized by a large number of small, inexpensive, and unreliable machines. Administrative control is often poor. Users spend the bulk of their time editing or compiling. Files tend to be small, and their physical location is relatively unimportant, but high availability is valuable.

This scenario is the easiest to solve since Deceit is being developed and tuned in this environment. All of the semantic parameter defaults should be adequate. Users will typically want to set the replication level to 2 or 3 on important source and text files; other files can be regenerated if necessary. The system administrator should set the replication level to be 2 or 3 on all important system directories, binaries, and libraries. Adding new servers is simply a matter of configuring ISIS to run on the server, and executing the Deceit server daemon. Files can be moved transparently from one server to another by the system administrator at any time to provide better disk balancing.

6.2 Data Collection and Dispersion

A large class of applications requires bulk data movement and manipulation. For example, NASA collects huge amounts of data at several remote stations which is processed in a central computing facility. A product development team uses large detailed specifications to drive simulations which can be at a distant location. This environment is characterized by a small number of large machines with large numbers of peripheral machines attached to them. Extremely large files are common. Users collect large quantities of data on some machines and analyze it at other machines. Since file sizes often are at the limits of disk space, controlling the location of the data is necessary. There may be large geographical distances within the system.

For a very large data file, the user can turn off automatic localization to prevent uncontrolled generation of file replicas. Also the minimum replica level should be 1 until the file has reached its final destination, and then it may be set to 2 to provide a single backup. Since data versioning may lead to version conflicts, the write availability level will probably need to be "medium" or even "low." Data files can be quickly copied from one server to another using the blast file transfer mechanism in Deceit by manually forcing the creation of a replica on the target server and then deleting the replica on the source server. At any time during the manipulation of the data location, the file data is available for reading and writing via any server.

7 Conclusions

We believe Deceit provides enough flexibility so that most applications can have acceptable performance and availability. All the basic features necessary for a full distributed file system with replication have been provided. Deceit performance is not understood if the environment does not satisfy the operational assumptions in Section 2.3. In such cases, we may be required to add new operational modes be added to Deceit.

A version of Deceit exists and is used at Cornell, although it is not yet in general use. Except for inter-cell communication, all of the features described above are implemented. Development is still at an early stage, and we expect fundamental architectural changes as our experience continues. Performance measures would be premature at this stage of our effort.

Two serious problems still need to be addressed. The first is *file contention bottlenecks*. Certain files and directories such as the root directory will be accessed very frequently by all servers. It is fortunate that these files tend to have read only access. It may be valuable to have special file modes which are optimized for this combination of properties[38,24].

Another problem is the use of links and directories as discussed in Section 5.2. The current solution is unsatisfactory, so we are looking for other solutions. Hopefully, a solution to this problem may also offer a solution to the root directory contention problem. One possibility that we are investigating is the use of file uplinks to allow non-volatile directories to be discarded.

There are some performance problems with the process group management protocol. Group joins are expensive, and broadcasts are more expensive than need to be. Also, using a huge number of ISIS groups has a unacceptable effect of ISIS performance. Since Deceit uses process groups in a restricted and well defined way, it is inefficient to use general ISIS groups for each file group. Several methods to improve performance are being investigated.

Acknowledgements

The authors would like to thank Patrick Stephenson and Robert Cooper for their careful reading of early versions of this manuscript.

References

[1] Gregory R. Andrews, Richard D. Schlichting, Roger Hayes, and Titus D. M. Purdin. The Design of the Saguaro Distributed Operating System. In *IEEE Transactions of Software Engineering*, pages 104–118. IEEE, January 1987. Vol. SE-13 and No. 1.

[2] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.* ACM, November 1987. Order no. 534870.

[3] Kenneth Birman and Thomas Joseph. Reliable Communication in the Presence of Failures. In *ACM Transactions of Computer Systems.* Association for Computing Machinery, February 1987. Vol. 5 and No. 1.

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

[5] Pearl Brereton. Detection and Resolution of Inconsistencies Among Distributed Replication of Files. In *Operating Systems Review*, pages 10–15. Association for Computer Machinery, January 1983. Vol. 17 and No. 1.

[6] Mark R. Brown, Karen N. Kolling, and Edward A. Taft. The Alpine File System. In *ACM Transactions on Computer Systems*, pages 261–293. Association for Computer Machinery, November 1985. Vol. 3 and No. 4.

[7] Luis Felipe Cabrera. QuickSilver Distributed File Services: An Architecture for Horizontal Growth. In *1988 IEEE 2nd Conference On Computer Workstations*, pages 23–37. IEEE, March 1988. 88CH2441-4.

[8] John L. Carroll, Darrell D. E. Long, and Jehan-Francois Paris. Block-Level Consistency of Replicated Files. In *The 7th International Conference on Distributed Computing Systems*, pages 146–153. IEEE, September 1987. Order no. 801.

[9] Dean Daniels and Alfred Z. Spector. An Algorithm for Replicated Directories. In *Operating Systems Review*, pages 24–43. Association for Computer Machinery, January 1986. Vol. 20 and No. 1.

[10] Danco Davcev and Walter A. Burkhard. Consistency and Recovery Control for Replicated Files. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 87–96. ACM, December 1985. Order no. 534850.

[11] Digital Equipment Corportion, Maynard, Massachusetts. *VAX/VMS - Introduction to VAX/VMS*, vax/vms version 4.0 edition, September 1984. Order No. AA-Y500A-TE.

[12] Jeremy Dion. The Cambridge File Server. In *Operating Systems Review*, pages 26–35. Association for Computer Machinery, October 1980. Vol. 14 and No. 4.

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

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

[15] Marek Fridrich and W. Older. The FELIX File Server. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 37–44. ACM, December 1981. Order no. 534810.

[16] David K. Gifford, Roger M. Needham, and Michael D. Schroeder. The Cedar File System. In *Communications of the ACM*, pages 288–298. Association for Computer Machinery, March 1988. Vol. 31 and No. 3.

[17] Anna Hac. Distributed File Systems - A Survey. In *Operating Systems Review*, pages 15–18. Association for Computer Machinery, January 1985. Vol. 10 and No. 1.

[18] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162. ACM, November 1987. Order no. 534870.

[19] Maurice Herlihy. Comparing How Atomicity Mechanisms Support Replication. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 102–110. ACM, August 1985. Order no. 536850.

[20] 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. Order no. 549880.

[21] John H. Howard, Michael L. Kazar, Sherri G. Menees, Davic A. Nichols, M. Satyanarayanan, Robert N Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. In *ACM Transactions of Computer Systems.* Association for Computing Machinery, February 1988. Vol. 6 and No. 1.

[22] Thomas A. Joseph and Kenneth P. Birman. Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. In *ACM Transactions of Computer Systems.* Association for Computing Machinery, February 1986. Vol. 4 and No. 1.

[23] Nancy P. Kronenberg, Henry Levy, and William D. Strecker. VAXclusters: A Closely-Coupled Distributed System. In *ACM Transactions on Computer Systems*, pages 130–146. Association for Computer Machinery, May 1986. Vol. 4 and No. 2.

[24] Keith A. Lantz, Judy L. Edighoffer, and Bruce L. Hitson. Towards a Universal Directory Service. In *Operating Systems Review*, pages 43–53. Association for Computer Machinery, April 1986. Vol. 20 and No. 2.

[25] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File Access Performance of Diskless Workstations. In *ACM Transactions on Computer Systems*, pages 238–268. Association for Computer Machinery, August 1986. Vol. 4 and No. 3.

[26] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, and Paul H. Levine. UIDs as Internal Names in a Distributed File System. In *Proceedings of the First Symposium on Principles of Distributed Computing*, pages 34–41. ACM, August 1982.

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

[28] 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.

[29] Barbara Liskov and Rivka Ladin. Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 29–39. ACM, August 1986. Order no. 536860.

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

[31] James G. Mitchell. A Comparison of Two Network-Based File Servers. In *Communications of the ACM*, pages 233–245. Association for Computer Machinery, April 1982. Vol. 25 and No. 4.

[32] Sape J. Mullender. A Distributed File Service Based on Optimistic Concurrency Control. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 51–62. ACM, December 1985. Order no. 534850.

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

[34] Roger M. Needham. Adding Capability Access to Conventional File Servers. In *Operating Systems Review*, pages 3–4. Association for Computer Machinery, January 1979. Vol. 13 and No. 1.

[35] Michael N. Nelson, Brent B. Welch, and John K. Ouserthout. Caching in the Sprite Network File System. In *ACM Transactions of Computer Systems.* Association for Computing Machinery, February 1988. Vol. 6 and No. 1.

[36] D. M. Nessett. Identifier Protection in a Distributed Operating System. In *Operating Systems Review*, pages 26–31. Association for Computer Machinery, January 1982. Vol. 16 and No. 1.

[37] Jerre D. Noe and Agnes Andreassian. Effectiveness of Replication in Distributed Computer Networks. In *The 7th International Conference on Distributed Computing Systems*, pages 508–513. IEEE, September 1987. Order no. 801.

[38] John Ousterhout and Fred Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. In *Operating Systems Review*, pages 11–28. Association for Computer Machinery, January 1989. Vol. 23 and No. 1.

[39] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24. ACM, December 1985. Order no. 534850.

[40] Loretta Guarino Reid and Philip L. Karlton. A File System Supporting Cooperation between Programs. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 20–29. ACM, October 1983. Order no. 534830.

[41] M. Satyanarayanan. A Survey of Distributed File Systems. Technical report, Department of Computer Science at Carnegie Mellon University, February 1989. CMU-CS-89-116.

[42] Frank Bernhard Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems.* PhD thesis, Cornell University, August 1988.

[43] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A Caching File System For a Programmer's Workstation. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 25–34. ACM, December 1985. Order no. 534850.

[44] Carl Staelin. File Access Patterns. Technical Report CS-TR-179-88, Princeton University, September 1988.

[45] H. Sturgis, J. Mitchell, and J. Israel. Issues in the Design and Use of a Distributed File System. In *Operating Systems Review*, pages 55–69. Association for Computer Machinery, July 1980. Vol. 14 and No. 3.

[46] sun microsystems, Mountain View CA. *Network File System Protocol Specification*, February 1986.

[47] sun microsystems, Mountain View CA. *Remote Procedure Call Programming Guide*, February 1986.

[48] sun microsystems, Mountain View CA. *Remote Procedure Call Protocol Specification*, February 1986.

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

[50] Liba Svobodova. A Reliable Object-Oriented Data Repository for a Distributed Computer System. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 47–58. ACM, December 1981. Order no. 534810.

[51] Andrew S. Tanenbaum and Sape Mullender. An Overview of the Amoeba Distributed Operating System. In *Operating Systems Review*, pages 51–64. Association for Computer Machinery, July 1981. Vol. 15 and No. 3.

[52] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–70. ACM, October 1983. Order no. 534830.