To the Graduate Council:

I am submitting herewith a thesis written by Raymundo Diaz Rodriguez entitled "The Design and Implementation of a User-Oriented Replicated File System for Unix." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

David C. Mutchler, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Raymunda Díaz

Date June/30/1989

THE DESIGN AND IMPLEMENTATION OF A USER-ORIENTED REPLICATED FILE SYSTEM FOR UNIX

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Raymundo Diaz Rodriguez

August 1989

to my wife and parents

ACKNOWLEDGMENTS

During the development of this thesis, a number of scholars have assisted me. In particular I am indebted to Professor David Mutchler, who served as my advisor, for his patience and constructive criticism. With my advisor's, and Professors Bruce MacLennan and David Straight, consultation, I was able to achieve results otherwise impossible. Whatever errors remain in my work, however, are solely my responsibility.

ABSTRACT

Replication is a way to protect files from becoming inaccessible or destroyed by hardware and/or software failures. Several algorithms have been proposed to maintain the consistency of replicated files, but few authors consider the implementation details of these algorithms. The absence of guidelines for the implementation of a replicated file system (as well as other factors) makes it difficult for the average academic Unix user to exploit file replication.

This thesis identifies the major design issues found during the implementation of a replicated file system for Unix. Besides providing some useful guidelines for the design of such system, the main result is a set of portable, ready to use, C Shell scripts (plus a user manual), that allow any group of Unix users to replicate, and share, any number of files and/or subdirectories.

Static weighted voting with no read quorums, and the optional use of witnesses, is used to maintain the consistency of the replicated files. Replication is done transparently and at the file level. In other words, the intervention of the system manager or any special privileges are not required, and the user does not have to modify the commands, editors, application programs, etc. to use replicated files.

TABLE OF CONTENTS

I. INTRODUCTION

Due to its computational power, one of the most used operating systems, among the Computer Science departments of many universities, is Unix. Students and faculty are likely to use it for activities involving the use of important files that contain text, such as this thesis, source programs, etc. Subjective measures of the importance of a file are: the time it would take its owner to recreate it (given that it is lost due to an unfortunate accident), or, the effect on the user's work schedule produced by a situation where access to the file is temporarily disabled.

Several ways to protect important files are available to Unix users, such as the Source Code Control System (SCCS), and editors that maintain backups. But, since all the copies, or backups, are kept on the same file system, none of them protect the files against hardware failure and/or operating system shutdowns. This problem, if it is in the form of a disk failure, may (potentially) destroy all the copies a file. And while chances are that this will never happen, in a less critical, and more likely situation, it may prohibit the access to the file until the problem is corrected. File replication is a way to protect the files from these types of problems.

Most, if not all, of the available algorithms to maintain replicated files, have been designed to achieve higher reliability and/or availability, but since most of the available literature consists of theoretical papers, that do not deal with implementation details, the transparency of the services provided to access replicated files is almost never considered. While this may not be a problem for replicated data base users, since normally the programs that they use are custom designed, it does not provide the average academic Unix user (such as students and faculty of a Computer Science department) means to apply them easily. This thesis is not an attempt to provide a new algorithm to meet these goals, but to study how Unix users who work in an academic (or similar) environment may use an existing algorithm to replicate their files transparently.

Perhaps the work that is most closely related to this thesis is the one presented in an article called "Block-Level Consistency of Replicated files" by John L. Carroll[87], Darrell D. E. Long and Jehan-Francois Paris. They were not only concerned about improving the availability and reliability of the replicated files, but also about the transparency of the implementation of the replicated file system (all of these are also our goals). The core of their work is captured in this quotation from their conclusions.

We have investigated the construction of a reliable device. Such a device appears to the file system as an ordinary block structured device, but is implemented as a set of server processes on several sites. This allows us to provide replication while leaving the operating system kernel and the file system unchanged. Since the file system is not modified, the file operation semantics remain the same.

They justify their "block level" approach by saying that the file level implementation leads to complications for the implementor. They say that:

> . . . the implementor has the choice to provide the replicated file system as a set of library procedures or move the replication into the operating system kernel.
>
> The first choice is unsatisfactory because the implementor must provide an interface that preserves the semantics using extant system services. In essence the implementor must build an entire replicated file system on top of the original file system. The second choice is also less than satisfactory because it requires modification of the operating system kernel.

What we are trying to do here is replication at the file level, not at the block level. They claim that this is a bad choice, but our point of view is different, since we are attempting to provide a replicated file system, not for Unix, but for Unix users.

In the particular case of Unix, their "reliable device" probably would be added to the /dev subdirectory. A serious disadvantage to this approach is the fact that the owner of /dev (root) may not be willing to "pay" the cost of adding such a reliable device. Even worse, in order to really increase the file's availability, the device should be available in the /dev subdirectory of several nodes. In those cases where the system manager has not considered replication, the users who are interested in replicating at least some of the files (perhaps a minority of users) must somehow get the support from all the system managers of each and every node. Otherwise they would have to implement the device on their own directory, which is not possible since regular Unix users may not handle special character files.

As we will see, our implementation of a transparent replicated file system only requires a user account on each node (with a reasonable disk quota), and also allows each file to be shared by a different set of users, in contrast with the sharing capabilities provided (by Unix) for the special files in /dev (owner, group and other's access permission).

Our opinion is that the services provided by Unix (in particular the command aliasing capabilities), are strong enough to build a transparent, portable, user oriented replicated file system, at the file level. While this implementation may not be as

efficient as its block level equivalent would, its strength is based in its portability, which gives the average academic Unix user the necessary, ready to use, tools to exploit the advantages of replication. Proving this statement is what this thesis is all about.

After providing a very brief introduction to replicated file theory, we will turn to the structured design of a replicated file system. Our goal is not to summarize all the possible design and implementation issues that could be found during the implementation of such system, but to convey some of the major decisions. The discussions presented in the analysis of each issue, hopefully will convince the reader about the statement made in the previous paragraph. The culmination of this work is the actual implementation of a replicated file system, as a set of C-Shell scripts (a user manual, that was written as an independent document, is presented in the appendix). Our replicated file system allows any group of Unix users to replicate and share any number of files and/or subdirectories. Static weighted voting with no read quorums, and the optional use of witnesses, is used to maintain the consistency of the replicated files. The access to the replicated files is totally transparent.

In sum this thesis makes these contributions: it (a) displays the major issues on the design and implementation of a replicated file system for Unix, (b) provides operative code for such system, and (c) provides a user manual for this implementation.

Introduction to Replicated File Theory

There are three major characteristics related to replicated files (a) reliability, (b) availability and (c) consistency. Similarly, systems that maintain replicated files have two major characteristics (a) one copy serializability and (b) their ability to maintain mutual consistency.

Apparently there is no agreement on a formal definition for these terms in the related literature. Informally, reliability is a measure proportional to the time it takes for a replicated file to become inconsistent or destroyed. Availability is a measure related to the steady state probability that the user is going to be able to access a replicated file at any given time.

A side effect to file replication is sharing. Users of a replicated file system may be interested in replicating files, not only to increase their availability or reliability, but to share them with other users. On the same line, the availability would not be

increased if the algorithm would not permit access to the file in more than one of the nodes that contain a copy of the file. Under both circumstances, the user may attempt to access the replicated file(s) concurrently. Without attempting to give a formal definition of consistency and/or one copy serializability, let us quote a few paragraphs from Bernstein[87]:

> When two or more transactions execute concurrently, their database operations execute in an interleaved fashion. That is, operations from one program may execute in between two operations from another program, This interleaving can cause programs to behave incorrectly, or interfere, thereby leading to an inconsistent database. This interference is entirely due to the interleaving. That is, it can occur even if each program is coded correctly and no component of the system fails. The goal of concurrency control is to avoid interference and thereby avoid errors.
>
> . . . . . .
>
> One way to avoid interference problems is not to allow transactions to be interleaved at all. An execution in which no two transactions are interleaved is called serial . . . Serial executions are correct because each transaction individually is correct (by assumption)
>
> . . . . . .
>
> We can broaden the class of allowable executions to include executions that have the same effect as serial ones. Such executions are called serializable.
>
> . . . . . .
>
> In a one-copy database, users expect the interleaved execution of their transactions to be equivalent to a serial execution of those transactions. Since replicated data should be transparent to them, they would like the interleaved execution of their transactions on a replicated database to be equivalent to a serial execution of those transactions on a one-copy database. Such transactions are called one-copy serializable.

Notice that Bernstein uses the term "inconsistent" as an immediate consequence of the problems that interleaving may cause. In this case he is referring to a one-copy database. For example, consider a non replicated file used by a bank to keep all the balances of the accounts. This file would be inconsistent if we add each account's balance, and the total is not equal to the bank's assets. The consistency of a one-copy data base is usually defined by the semantics of the data. In the case of a replicated file, this also includes cases where the copies of the file are not identical. In our example, if the file is replicated, it would be inconsistent if the totals are different (obviously, if we add the balances when all the transactions are done).

From the practical point of view, the implementation of most software systems share some common goals, mainly, fast response time, efficient use of storage, and in many cases, transparency. Therefore, the criteria used for the design of most replicated file systems should be:

1. Maximize file availability and reliability.

2. Maintain file consistency and one copy serializability.

3. Minimize response time, and storage usage.

4. Maximize transparency.

5. Maximize portability (while this is not important in many systems, it is crucial for a replicated file system, as we will discuss later).

There are many ways to achieve the first two. One of them is known as voting (introduced by Thomas[79] and generalized by Gifford[79]). As its name suggests, this algorithm requires each node to collect a quorum of votes before accessing the file. If the node collects enough votes then it may access the file. Instead of summarizing other available algorithms, let us describe a special case of voting, called static weighted voting, since it is the algorithm that we are going to be using.

Static weighted voting assigns each node a number of votes (the question of how many votes to assign to each node, is beyond the scope of this presentation, see for example Garcia[85] and Barbara[86]). File access is classified into two different types, writes and reads. When a node wants to write to a replicated file, it has to communicate with other nodes and collect a number of votes greater or equal to a write quorum. Similarly, the read quorum must be collected for a read operation. If N is the total number of votes, W the write quorum and R the read quorum, the consistency and one copy serializability of the files are assured if: (a) $W>N/2$, (b) $W+R>N$, and (c) what is known as a two phase commit protocol is used. This protocol requires the file to be locked (impossible to be accessed) before collecting the quorum, and prohibits any node, whose copy is locked, to vote. Then the write access to a replicated file follows this algorithm:

1. If the file is locked, deny the access.

2. Lock the file.

3. Communicate with the other nodes that have a copy of the replicated file. Each node checks if its copy is locked, if it is it does not vote, if it is not, it locks its copy and replies with its number of votes and the version number of its copy (nodes that are down or unreachable do not vote).

4. If the number of collected votes is not greater than the write quorum, unlock the file, ask the nodes that voted to unlock their files, and deny the access. Continue otherwise.

5. If the local copy of the file is out of date (another copy has a greater version number), get the updated version.

6. Increase the version number, and send the updated version to all the nodes (not only to the nodes that voted, maybe a node that was down is now up).

7. Unlock the file and ask the nodes that voted to unlock their copies.

Here the first part of the two phase commit protocol corresponds to steps one to four. Steps five to seven are the second phase. The algorithm for the read access is very similar. Voting is said to be a strong algorithm to maintain the consistency of replicated files, since it is resilient to node and link failures and is able to work even if network partitions occur.

Other variations of voting include dynamic voting, in which the write and read quorums are changed every time a partition is detected, to improve the availability (see Davcev[85] and Jajodia[87]), and the primary site algorithm, where a node is given one vote while the rest have no vote rights (see El Abbadi[86]).

A peculiar variation is to use witnesses. Perhaps this quote from Paris[86] is the best way to explain what witnesses are:

> We propose to replace some of these copies by mere records of the current state of the file. These records, called witnesses, will be assigned weights and participate to the collection of quorums.
>
> We show, that under very general assumptions, the reliability of a replicated file consisting of n copies and m witnesses is the same as the reliability of a replicated file consisting n+m copies.

Clearly the use of witnesses could save a lot of storage, especially for large files. Besides his conclusions about the reliability, Paris only proves that the availability of a replicated file consisting of three copies is similar to the availability of two copies and one witness. Proving whether or not his results still hold if the number of copies and/or his assumptions are different, is beyond the scope of this thesis.

Related Literature

We have already mentioned and quoted a closely related paper about the implementation of a transparent replicated file system, however it may be difficult to read if you do not have any background on the theory behind replicated files. An excellent survey of many algorithms to maintain the consistency of replicated files, as well as an introduction to the major issues on replicated file theory, is presented in Davidson[85]. However this paper fails to survey dynamic voting. Another very good

introduction to voting, especially to dynamic voting, may be found in Sushil Jajodia and David Mutchler's paper (Jajodia[87]).

Classic papers about replicated files are Garcia-Molina[82] which explains the two phase commit protocol, and the paper where the first voting approach was introduced by Thomas[79], and later generalized to weighted voting by Gifford[79]. Vote assignment may not be a trivial problem, it is discussed in Garcia[85] and Barbara[86].

Other approaches to maintain the consistency of replicated files include: the primary site algorithm, presented in El Abbadi[85]; and the missing writes protocol, found in Eager[83].

Witnesses were proposed by Paris[86]. This paper shows how they may be useful to save disk space without affecting the reliability of a replicated file. If the reader is interested in changing the way that our system detects the updates, Metzner[82] may be useful.

II. DESIGN ISSUES

In this section we will cover some of the major issues on the design of a replicated file system. It seems easier to use our system once we understand all the design and implementation issues (i.e. to study the appendix after reading the body of this thesis). On the other hand it seems difficult to read the system's user manual, without knowing what our design decisions were. For these reasons, the reader may find it useful to skim the appendix at this point. Before we start, let us define a few of the terms that we will use in the rest of this thesis, and that may confuse the reader:

a. The words machine, node and computer are used as synonyms. However, since several users of the same computer may want to share a replicated file, in the discussion of the issues, each machine-name/user-name pair, will be taken as a node.

b. Similarly the words link, communication channel, port, etc. are also understood to be synonyms.

c. The absolute, or full name, of a file, always begins with a /, and gives its actual path (in a Unix file system) from the root of the subdirectory tree to the file.

d. The base name of a file is the string on the right of the last / of the file's absolute name (also known in the C-Shell documentation as the tail of the name).

e. The file name extension is the substring of a file's base name on the right of the last dot.

f. The words "directory" and "subdirectory" are supposed to be equivalent.

g. A "plain file" is a file, using Unix terminology, that is not a directory, pipe, link, socket, or special file.

h. A "file" is either a directory or a plain file.

i. From the Unix system point of view a file is locked when the user does not have read or write access to it, and unlocked otherwise. On the other hand, in a two phase commit protocol, a file (as we have seen) is locked when only one node may read or update it, and unlocked when no one has access to it. To avoid ambiguity, we will say that a replicated file is "locked" when write or read access have been removed from all its copies, and that it is "unlocked" when only one node has read or write access to it, and therefore, when only that node may read or update it.

It is very difficult to address some of the issues without a reference model. In most of the discussions we will use many Unix terms, and usually the goodness of a decision will be proved assuming that Unix is the operating system. For these reasons

the reader is supposed to be familiar with terms found in Unix manuals, books, etc. as well as with the C-Shell.

Operating System

We have chosen to use Unix, since it is one of the most popular operating systems used in academic environments. Unix is also a good selection for an operating system, since, as we will see, without its aliasing capabilities, it would be difficult to achieve transparency. Other Unix characteristics which make it suitable for our project are: the "rsh" command's ability to start a process in another machine, even using some one else's account; and it is the operating system most suited to run C.

Decision: Unix.

Programming Language

Our first major decision is to choose a language to code the replicated file system. The word "replicated" implies "several machines". If all the copies of a replicated file are to be placed on the same file system (one machine), we should not call that file "replicated", since its availability is not increased. Thus, the most important characteristic of the language to use should be portability. Since we have previously decided to use Unix, this narrows our options to C and Unix Shell programming.

Even though Unix Shell is interpreted, and therefore slower than compiled C, it has some advantages, among them: it does not require compilation and makes programs easier to export. For the same reason, it takes less time to develop and modify Shell than C.

Decision: write the programs using the Unix C-Shell programming language.

Method of Communication

Unix provides several high level communication facilities. There is even a replicated file (non transparent) maintenance command called "rdist". But, besides being a good academic exercise, since portability is an important characteristic, it seems to be better not to use commands that have been released recently, are too complex, copyrighted, non portable etc. The most primitive forms of communication on Unix (at the command line level), and therefore the most portable, are the "rsh" and "rcp" commands. These should be the method of communication.

Decision: use the "rsh" and "rcp" commands as the only method of communication between the nodes.

Machine Naming

As we have seen, the most portable forms of communication on Unix are "rsh" and "rcp". These commands are capable of using nicknames to identify the remote hosts. Thus the same machine may have different valid names, even worse, each machine may assign a different list of nicknames to the same node. Our replicated file system should be able to handle this problem.

One way to solve this, is to require the user to provide a function to map all the possible names of all the machines known to a node, to the valid local names. This function may be different at each node, and could, if necessary, return different names for the same machine when executed at different nodes.

Not only the machine name may vary, also the user's name may be different. Several users may want to share a replicated file, or the same person may have a different user name at each node. To further complicate things, consider the following scenario: a user has accounts (with the same user name) at three machines A, B and C, and wants to replicate two files. Suppose that all the copies of these files are to be placed on the user's home directory at each node, except for B's copy of one of the two files. How are nodes A and C supposed to know where to find B's copies of the replicated files?

If the user is not only replicating a file, but also sharing it with some one else, or if the user has different login names, the name of each machine should also imply a

home directory, for each replicated file. This way different replicated files may be placed at different home directories on the same node.

There are several things that we may do. We could require the user to provide another function, to map a node name to a user name. But this is a bad choice, since it requires the copies of all the replicated files to be placed in the same storage area (home directory) at each node, and does not allow the user to share different files with different persons at each node. To solve this, we could use a function of two variables: machine name and replicated file name, that returns a user name. The problem would be that the user would have to modify the function every time that a replicated file is added (or removed). Perhaps the best solution is to make our system think of each machine name, user name pair as a different node. While this has the undesired side effect of allowing two or more copies of a replicated file to be placed on the same machine (strictly speaking this is not a replicated file), it provides a very simple way to permit the user to share replicated files with more than one user at each node, without having to change the group's or other's file access mode (see Unix manual for chmod).

We must not leave out the possibility that one or more nodes may be mounted on a network file system, such as the Sun's NFS. On these systems, several computers share the same file server, therefore it is not a good idea to attempt to replicate a file on such systems, since all its copies will be placed on the same node. Our replicated file system should be "smart enough" to detect when two or more nodes are sharing the same file server. Since the NFS is not the only network file system available, and probably more will become available, we should not use its implementation details to detect if a node is mounted on a network file system.

We already require the user to write a function that returns the node name; therefore it seems quite reasonable that the best solution is to require this function to return the name of the file server, when given the name of a machine that shares it. This way no replication may occur on the nodes that share the same file system, since our program can easily detect when two or more copies are to be placed on the same machine.

Decisions: Each host name, user name pair is to be considered as a node. The user must provide a (very simple) program to map machine nicknames to valid host names that may be used with the "rsh" and "rcp" command. This program should return the name of the file server when given the nickname of any of the machines that share it.

File Naming

As we have seen a good replicated file system should provide means to access replicated files transparently, just as if they were any other file. This was not the only design criterion for a replicated file system, another thing that we expect from it is short response time. These are conflicting goals, since if there is no obvious way to tell when a file is being replicated (for example a naming convention), the system must somehow determine that by doing some extra computation. The next step of our design, is to decide which one to sacrifice, or to find a way to achieve both goals.

In order to achieve transparency, we must not impose any restrictions on the subdirectory structure. All the machines that have a copy of a replicated file must be free to place the user's home directory anywhere on the subdirectory tree. Similarly, we must not impose any restrictions on the structure of the user's home directory.

We should also be concerned about the base name of the replicated files. Should we let the copies of a replicated file have different base names? In order to have strictly transparent access to a replicated file, we must be able to name it, and rename it, as we wish. On the other hand, if we do not impose any restrictions on the names of each copy of a replicated file, we may not achieve good response time.

Let us consider how "total" transparency affects other design criteria. Each machine should "know" the subdirectory structure of the others i.e. the localization of the home directories of each user. But not only that, it should also keep track of the user's subdirectory structure, and their changes. This is computationally expensive; it implies that for every replicated file, each machine should maintain information about the absolute names of all the copies of the file (perhaps in the form of a table). Furthermore, whenever the user decides to change the subdirectory structure, or to rename files, and these changes affect a replicated file, the system must communicate with the other machines and order them to update their tables. Thus every execution of "mv" implies a table look up, and in the case of replicated files, communication with all the nodes that contain a copy of the affected replicated file. If the user is not lucky and one of the nodes is down, the change to the subdirectory structure must be postponed until that node is fixed.

It seems like we must give up on transparency, if we want to avoid this kind of problem, as well as keep a good response time (on the affected commands). As we will

see, this is not necessary, since Unix provides means to achieve both transparency and response time easily. Let us impose two restrictions on the naming of replicated files:

1. The base name of all the replicated files of each node, must be unique. For example, if a node contains a replicated file whose base name is "foo", no other replicated file at that node may have "foo" as its base name.

2. Each machine must use the same unique name to identify all the copies of a replicated file i.e. all the copies of "foo" must have "foo" as their base name.

This way each machine may keep a subdirectory of symbolic links, one for each replicated file, where the name of each link will be equal to the base name of the corresponding replicated file that it points to. As long as all the nodes agree on the localization of this subdirectory, each machine does not need to know about the subdirectory structure of the other machines; all it needs to know is where to find the link subdirectory.

Thus moving replicated files only requires a symbolic link modification on the file system of one machine. However, since many files may have the same base name, we still need to keep list of absolute names, to be used every time the user attempts to modify the subdirectory structure, to identify whether or not the change affects a replicated file.

If we impose yet another restriction on the replicated file's base names to identify them, for example adding an extension, like ".REP", we could avoid the need of such list. But besides losing more on transparency, we would also lose on what we may call the "correctness" of our file system, since other files that are not replicated may match our naming restriction and confuse the replicated file system. We cannot control the naming of other files, without modifying the operating system's file system, and perhaps affecting extant programs. On the other hand we may assume that the user is going to be careful and that this will never happen. Maybe it will be better if we only impose the two restrictions described above, and argue that, since it is expensive to replicate files (in terms of storage and communication costs) the table needed to identify them is not going to be large (at least for the average user), and that any search technique, such as hashing, could be used to improve the response time.

Analyzing these restrictions we will find that instead of being a burden, they actually help the user. Forcing all copies of a replicated file to have the same base name seems to fall on the line of consistency. After all, each copy of a replicated file is supposed to be identical, why should not we extend this all the way up to their (base) names? Similarly it is difficult to find a scenario where identical copies of a file must

have different (base) names. Forcing the user to give the same base names will help to identify which copies belong to a replicated file. On the other hand, those files that have different absolute names, but equal base names (such as core, foo, temp, etc.) are usually not important enough to be replicated.

Decisions: Impose no restrictions on the subdirectory structure of each node. Each node keeps a special subdirectory of symbolic links each pointing to the local copy of each replicated file. The user must give unique base names to all the replicated files and use the same base name for all its copies. Each node keeps a list of the absolute names of the copies of each replicated file that it contains. The "mv" command must update this list when the copy of a replicated file is moved.

Locking Mechanism

Before we decide how to lock a replicated file, we must define what locking means. Certainly, the user must not be able to update the file while it is locked, but that is not all. A copy of a replicated file must not be accidentally removed from any of the nodes. Therefore we are forced to use the file access mode as a locking mechanism. This is the only way that we can protect the replicated files from being (accidentally) modified by an output redirection, any command or application program, without having to modify the operating system. Thus, a file is said to be locked if the write access is removed, and unlocked otherwise.

However some Unix commands do not respect the file's access mode, and operate even if write access is not given. Consider the following facts about Unix behavior:

1. The commands "chmod", "mv" and "rm" work on plain files regardless of their access mode.

2. Only execute permission on the complete subdirectory path is required to change the access mode of any file.

3. Besides Unix super users, the access mode of a file may only be changed by its owner.

Obviously, we need to rewrite the "chmod", "mv" and "rm" commands. All should check if their arguments are replicated files. Our new version of "chmod" should not allow the user to:

a. Change the write access mode of a replicated file; this operation should be done by a separate command. Giving write access will deprotect the file from other commands.

b. Remove the owner's read permission on any replicated file. Out of date nodes must be able to read the updated copy of the replicated file (see discussion below)

c. Remove the owner's execute permission on any path than contains a replicated file, for the reason given in (b).

Similarly, our version of "rm" should not let the user remove a file if it is unable to dismount the copies from the other nodes as well as properly updating their list of absolute replicated file names (this was discussed in the previous section). The "mv" command should not allow any changes to the replicated file's base names, and must update the local list of replicated file absolute names when a replicated file is moved (as we have seen).

In order to maintain both consistency and one copy serializability, both read and write access must be removed from the file while it is locked, since voting requires a read and a write quorum. However, the user must have read access on the replicated files at all times, since an out-of-date node cannot collect both a read quorum on the most updated node (to be able to read the most updated copy), and a write quorum (to be able to overwrite). We either give up on response time, by requiring out-of-date nodes to change the read access mode of the most updated copy, before they attempt to update themselves (without collecting a read quorum), or give up one copy serializability.

Since the replicated file system is expected to be used by academic Unix users it is very probable the files are going to be texts. Many (if not all) of the updates to these text files are not a function of a previous read. In other words, situations that are usually found in a replicated data base system, for example, in a banking application, a transfer of funds from one savings to checking account, are totally different than the kinds of updates involved in the debugging of a program, or the writing of an essay, which are more likely to be found in an academic environment. Anyway, if one copy serializability is important, we are still giving the user the option to treat every read as a write access.

The fact that only the owner may change the access mode of a file implies that only its owner may unlock it, therefore, ownership determines whether or not a file

may be replicated. There is no way to avoid this restriction. The owner of the file may prohibit the replicated file system from reading or updating it, also the owner could avoid using the new versions of "rm" and "mv", etc. However, if the user wants to share and replicate a file, our replicated file system allows replication among equivalent users (equivalent users are defined in the ".rhosts" file), since each machine name, user name is considered as a node. Also the owner of a (replicated) file may give read access to other users by changing the file's access mode.

Removing write access is not a complete locking scheme. It is necessary, to forbid illegal updates, but it has a weakness: it cannot be used in a two phase commit protocol. In this case, a node that wants to update a replicated file must, roughly speaking, determine if the update may proceed (by voting for example), unlock the file, and then update it. If we only use the access mode, concurrent updates will not be aware of each other. Since the file is initially locked (no write access), when each node checks if the file is locked in the rest of the nodes, it will not be able to distinguish an idle copy from a locked copy, during the first phase of the commit protocol, unless we reverse things, and consider a file with write access to be locked. But that is dangerous, since for the fraction of time that the user may have write access, during the first phase of the protocol, the replicated file may illegally be updated or destroyed, therefore becoming inconsistent.

A separate data structure is required to distinguish locked from unlocked replicated files during the first phase of the commit protocol. This data structure can be implemented by a list of flags, one for each replicated file, indicating that the file is locked when its corresponding flag is up, and unlocked otherwise. The first phase of the protocol will use this data structure to determine if an update may proceed, allowing two or more concurrent updates to collide, before the files access mode was changed.

Decisions: Remove write access from the replicated files, initially and when they are not being updated. Use a separate data structure to mark the files as locked during the first part of a two phase commit protocol. Give write access only to one node at a time (second phase of the protocol). The user must have read access to the replicated files at all times. The chmod command must be rewritten to enforce these restrictions on the read and write access modes of the replicated files. The commands mv and rm must also be rewritten to protect the replicated file from being removed or moved, since they do not respect the access mode in some cases, and since these operation need to update the local list of absolute replicated file names.

Update Procedure

There are several available algorithms to process the updates to replicated files. Some of them are better in the sense that they provide higher availability, or are more likely to maintain consistency. Before we choose one, let us consider the characteristics of the environment where the replicated file system is expected to be used.

It is usually supposed that the updating process "knows" when the file is replicated, and that it will call an unlocking procedure before attempting to update it. In our case, that is not true. Since we are seeking transparency, we must not need to modify all the Unix commands or programs that may update the replicated files; locking and unlocking must be an independent procedure.

There are at least three ways to process the updates to the replicated files transparently:

1. Unlock the copies of all replicated files at login, let the user update the files during the session, and lock them at logout.

2. Start up a monitor at login, that would somehow trap the updates to the replicated files (if any), and both unlock and lock the files properly while running on the background.

3. Modify the user environment in such a way that the replicated files are unlocked just before, and locked immediately after, the execution of the commands that may affect them, by rewriting or aliasing the commands.

Since the files are going to be unlocked only once, the first option seems to be the best in terms of response time degradation. The problem is that if the login sessions are too long, the file will remain unavailable at the rest of the nodes for the same period of time, and since the updates are not going to be propagated as they arrive, the file reliability will also be affected.

Before we continue, we need to collect statistical information to determine the average length of the login sessions. A statistical study to show whether or not this data is representative, and how it may affect our results if not, is beyond the scope of this thesis. However, studying the computing environment provided to the students and faculty of the Computer Science department of this university, may give us a rough idea about the activity of Unix users in similar installations.

It is very easy to find out what is the average length of the login sessions on any Unix environment. Unix systems maintain a file called /usr/adm/wtmp (see the Unix manual for the who command). Unix adds a line to this file every time a user logs in or out, and every time the ftp command (connecting the local host with another machine) is executed at a remote host (unfortunately the system manager usually chooses to erase its contents frequently). Each login line, contains the user name, terminal identification and a time stamp. Each logout line has the terminal identification and another time stamp. The following Unix command may be used to extract the login information from this file:

```
who /usr/adm/wtmp | awk\
'{    if (NF>4)\
      {     if ($1!="shutdown" && $1!="reboot" &&\
                $1!="root"     && $1!="sysmgr")\
                  beginning[$2] = $0;\
      }\
      else\
      if (length(beginning[$1]))\
      {     print beginning[$1];\
            print $0;\
            beginning[$1] = "";\
      }\
}'
```

This will output a text stream, with an even number of lines. Each pair of lines is a user login/logout entry. The time difference between time stamps of each pair of lines produces the lengths of the login sessions for all the users. Notice that the activity of the system manager and "root" is removed from the output. The results obtained from this command, (a similar one was executed to extract the average time between failures) are given in Table I.

The first seven machines on table I correspond to Sun workstations, assigned to faculty members of the Computer Science department of this university. Since the console of these machines is only available to one professor, and other users must execute the "rlogin" command to access them, most of the login sessions correspond to one user (Dr. Jean Blair at blair.cs.utk.edu, Dr. Jeff Case at case.cs.utk.edu, etc.).

Table I. Average length of login sessions (ALLS), in minutes, and average time between failures (ATBF), also in minutes, on several Unix machines, calculated from the given date, until May 16, 1989.

machine	data collected from	ALLS	ATBF	number of sessions
blair.cs.utk.edu	7/15/88	1142.22	12202.2	267
case.cs.utk.edu	8/12/88	1229.43	8558.8	261
char.cs.utk.edu	7/12/88	954.04	12121.7	384
cockett.cs.utk.edu	7/10/88	549.47	12727.3	655
maclennan.cs.utk.edu	7/14/88	2144.43	12208.4	139
mayo.cs.utk.edu	8/16/88	3505.51	7845.2	55
mutchler.cs.utk.edu	7/16/88	380.59	9396.3	928
css.itd.nrl.navy.mil	4/17/89	111.25	5120.2	5033
ornl-msb8k.arpa	5/01/89	86.50	718.0	206
msr.epm.ornl.gov	4/15/89	183.28	4584.3	2642
alphard.cs.utk.edu	8/05/88	99.10	7438.4	9491
utkcs2.cs.utk.edu	3/14/89	78.37	6982.1	5231
cetus3a.cs.utk.edu	8/02/88	68.11	5924.0	1184

Notice the extreme contrast between the length of the login sessions (as well as the number of sessions) at faculty workstations, and the length of such sessions at the three machines on the bottom of the table. Alphard, utkcs2 and cetus3a are predominantly used by students, who have access to these machines through "public" terminals or workstations, therefore, they must be present while they are logged in.

Our replicated file system must be useful to both students and faculty. The average length of the login sessions at the faculty workstations is a little more than twenty four hours, and since those machines are some times used by students, the actual average of the faculty login sessions should be a little bigger. If the system locks and unlocks the replicated files every twenty four hours on average, probably it wouldn't perform better, in terms of availability and reliability, than a ".logout" file with one or more rcp commands on it. Even though this would not be a problem for short sessions, locking and unlocking ALL the replicated files in a short period of time probably would waste too much CPU time, since the replicated files are not always going to be updated. And as the number of replicated files increases, the number of processes started during the login procedure may severely affect the machine's response time.

Given the variance of the results, we must find a way to unlock and lock the replicated files regardless of the length of the login sessions. Let us consider the second possibility, starting a process that would run on the background which by monitoring user activity, would decide when to unlock and lock each replicated file.

This strategy cannot provide transparency and replicated file consistency at the same time, without having to keep an update log file. If such file is maintained, either the user has to inform the monitor that he/she wants to access a replicated file, or it somehow would have to continuously check the time stamps of the replicated files, or the user activity on the system (using the information generated by the ps, w, who, etc. commands).

In the first case, the access would not be strictly transparent. Considering the second option, if the user does not have read write access to the replicated files, there is no way to update them before they are unlocked, unless the replicated files are left with write access. We cannot do that since write access is our locking mechanism, and we may add the fact that it allows the user to access the replicated files before they are actually unlocked. On both cases we would need a log file (perhaps a copy of the replicated file) to undo the transactions performed on the file, whenever the unlocking procedure fails.

Before we give any further thinking on the first two choices, let us consider the third one. As we will see it will prove to be the best option.

Unix has the ability to give aliases to the commands. We may alias all the commands that could potentially update a replicated file in the ".cshrc" file in such a way that we would make the operating system execute a program that checks for the arguments looking for replicated file names, before actually executing the command. This way, replicated files can be unlocked by this program before they are actually updated. For example, if we want to use "vi" to access the replicated files, and the name of the program that does the unlocking/locking is "xxx", the entry in the ".cshrc" file would look like this:

alias vi xxx vi

This way "xxx" has the opportunity to check all the arguments passed to "vi", and knows who to call after it decided whether or not to unlock one or more replicated files. Also, if a replicated file was one of the arguments, "xxx" may stay on the background waiting for "vi" to terminate to lock the affected replicated files, and to determine if they were actually updated. Besides being a transparent way to access the replicated files, it is better than the other options we have considered, since it does not unlock the replicated files when they are not going to be used and we would never need to undo an update.

Decision: Add aliases to the ".cshrc" file in such a way that the replicated files are transparently locked/unlocked before/after the execution of the commands used to access them.

Algorithm to Maintain the Consistency

Since our main goals are to provide transparency, portability and consistency, we should not be too concerned about the other characteristics of the algorithm such as the availability, as long as they are reasonable acceptable.

Of all the algorithms available, static voting seems to be the best choice. Besides being easy to implement, it is not a weak consistency algorithm and it may also be used to implement other algorithms.

Voting requires each file to have both a read and a write quorum. Read quorums are difficult to implement using our locking mechanism. Consider the case of an update

arriving to an out-of-date node. If both read and write access are removed while the replicated file is locked, the most updated node must unlock the replicated file (using the read quorum) to make it readable, but this is impossible, since the out-of-date node unlocked the file before (using the write quorum).

As long as the user knows that the fact that he has read access to the replicated files does not imply that he is looking at one of the most updated copies, not using read quorums is the best option (given our locking mechanism). This will also reduce the response time of the lock procedure, since the node that updated a replicated file does not have to wait until all the nodes update themselves before locking the file since only write access is removed (see the section about exploitation of parallelism).

Decision: Use weighted static voting with no read quorums to maintain the consistency.

A summary of the preceding design decisions as well as the implementation decisions appears in the conclusion of this thesis.

III. IMPLEMENTATION ISSUES

So far we have considered some of the issues related to the design of a replicated file system for Unix. In this section we will investigate the problems that we may encounter during the implementation of such system in Unix (however it is difficult to separate all the implementation from the design issues). We will try to be as general as possible. The reader is encouraged to consult the user's manual (presented in the appendix) for minor implementation details not covered here.

Plain File Handling

The first issue here is to define what is a file, and how are going to treat them. Since we expect our system to be used in an academic environment, probably most of the files, important enough to be replicated, will be text files. Thus, each replicated file can be taken as a sequence of characters; it is not important to view them as a collection of records.

These character files may be treated down to the character level or to the block level. Viewing them as a sequence of blocks may significantly reduce the amount of data to be sent through the links after an update, i.e. out-of-date nodes would only need to get a copy of the updated blocks, not a copy of the updated file. Also, if our locking mechanism locks blocks, not files, several users may access the same replicated file at the same time, as long as they use different blocks, therefore the availability would be dramatically increased.

But treating replicated files down to the block level imposes some severe implementation complications. We would need to assign an individual version number to each block. And on top of that the unlock procedure should either copy the block or calculate its checksum before each access, to find out later if it was actually updated. Furthermore checksums are not reliable means to detect the updates (i.e. an unchanged checksum is not a guarantee that the block has not been changed), and if we view each file as a sequence of characters, we would just be taking the block size down to its minimum.

Due to these complications, it is easier, and computationally less expensive, to treat the replicated files as a whole. This way we could simply use the time stamps, kept

by Unix for every file on the directory entries, to detect if it was updated or not. The command "rcp" can be used directly, since we would not need to create temporary files (the updated blocks) after each update (rcp does not copy pieces of files). And only one version number is required. If the user is too concerned about inefficient network usage, he/she may break large files into several small ones (perhaps by sacrificing some transparency).

Now that we have decided how to treat plain files, we should ask ourselves how to treat the symbolic links that may point to them. It probably does not make too much sense to talk about replicated symbolic links and/or replicated special files. Replicating special files (files on /dev) may sound like an interesting idea but that is a totally different problem. About links, besides being difficult to copy (due to their semantic meaning for Unix commands) since the subdirectory structure may vary from node to node, we could not make them strictly identical. In any case, replicating symbolic links seems useless since the user may directly ask the system to replicate the file pointed to by the link, and there is no such thing as an update to a link, they are only created or destroyed.

Decisions: replicated plain files are viewed as a whole. Only one version number is assigned to each replicated plain file, and the entire plain file is sent to out-of-date nodes. Symbolic links and other special files cannot be replicated.

Replicated Subdirectories

As with plain files, it is easier to think of each replicated directory as just one file. But let us analyze this carefully.

For the same reasons that we have decided to use the same base names for the copies of each replicated file, we may also extend our notion of consistency of the replicated directories not only to the contents of their files, but to the directory entries. Thus a strictly identical copy of a replicated directory must have the same number of files, each file in the directory must have the same name, and the hierarchy must be preserved. On the same line of thought we would expect each file in the replicated directory to have the same time stamps, but this, as we will see, is impossible to implement (without losing consistency) given that the time stamps are used to detect the updates.

It is probably not very useful to handle replicated subdirectories of replicated files. This assumes that a copy of the replicated subdirectory is going to be kept in some nodes, but also that each of its files may be kept in a potentially different set of nodes. For example, consider a replicated subdirectory that contains file "foo", and suppose that this subdirectory is being replicated at nodes A, B and C. Also suppose that file "foo" is replicated at nodes D, E and F. It would be very difficult for nodes D, E and F to know that they need to update nodes A, B and C when the file is modified, and vice versa. In this case it is better to move "foo" out of the directory and treat "foo" and the directory as two different replicated files.

Another thing we should consider is the replication of directories that contain links. These links may be a totally portable reference, like "." or "..", or it may also point to a file within the replicated subdirectory, etc. Links may save the user a lot of disk space, thus we should permit the replicated subdirectories to contain them. For this reason links should not be resolved when an update to the subdirectory is propagated; besides wasting space, the file pointed to (for example /bin/rm) may not be portable. It is difficult to decide how we should view the files pointed to by the links. Should they be identical in all the nodes? Probably not. Then is the link really being replicated? May be it would be better if we just worry about keeping the semantics of the links and the copies of the replicated directories totally identical, and let the user worry about how links are resolved on each node.

Decisions: Each copy of a replicated directory must have the same number of files, with the same names and hierarchy. Replicated directories are allowed to contain symbolic links, but they are not resolved before transferring them from one node to another. Replicated directories should not contain independent replicated files.

Exploiting Parallelism

The implementation of our system obviously requires the use of "rsh" or similarly slow commands. In the first phase of voting, when a node is collecting the quorum, it needs to communicate with each node to find out if it has an out-of-date copy, if the other nodes are up, etc. As the number of nodes increases, the sequence of "rsh" commands may take too long, and seriously affect the response time. Fortunately, the C Shell has job control facilities that allows us to run things in parallel. The C Shell

permits communication between parent and child processes via environment variables; however this only works one way, from the parent to the child, i.e. the child cannot set the parent's variables. To show this, consider the following loop (where the variable "hosts" has been properly initialized):

```
foreach host ( $hosts )
        set $host = `rsh $host hostname` &
end
wait
```

Each set statement will be executed in a subshell, effectively in parallel, but no variables will be added to the process that executes the loop. If we move the ampersand inside the quoted string, the set statements will not be executed in parallel. Another problem, that has nothing to do with parallelism, is that the host names must be legal variable names. We can avoid these problems, by using temporary files as variables. For example:

```
foreach host ( $hosts )
        rsh $host hostname >! #$host.answer &
end
wait
set nodes_up = 0
foreach host ( $hosts )
        if ( ! -z #$host.answer ) then
                @ nodes_up++
        endif
end
```

This way, the variable nodes_up will end up with the number of successful "rsh" commands, and the response time would be a little more than the slowest "rsh" command, since it is slow due to communication delays, not because intensive use of the CPU. If the time that the slowest rsh takes to run is unacceptable, we need to change the wait command. In this case we only need to wait for a number of rsh (to be executed successfully) equal to the write quorum. The C-Shell is not so powerful, but given that the jobs command produces no output when there are no background processes under job control, we could do something like this (again assuming that the variable "hosts" has been properly initialized, and that each node is assigned just one vote):

```
foreach host ( $hosts )
        rsh $host hostname >! #$host.answer &
end
@ quorum = $#hosts / 2
loop:
        set nodes_up = 0
        foreach host ( $hosts )
                if ( ! -z #$host.answer ) then
                        @ nodes_up++
                endif
        end
        if ( $nodes_up <= $quorum ) then
                jobs >! #jobs.output
                if ( ! -z #jobs.output ) then
                        goto loop
                endif
        endif
```

Now, instead of waiting until all the rsh commands are done, we would only wait until the quorum is collected. Notice that the locking procedure cannot use the jobs command as gracefully, to determine if all the rsh commands started by the unlocking procedure are done. While this is not a problem if we want to include as many nodes as possible in the update (i.e. the locking procedure may ignore the remaining background processes and attempt to update all the nodes), this may be a problem in some variations of dynamic voting.

Let us analyze where else can we exploit parallelism. Each node must be able to update itself, since updates to the replicated files may arrive at an out-of-date node, and it needs to know how to get the most updated version, before processing the update. Our system may put the code needed to update an out-of-date node on a separate program. Let us call this program rep-update. This way when an update arrives to an out-of-date node, all it needs to do is to execute "rep-update" to get the updated copy from the most updated node. Obviously the parameters for this program would be the name of the most updated node and the name of the replicated file.

We may also exploit parallelism on the update propagation process. After an update is processed by a node, the rep-update program may be started by this node on the other nodes using the rsh command. In other words, once an update is processed at any node, it may "tell" the other nodes that they are out of date. This is possible since the replicated files are readable at all times. Therefore, the update propagation procedure

may look like this (assuming that the variable rep_file and hosts have been properly initialized):

```
set here = `hostname`
foreach host ( $hosts )
   rsh $host -n\
      "rep-update $rep_file $here >&! /dev/null &" &
end
```

Notice that the standard input and the standard output connections of "rep-update" are broken. This way the rsh command ends before "rep-update" does, effectively distributing the processes required to propagate an update among all the nodes, therefore significantly reducing the response time of the update propagation procedure. This also has the effect of reducing the number of active background processes on the node that is locking the file. Even though the remote nodes may execute an "rcp" which starts a background process on the local node, they are not going to be synchronized due to different communication delays.

Decision: execute all the sequences of rsh commands in parallel. Implement the update propagation as a parallel distributed process. Collecting the quorum is also a parallel process, but this process should be terminated when the quorum is collected, not when all the parallel processes are done.

Update Detection

There are several things that we may do to detect that a replicated file was updated (after it was unlocked). Basically our options are: (1) to calculate checksums, (2) to save a copy of the file, (3) to save the time stamp.

Using the time stamps to detect updates is one of the easiest ways to detect when a file has been modified. This way we will not be affecting the response time, or wasting disk space; however it has some problems.

Unix keeps the year, month, day, hour and minute of the modification on each directory entry (time stamp) but not the seconds. If the file is successfully unlocked, updated and locked in the same minute (according to the local clock) the update goes undetected, and the file becomes inconsistent. We need to know if the three processes,

unlocking, update and/or locking may be done in one minute or less. To do this consider the contents of table II.

Notice that, if all the nodes are up, the slowest rsh would take twenty two seconds on cs2, thirty five seconds on mb8, seventeen seconds on mut, etc. Clearly, since the "rsh" commands are going to be executed in parallel, in order to maintain the consistency of the file, in cases when the update process takes only a few seconds, the unlocking procedure must follow these steps:

1. Save the value of the local clock.

2. Collect the quorum.

3. If it was successful, check the clock again, if the minutes are the same as the minutes on the saved time, sleep until the next minute (less than sixty seconds). Otherwise continue.

4. Unlock the file.

So far using the file time stamps as an update detection mechanism seems to work fine. However, there are still some more details to be taken care off. We should not allow out-of-date nodes to preserve the time stamp of the replicated files when they are copied from the most updated node, since the node's clock cannot be synchronized (by a non super user). For example, if a node clock's value is 2:00pm, when a replicated file is updated, and the clock's value of an out-of-date node is 1:05 when it updates its copy, preserving the time stamp, it will not be able to detect the updates until fifty five minutes are past.

Another problem with time stamps is that changing the access mode of a file does not modify the time stamps. Therefore unlocking a file, changing its access mode, and locking it again, will not be seen as an update, and the other copies will end up with inconsistent access modes. The first problem (difference between the clocks) can be solved if the unlocking procedure changes the time stamp of the replicated file, but this cannot be done easily with subdirectories (the touch command does not work on directories); besides, we would be resetting the file's modification time every time its accessed or modified. Since the time stamps are going to be inconsistent, it seems that giving up on the consistency of the access modes of the replicated files is not too much of a loss. Also we may argue that we need to respect the "umask" settings of each node (which may be different).

Table II. Remote shells (rsh) communication delays, and time outs (rsh failures) for a partially connected network. All results are in seconds, a - means non applicable, a * means non available. To obtain these results, each node executed "rsh host exit", every fifteen minutes, with each possible host, for a period of two weeks, measuring the execution time. Messages such as "connection refused", "logins disabled", etc. were taken as a time out.

	c21 rsh tout	css rsh tout	mb8 rsh tout	msr rsh tout	mut rsh tout	cs2 rsh tout	ux1 rsh tout
c21	- -	* *	* *	013 049	012 053	020 053	008 029
css	* *	- -	045 054	034 050	* *	025 055	017 050
mb8	010 565	035 558	- -	018 415	012 567	022 566	009 258
msr	013 082	046 079	013 *	- -	022 067	029 068	012 022
mut	014 048	* *	016 049	015 046	- -	017 *	006 015
cs2	019 084	017 051	022 081	023 055	015 *	- -	012 025
ux1	013 121	010 026	014 128	013 065	007 *	016 *	- -

An equivalent way to save the time stamps, is to save the output of a recursive long listing of the file before the it is unlocked, and compare that with the output of the same command after it is locked (this works for replicated plain files as well as for subdirectories). The difference between both outputs would tell us which files were modified including changes in the access mode, but still, this strategy cannot handle the problem of differences between the clocks. Besides, interpreting the outputs for replicated subdirectories efficiently (in such a way that the minimum number of files is transmitted) may be a difficult task, especially if the subdirectory structure changed.

Decisions: Use time stamps to detect the updates. The system can only try to maintain the consistency of the replicated files access mode. The time stamps of the replicated files are not preserved, when an out-of-date node updates itself, it sets the time stamp of the replicated file, to its clock value.

Replicated Directory Update Detection and Propagation

The user may want to replicate a subdirectory containing a large number of files, and/or subdirectories (depth greater than one). Even though we are not too concerned about efficiency, we should not send the whole subdirectory to the other nodes after each update.

Keeping a version number for each file in the subdirectory is out of the question. This is no better than a replicated file system that does not allow the user to replicate subdirectories (in terms of response time required to determine if the updates affect a replicated file, which file to unlock, etc.). Apparently the solution is to make the locking procedure check the time stamps of all the files in a replicated subdirectory, looking for those that were modified after the file was unlocked, and then send only these files to the other nodes. To explain why this would not work so easily, let us consider some facts about Unix:

1. The commands rcp, cpio and tar (which may be used by our system to transmit the updates) are file oriented, i.e. they do not overwrite directories. For example, suppose that subdirectory "foo" has three files A, B and C, and that rcp is used to copy "foo" into another node's subdirectory "dir1" that also has a subdirectory named "foo" with file Z on it. After rcp is done "dir1/foo" will contain four files, namely A, B, C and Z.

2. The directory's time stamp is only changed when a file is added or removed from it.

3. The command rcp follows the symbolic links before copying. It also requires write access on the subdirectories of the directory to be copied.

Since the time stamps of the deleted files are no longer available, we also need to consider the time stamps of the subdirectories. If we are not careful, as the user removes and adds files from, and to, the replicated subdirectories, the number of files will grow, since the removed files are not deleted at the remote nodes (by a recursive rcp or equivalent command). Furthermore we cannot use the rcp command to copy replicated subdirectories. Besides following the symbolic links (we have already decided that we do not want to do that), both read and write access on the copy of the most updated node is required, in order to let out-of-date nodes use rcp to copy a replicated file.

While the "rep-update" program that we have already talked about only needs to execute the rcp command to copy replicated plain files, it requires more complicated things to copy replicated subdirectories. The algorithm that it must follow to copy replicated subdirectories should be:

1. Get the names of the files (of the replicated subdirectory) that were updated.

2. Create a sequential (tape) archive file with these files, on the most updated node.

3. Copy the tape archive from the remote to the local host using rcp (actually it is more efficient to pipe the output of the tar command through an rsh command).

4. Remove these files from the local copy of the replicated file.

5. Extract the contents of the tape archive on the local copy of the replicated directory.

Let us analyze this algorithm step by step. After a replicated file is updated, the most updated node (where the update arrived) must find out which parts of the subdirectory were updated, in order to send the minimum number of files. For example, if a replicated directory contains one hundred files, and only one of them is updated, then the other nodes should only be asked to copy one file. This can easily be implemented with the find command. This list of updated files must be saved for later use, since the remote nodes may take some time before they actually start the "rep-update" program.

While the most updated node is traversing the updated replicated subdirectory, looking for updated files, it must prune the search when it finds an updated subdirectory, since if a file is removed, there is no way to know what the name of that file was. Remember that files are not removed from the remote nodes by the tar

command. The "rep-update" program must do that. This has some serious complications. Consider a replicated directory that contains two subdirectories, with fifty plain files each. If the update consisted of the removal of one of the plain files, the remote nodes would need to copy a subdirectory with forty nine files. Obviously, there are more efficient ways to handle this: we may reconsider analyzing the outputs of recursive long listings before and after the update to determine which files need to be transferred, removed, moved, etc. But as we said, this analysis may be a difficult, time consuming task. Another easier way to solve this is to ask the user to change the subdirectory structure before he replicates it. The user has the option to convert wide subdirectories to equivalent deep subdirectories, or not to add or remove files from replicated directories.

Once the list of updated files is ready (step one), it would be very inefficient to start the execution of a rcp command for each file in the list; besides we have already seen why we should not use this command with subdirectories. It is better to join all the files into a sequential file, and then "rep-update" may import them in just one session. It is not until all the updated parts are safely placed on the remote nodes that the atomic operation of updating the remote copies begins. Since this atomic operation involves local processing only, it will be done in less time, thus reducing the probability that a failure may disrupt it.

There is just one minor detail: where should the lists of updated files for each replicated directory be kept? We are already keeping a data structure at each node (the set of flags that we talked about) to identify when a replicated file is locked. We must add to this a list for every replicated directory. Each node that has a copy of the replicated directory must maintain this list and keep it available since out-of-date nodes need to know its contents to update themselves. This set of lists must be replicated among the nodes. To show why this is required, consider the following example: Three nodes, A, B, and C contain a replicated directory. Node C is down. An update arrives at node A, and the replicated directory is updated. Node A asks node B to update itself, and after B is updated node A fails. Node C comes back up. An update arrives to node C, it will find that it is out of date and run the "rep-update" program. If node A is down how is it supposed to get the list of updates? Clearly node B must have it. Furthermore, what if the version number of C's copy is strictly less than one minus the version number of B's copy? This is yet another implication: if an out-of-date node holds a copy of a replicated file whose version number is not the previous version of the most updated copy, it must copy the whole replicated subdirectory to update itself.

Once again, the easiest solution is to ask the user not to replicate large directories if for the given parameters (update rate, node failure, etc.), out-of-date nodes are expected to fall behind too frequently. This is one of the places where our implementation may be improved.

Decisions: The rcp command should not be used to copy replicated subdirectories (or parts of them). The user must be careful in the design of the hierarchy of a replicated directory. It should contain a small number of plain files, or any number of subdirectories each with a fraction of the files. If rsh failures occur too frequently compared to the replicated subdirectory update rate, it is better to replicate small subdirectories, since an out-of-date node whose version number is not the previous version of the most updated node must copy the whole subdirectory.

Local vs Remote Locking

One of our design decisions was to use the file access mode and a separate data structure to lock the files. This data structure is no more than a set of flags. When a flag is up, it indicates that its corresponding replicated file is locked. In this subsection we will discuss whether or not this data structure needs to be replicated.

Reviewing the voting algorithm presented in the introduction, it appears that this should not be an issue. The difference is that the use of the word lock in this algorithm is not what our replicated file system uses it for. In our case, a write permission is removed from the replicated files initially, and when they are not being used. Then what is the good of asking the other nodes to lock the file? Obviously there is no way for them to update it.

Given our locking mechanism, the only need to ask other nodes to lock the file during the first part of the two phase commit protocol, is to make the algorithm resilient to network partitions. For example, suppose the file is initially idle in all the nodes, and then, a node locks the file (locally) by raising its corresponding flag and by changing its access mode. Immediately after the file is unlocked and before it was updated, link failures isolate the node in such a way that it is connected to a group of nodes whose total vote count is less than the write quorum (known as a non distinguished partition). Then if an update arrives to another node that is able to collect the write quorum (a node in a distinguished partition) before the links are fixed (before the

partition disappears), both updates will be processed at the same time, making the file inconsistent.

Nevertheless locking the file in all the voting nodes has some disadvantages. Perhaps the most obvious is that it increases the probability of deadlock. This can be seen if we review our previous example. If the file is locked in all the nodes, before the partition occurs, and the links are not repaired before the update is processed, the nodes in the distinguished partition will remain locked, unless the node that updated the file waits until the links are fixed. Another disadvantage is extra overhead on the read accesses since, even when the file was not updated, we still need to communicate with the other nodes and ask them to lower their flags.

We either make our replicated file system not resilient to some occurrences of network partitions, by unlocking the file locally, or we increase the probability of deadlocks and add some overhead to the read operations. To make this decision we need to collect statistical data to determine the probability that the situation presented in the previous example will ever occur. For the moment consistency should be our main goal, thus, we must lock (i.e. raise the corresponding flag) in all the nodes.

Decision: The data structure used to identify when a file is locked, is updated in all the nodes by the locking and unlocking procedures.

Incorporating Witnesses

Witnesses may very easily be handled by our replicated file system. Remember that each node must keep a subdirectory of symbolic links to achieve transparency. If a node is a witness, these links may point to a non existing file, for example "/witness". This way the unlocking procedure can easily prohibit updates at these nodes, all it needs to do is to check if the link points to this non existing file. Furthermore, an out-of-date node that tries to update itself from a witness node, will always fail to get the copy since "/witness" does not exist. The only problem is how are out-of-date nodes supposed to know which node has a most updated copy and which is just a witness. Instead of imposing restrictions on the node naming convention we may ask the user to put the witnesses at the end of the node list. This way, since the our implementation uses "foreach" loops, will only try to get a copy from a witness after it has exhausted all the other possibilities.

Decisions: Use a file naming convention to identify copies of the replicated files that belong to witnesses. This restriction does not affect the name of the replicated files, but the links kept by the replicated file system. The links should point to a nonexisting file, if the node is a witness.

IV. EVALUATION OF THE REPLICATED FILE SYSTEM

So far we have covered all the major design and implementation issues of a replicated file system. We should now test these programs to find out how well we achieve our original goals. The proper approach would be to evaluate the system's ability to maintain the consistency and one copy serializability of the replicated files, as well as the provided availability and reliability. We either design a model for our system to calculate these measures, or produce statistical data from the use of our system. The first option is beyond the scope of this thesis. On the other hand, in order to get the data we need people to use the system for a period of time. This would be too difficult to achieve, given our time limitations.

Nevertheless, just by looking at the code, we are able to determine where the consistency and one copy serializability is most likely to be lost. The next few paragraphs are a brief discussion about this, as well as some guidelines to improve our implementation.

Loss of Consistency

There are some places where our programs cannot be interrupted, for example, when it updates the absolute name list, the data structure to distinguish locked from unlocked files, etc. In all these cases, if the program is killed, the stored information about the replicated files, or even a copy of the replicated file, may be scrambled. Since our implementation is user oriented, we cannot protect the system against the kill command, system reboots, shutdowns, etc. The only thing that we can do is to arrange the statements of our programs in such a way that slow commands such as the rsh, are not included within the boundaries of the atomic operations.

Another rather obscure situation that would make our system lose the consistency of the file is clock failure, but given that it must occur while a file is locked, we should consider this event as impossible.

Suggestions for Improvement

1. Translate the new versions of the chmod, rm and mv commands, as well as the program to be used to alias the commands that may affect the replicated files, to C. This will improve the response time without affecting the portability. The rest of the programs should NOT be translated, since the required libraries to implement some of the commands may not be portable. However using TCP (Internet Transmission Control Protocol) primitives may produce portable code.

2. Improve the update propagation procedure, especially for replicated directories. The diff command may be used to generate commands for the editor ed which will generate the new file from the old one, thus reducing the amount of information to be sent from the most updated to the out-of-date nodes. Something similar can be done with the output of a recursive long listing of replicated directories. A major flaw of our implementation is that if a file is added to, or removed from a replicated directory the whole subdirectory is transmitted. Instead the most updated node can order the out-of-date nodes to remove or add the affected files.

3. Design a separate program to do static vote reassignment based on the previous history of the update, and failure rates. Since the votes assigned to each node must be kept in a file, a separate process that uses the same replicated file locking mechanism, may lock the file using a quorum equal to the total number of votes, and instead of updating it, may perform this vote reassignment. This would not be as good as dynamic voting but it may be useful to adapt our system to a changing or unknown environment.

4. Study the use of the system to decide if using another algorithm to maintain the consistency may significantly improve the availability. Most of the work that has been done to benchmark the algorithms is based upon general assumptions. Many authors assume uniform demand, no communication delays, exponential time between failures, etc. which may not hold on an academic environment. For example, a replicated text file may be idle for a long period of time, and then the user may frequently update it for a short period of time. Also, as we have seen, communication between nodes may even take more than one minute. Whether or not Paris' results on witnesses, or the goodness of dynamic voting, are still true under the circumstances presented in an academic environment, are difficult statements to prove using models. Collecting representative statistics seems to be a reliable (but time consuming) way to benchmark the algorithms

in the real world. Alternative ways to measure the availability of a system (such as the mission availability), which may produce more realistic replicated file system benchmark results, are presented in Brender[68a,68b].

5. Implement dynamic voting. This can very easily be accomplished, since this particular algorithm requires very little extra information that may be easily stored and retrieved by appending it to the names of the links in the form of extensions, or even better to the number of votes in the votes file.

6. We did not considered the time stamps and access modes of the files in a replicated subdirectory as being part of its contents. Under this perspective the copies of the replicated subdirectories should produce exactly the same result for a recursive long listing of them. It will also be desirable to have all the copies of a replicated file to have the same time stamps. Is there a way to maintain the consistency of the time stamps and the access modes of the replicated files without using the file modification time as the file's access time and using the time stamps to detect the updates?

V. CONCLUSION

We have covered some of the most important design and implementation issues of a user oriented replicated file system for Unix. While our approach was not general, the reasons that we gave for each of our decisions may be useful guidelines for those readers interested in the implementation of a similar system. The culmination of our work is the actual portable implementation of such system, plus a user manual. The programs are available to the general public. If you are interested in getting a copy of them, you may do so, free of charge, by contacting my advisor, Dr. David Mutchler. Please send electronic mail to mutchler@utkcs2.utk.cs.edu on Internet, or US Mail to David Mutchler, Computer Science Department, The University of Tennessee, Knoxville TN, 37996-1301.

Our system has the following major characteristics:

1. The programs were written in C-Shell for portability and ease of modifications.

2. Machines communicate using only the rsh and the rcp commands.

3. Each user login name, machine name pair is considered to be a node. This allows several users of the same machine to share a replicated file without having to change its access mode.

4. At every node, the user (perhaps aided by the system's manager), must provide a very simple program (barely a case statement) to map machine nicknames to valid host names, i.e. to the strings to be used as the host name in a rsh command.

5. If any of the nodes share the file server with other machines, the function described above (3) must return the name of the file server when passed the name of any of the machines that share it.

6. The base names of each node's copies of the replicated files must be unique. And the base name of each copy of a replicated file must be the same in all the nodes.

7. There are no restrictions on the subdirectory structure of each node.

8. Each node keeps a subdirectory of symbolic links. A link is maintained for each replicated file. This way the copies of the replicated files may be moved within the same file system without having to inform other nodes about the changes.

9. Each node also keeps a list of the absolute names of the replicated files. This list is used by new versions of the mv, rm and chmod commands to protect replicated files from being destroyed, modified, locked, etc.

10. Write access is removed from the replicated files while they are not being used. Read access is never removed. The new version of the chmod commands enforces these restrictions.

11. All the commands that may update a replicated file are aliased with the name of a program that takes care of the unlocking and locking procedures. This way the access to the replicated files is totally transparent.

12. Static weighted voting with no read quorums is used to maintain the consistency. One copy serializability is maintained only if the user treats every read access as a write operation.

13. Plain replicated files are viewed as a whole. Only one version number is kept for each copy of a replicated file, and the whole file (not just the updated portions) is transmitted after an update.

14. The number of files as well as the hierarchy of the files in a replicated directory is forced to be consistent among all the copies.

15. Replicated files cannot be moved into replicated directories. A directory cannot be replicated if it contains replicated files.

16. Replicated subdirectories are allowed to contain symbolic links, which are not resolved when they are copied from one node to another.

17. All the sequences of rsh commands are executed in parallel.

18. The parallel execution of the rsh commands necessary to collect the quorum is stopped when the majority of votes is collected even if not all the rsh commands are finished.

19. The modification time stamps kept by Unix in the directory entries are used to detect the updates to the replicated files.

20. The time stamps of the copies of each replicated file, as well as the access modes of the files in a replicated directory are not preserved.

21. Better efficiency is achieved if the replicated directories contain a small number of files, or a large number of files distributed over self contained subdirectories.

22. The system is able to handle witnesses.

BIBLIOGRAPHY

[Barbara 86] D. and Hector Garcia-Molina, "The Vulnerability of Vote Assignments", ACM Transactions on Computer Systems, 4(3), pp. 187-213, 1986.

[Bernstein 87] Philip A., Hadzilacos Vassos, and Nathan Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley Publishing Company, 1987.

[Brender 68a] David M., "The Prediction and Measurement of System Availability: A Bayesian Treatment", IEEE Transactions on Reliability, Vol R-17 No. 3, pp. 127-138, September 1968.

[Brender 68b] David M., "The Bayesian Assessment of System Availability: Advanced Applications and Techniques", IEEE Transactionson Reliability, Vol R-17 No. 3, pp. 138-147, September 1968.

[Carroll 87] John L., D. E. Darrell and Jehan-Francois Paris,"Block-Level Consistency of Replicated Files", Proceedings of the IEEE 7th International Conference on Distributed Computing Systems, 1987.

[Davcev 85] D. and W. Burkhard, "Consistency and Recovery Control for Replicated Databases", Proceedings of the 10th ACM Symposium on Operating Systems Principles, pp. 87-96, 1985.

[Davidson 85] Susan B. and Hector Garcia-Molina,"Consistency in Partitioned Networks" ACM computer surveys, 17(3) pp. 341-370. 1985.

[Eager 83] D. L. and K. C. Sevcik, " Achieving Robustness in Distributed Database Systems", ACM Transactions on Database Systems, 8(3) pp. 354-381, 1983.

[El Abbadi 85] A., D. Skeen and F. Christian, "An Efficient, Fault-Tolerant Protocol for Replicated Data Management", Proceedings of the 4th ACM Symposium on Principles of Database Systems", pp. 215-228, 1985.

[El Abbadi 86] A. and S. Toueng, "Availability in Partitioned Replicated Databases", Proceedings of the 5th ACM Symposium on Principles of Database Systems, pp. 240-251, 1986.

[Garcia-Molina 82] Hector, "Reliability Issues for Fully Replicated Distributed Databases" IEEE Computer magazine, 15(9), pp. 34-42, September 1982.

[Garcia-Molina 85] Hector. and D. Barbara, "How to Assign Votes in a Distributed System", Journal of ACM 32(4), pp. 841-860, 1985.

[Gifford 79] D. K., "Weighted Voting for Replicated Data", Proceedings of the 7th ACM SIGOPS Symposium on Operating Systems Principles, pp. 150-162, December 1979.

[Jajodia 87] Sushil and David Mutchler, "Dynamic Voting", Proceedings of ACM SIGMOD, pp. 227-238, 1987.

Jajodia, Sushil and David Mutchler, "Dynamic Voting Algorithms for Maintaining the Consistency of Replicated Data Bases", Accepted for publication by ACM transactions on Database Systems.

Joy, William, "An Introduction to the C Shell" University of California, Berkeley

[Metzner 82] John J., "A Proposed Parity Structure for Large Remotely-Duplicate Data Files", Proceedings of the IEEE 3rd International Conference on Distributed Computing Systems, 1982.

[Paris 86] Jehan-Francois, "Voting with Witnesses: A Consistency Scheme for Replicated Files", Proceedings of the IEEE 6th International Conference on Distributed Computing Systems, pp. 606-612, 1986.

[Thomas 79] R. H., "A Majority Consensus Approach for Concurrency Control for Multiple Copy Data Bases", ACM Transactions on Database Systems, 4(2) pp. 180-209, June 1979.

APPENDIX

SURFS: A Replicated File System for Unix

Raymundo Diaz Rodriguez
Computer Science Department
The University Of Tennessee

Introduction

Hardware and/or software malfunctions affect the availability and reliability of a file. A replicated file system increases the availability and the reliability of the files by keeping several copies of them, placing each copy in a different computer system. A strong replicated file system should also maintain the consistency of the copies of each file.

SURFS is a set of programs that work together to maintain the consistency of replicated files and/or subdirectories among Unix systems. SURFS was designed to be used in an academic or similar environment, primarily for text files (however it may be used in other Unix environments on any kind of files). SURFS provides transparent access to the replicated files i.e. they can be treated as regular files, thus the user's programs and/or system's commands do not need to be modified.

The design and implementation of SURFS were presented as a masters thesis. The idea was not to provide a new algorithm to maintain the consistency of replicated files, but to produce a user-oriented, transparent implementation of such an algorithm. The strength of SURFS is based on its transparency, portability and ease of installation, as well as the fulfillment of the obvious requirements for a replicated file system. This manual is not a technical reference, and is intended for readers with some knowledge about Unix and the C-Shell.

For further details, the reader is encouraged to consult the author's thesis. Inquiries should be directed to the author's thesis advisor, Dr. David Mutchler, by electronic mail to mutchler@utkcs2.cs.utk.edu on Internet, or by US Mail to David Mutchler, Computer Science Department, The University of Tennessee, Knoxville TN, 37996-1301.

Before You Start

Although SURFS (Single User Replicated File System) is designed to be used by just one person, it also allows several users to share replicated files at one or more sites. However, each member of this group of users must have complete trust in each other, since (because of the way that SURFS works) every one of them will not only have access to the replicated files, but to all the files of every person in the group (from an abstract point of view, such a group of users may be considered as the same person with different login names on one or more machines). Such user equivalence is defined in the ".rhosts" file; the reader may find more about this in the Unix manual pages for the rlogin command.

The first thing that you must do is to create your private equivalence list (i.e. the ~/.rhosts file) used by the rlogin and rsh commands. This should be done at each of the nodes that are going to be used to replicate files. Be sure that rlogin (or rsh) works from each node to any other node in every subnetwork where you expect to replicate files (see the question about partially connected networks in the last section of this manual). If you want to replicate files with some one else, or if your login name is different in one or more nodes, rlogin should not prompt for a password (see the Unix manual for rlogin).

Because of the way that Unix assigns the names to the remote hosts, SURFS requires you to write a very simple program called rep-hostname. Probably the best thing to do is to acquire the program from someone else who is already using SURFS in the same system.

The characteristics of this program are:

1. When it is called with no arguments, it should return the nickname of the local node. This nickname does not need to be valid; i.e. it is not going to be used by the rsh command.

2. Otherwise, it should "translate" the nickname of the node, given as its only argument, to the valid node name i.e. the string to be used by "rsh".

3. In the nodes that share the same file server (for example, a node mounted on the Sun's NFS), it should return the name of the file server when the argument is the nickname of any of the nodes that share the file server.

In many cases (if no nodes use a shared file server), the string returned by the hostname command uniquely identifies each node of the network, and also, can be used by

rsh at all the other nodes. In such cases, rep-hostname should be equivalent to (or actually) this C-Shell script:

```
#! /bin/csh -f
if ( $#argv ) then
        echo "$1"
else
        /bin/hostname
endif
```

However, if any of the nodes use a shared file server, or if two or more nodes in the network to be used to replicate files require different names to refer to the same node, then you should select your favorite unique nickname for each node and write rep-hostname in such a way that it returns the valid name corresponding to the nickname passed as an argument. To avoid confusion, let us consider an example. Suppose that you want to replicate files among three nodes, namely draco, utkux1 and utkcs2. Suppose that draco is mounted on the Sun's NFS (shares the file server with other machines), and that the name of the file server is alphard. Also, suppose that these strings may all be used with an rsh command in any of the three nodes. In this case both utkux1 and utkcs2 may use the version of rep-hostname presented above, but draco's version should be equivalent to:

```
#! /bin/csh -f
switch ( "$1" )
        case utk*:
                echo "$1"
                breaksw
        default:
                echo "alphard"
endsw
```

Notice that the default is the name of the file server, since we do not know the names of all the machines that share it, which could use SURFS (assuming that none of the names of these machines begin with "utk").

Using the same example, suppose utkux1 does not know utkcs2, but knows cs2, i.e. cs2 is the name that utkux1 gives to the machine that draco knows as utkcs2. Furthermore suppose utkux1 does not know alphard, but knows foo. Then utkux1's version of rep-hostname would look like:

```
#! /bin/csh -f
switch ( "$1" )
        case utkcs2:
                echo "cs2"
                breaksw
        case alphard:
                echo "foo"
                breaksw
        default:
                echo "utkux1"
endsw
```

Studying the /etc/hosts file may give you an idea of how to name each node. However "yellow pages" and/or other gadgets may be used to resolve node names in your environment; if the SURFS programs give you error messages such as "ERROR: rep-hostname returns invalid answer" or "ERROR: duplicate node names" while you are mounting the files, consult your system manager.

If any of the mounted nodes uses a shared file server, you must write a program called "rep-pwd". This program should work as the regular pwd command, but must return the same path (for the same file), regardless of the machine (of those that share the file server) where it is executed. Also the returned path must be valid in all the machines that share the file server.

In the particular case of UTK Computer Science Department's Sun work station network, if the user's home directory is on alphard (i.e. if the name of the shared file server is alphard) the pwd command returns "/home/USER" when it is executed on alphard, and "/n/alphard/root/home/USER" in any of the other work stations. Since "/home/USER" is only valid on alphard, rep-pwd should look like this (although this details are installation dependent):

```
#! /bin/csh -f
if ( `/bin/hostname` == "alphard" ) then
        echo  "/n/alphard/root"`/bin/pwd`
else
        /bin/pwd
endif
```

Installing SURFS

You are now ready to install the software. You must follow this procedure at each node that is going to be used to replicate files.

1. If you have your own versions of the commands rm, mv and/or chmod, you must rename them. For example: my-rm, my-mv and my-chmod.

2. Place the SURFS programs in any of the subdirectories stated by your path variable (see the Unix manual for csh), usually on "~/bin". SURFS's versions of chmod, mv and rm must be placed on a subdirectory that is defined before "/bin" in your path variable. You may need to modify the setting of this variable in your ".cshrc" file.

3. Add your version of rep-hostname to any subdirectory stated by your path variable.

4. If the node uses a shared file server, also add your version of rep-pwd. If not, create a link to the pwd command in any of the subdirectories defined before "/bin" on your $path variable, and name it "rep-pwd". For example:

```
ln -s /bin/pwd ~/bin/rep-pwd
```

5. SURFS requires the user to have a subdirectory in his home directory, named ".repinfo". It uses this subdirectory to store all the information it needs to know about the replicated files. Executing rep-install will create this subdirectory, and some files that must be there. SURFS trusts you and believes that you are never going to alter the contents of this subdirectory.

6. SURFS versions of chmod, rm and mv execute the corresponding commands found in the /bin subdirectory. If you do not want them to work this way (i.e. you have your own versions), it is very easy to modify them. The first word of the last line of these C-Shell scripts is the name of the command to execute. For example, in the case of rm, the first word of its last line is "/bin/rm", you may change this to something like "~/bin/my-rm".

Your ".cshrc" file should not create a non standard environment for the rsh command. SURFS will not work properly if this file generates some output, or defines strange aliases, sets C-Shell variables (such as noglob), etc. Barely, all that SURFS needs is the settings of the "path" and the "umask" variables. However, you are free to define your interactive environment as you wish (except for the position of "/bin" in the definition of the path variable).

Mounting Files

Once you have gone through the installation procedure, you must tell SURFS which files are going to be replicated. SURFS allows you to mount only one file at a time.

If you are planning to replicate a lot of files, it may be better if you group them in subdirectories before you mount them. The procedure to follow every time that you want to add another replicated file or directory is the following:

1. Login to the node that has the file to be mounted. Select a name for the replicated file; it must not start with a - or contain metacharacters. Once the file is mounted you cannot rename it (however you may move it to other non-replicated subdirectories).

2. The file must not already exist in the nodes to be mounted. For example, if you want to replicate "~/foo" which is at node A, on nodes A, B and C, and you want to place B's copy in a subdirectory called "~smith/Bdir" and C's copy in "~john", then subdirectories "~smith/Bdir" and "~john" must not contain a file named "foo". SURFS refuses to overwrite existing files; rep-mount will give you an error message in this situation.

3. Be sure that the file's base name (i.e. the string on the right of the last / of the absolute name) is not equal to the base name of any of the files that are already being replicated in the nodes to be mounted (rep-info executed with no arguments gives you the absolute names of the files that are being replicated).

4. To execute rep-mount you must give it the file name and the list of nodes where it is to be mounted. Each node is a machine/user pair (a slash "/" must be used to separate them). If no user name is given your local login name will be used. For our previous example:

rep-mount foo B/smith C/john

5. Rep-mount will prompt for the names of the subdirectories where each copy of the file is to be placed. For our example, when it prompts for B's subdirectory you should reply "~/Bdir", "Bdir" or "~smith/Bdir". Similarly for C, the answer could be "~". If a node is to be a witness, your reply to this question must be "/witness" (see the section about witnesses).

6. Each copy of a file is assigned a number of votes, and only those nodes that can collect a majority of votes may modify the replicated files. Rep-mount will prompt you for the number of votes to be assigned to each node.

All the involved nodes must be available when rep-mount is executed, otherwise it will fail. The order in which you give the node list is important. You should sort the list according to these (perhaps conflicting) criteria:

a. The names of the witnesses must be the last ones.

b. Sorting from slowest to fastest (in terms of the response time of rsh) may improve the response time of rep-unlock if the votes are fairly assigned.

c. If some nodes have a significant advantage on the number of votes, putting them at the beginning of the list may also improve the response time.

If every thing is OK, the file will be ready to be used in a few minutes after rep-mount is done. It may also take a few minutes (after rep-mount is done) before the data arrives to the other nodes.

There are many things that may make rep-mount to fail, explaining all these situations will take too long. In any case, rep-mount always gives explanatory error messages. Just one final advice: if rep-mount decides to abort, you should not kill it since that may scramble the contents of the ~/.repinfo directory at one or more nodes. In general you may use control-C to interrupt SURFS, but you should never try to kill any of its programs.

Access to Replicated Files

The owner always has read access to the replicated files. The fact that you may read a replicated file does not imply that you are looking at its most updated version. Rep-unlock must be executed first, if you want to update the file, or to be sure that the copy is up to date. Rep-unlock is executed transparently using rep-write and aliases, as described in the next section. Probably you will never have to execute rep-unlock and/or rep-lock therefore you may want to skip the rest of this section.

Rep-unlock receives the name of the replicated file to be unlocked. Since the base name of a replicated file is unique, you may execute rep-unlock on any subdirectory without giving the absolute name of the file to be unlocked, i.e. to unlock ~/foo you may issue the command "rep-unlock foo" while your current working directory is not "~". Rep-unlock may also receive the "-wait" and "-next" options explained below.

What rep-unlock does is to communicate with the other nodes that have a copy of the file to be unlocked, and decides if you may update it. If it determines (by voting) that you may in fact update the file, it brings the most updated copy (if the node's copy is out of date), and it changes the access mode giving you write permission. If you change your mind about unlocking a file, it is safe to interrupt rep-unlock while it is running by

hitting control-C (which will start some background processes). Rep-unlock clears the screen and tells you the reasons why it cannot unlock the file (if that is the case).

If rep-unlock is successful, you will be given write access to the replicated file. Then, to lock the file and to update the other nodes (if necessary), you must execute "rep-lock". As with rep-unlock you do not have to give the absolute name of the file to be locked, etc. But rep-lock only takes the "-wait" option (explained below) and it refuses to be interrupted. If the -wait option is used, rep-lock will blindly believe that the same option was used with rep-unlock. Lying to rep-lock, i.e. using the -wait option when you did not with rep-unlock or vice versa, will not only add some overhead, but may even affect the file's consistency.

Rep-lock checks if the file was updated, and locks the file by removing write permission. If the file was in fact updated, it will ask the other nodes to update themselves. Both rep-lock and rep-unlock may only work with one replicated file at a time.

Achieving Transparency

Besides not being transparent, using rep-unlock and rep-lock may be undesirable, since you may forget to use rep-lock, after you are done. It is better to alias the commands that you expect to use to access the replicated files in your "~/.cshrc" file. Rep-write should be used to alias these commands, here are some examples:

```
alias vi      rep-write -next vi
alias safevi rep-write -next -wait vi
alias cat     rep-write -wait cat
```

What rep-write does is to check the argument list looking for replicated file names. If none are found, it executes the given command normally. Otherwise, it will "jump to the background", and execute the given command on the foreground. While rep-write is running in the background it executes rep-unlock with each of the given replicated files. When the command is done rep-write automatically takes care of the locking.

The default behavior of rep-write has a serious problem. In our example, we want the command "cat" to output the most updated version of the replicated files. If rep-write executes "cat" and "rep-unlock" in parallel, on the foreground and the

background respectively, "cat" will be done before the file is unlocked and may output an old version of the replicated files. You must use the -wait option in any of these situations:

1. The command expects to read the most updated version of the replicated files.
2. The command needs to have write access to the files before it is executed.
3. The node is out of date.

Only the last two cases apply when using the -wait option with rep-unlock. It is quite probable that you must use this option if more than one file is to be unlocked by rep-write, this way the command will not be executed unless all the files are successfully unlocked.

In the case of "vi", although it is not an error to execute it on files with no write access, it is not prepared to modify such files even if the write access mode changes (after vi started). For example, if you type "vi foo" and "foo" is a replicated file, "vi" will be executed on the foreground, it will read the file while rep-write has not given write access. Eventually, rep-write will change the access mode (hopefully), while you are still editing the file. At this point if you give vi the command ":w" to save the changes, it will give you an error message, since it still "thinks" that you do not have write access. Vi does not check the current access mode of the file unless you issue the command "set noreadonly", (which also affects the view command) or use ":w!". You may add this set command in the ".exrc" file or to the EXINIT variable (see the vi manual). In this case we were able to "fix" vi; if you cannot do the same thing with the program(s) that you want to use to access the replicated files, you must use the -wait option.

SURFS uses time stamps to detect the updates to the replicated files, however it only uses the date, hour and minutes. If a program may update the file in just a few seconds, and the rsh commands may also be executed very fast, you (or rep-write) may be able to unlock, update and lock the file, in the same minute. Under this circumstance the consistency of the file may be lost, since SURFS would not be able to detect the update. Using the "-next" option with rep-unlock, and for that matter with rep-write, makes it to wait until the next minute (according the the value of the clock when the quorum collection started) before it gives you write access. It is not necessary to use this option with commands that cannot update a file, or if the command takes at least 60 seconds to run.

SURFS does not maintain the consistency of the time stamps of the copies of each replicated file. SURFS does not respect the "umask" settings of each node, instead the

access modes are preserved (this is not true for replicated directories). The limit on the number of files that you may replicate is given by your disk quota.

Access to Replicated Subdirectories

SURFS also allows you to replicate subdirectories. There is no specific limit on the size and or number of files in the directory (besides your disk quota). Symbolic links that are part of a replicated directory are not resolved; even if they are meaningless on one or more nodes. SURFS does not consider this to be an error; all it worries about is to keep the link values identical. The time stamps of the files in a replicated directory are not preserved. SURFS respects the "umask" settings of each node, it does not maintain the consistency of the access modes of all the files in a replicated directory.

Using rep-write to modify files in replicated subdirectories is done exactly as for replicated plain files. Using the example for "vi" (given in the previous section), if you try to edit a file that belongs to a replicated subdirectory, rep-write will attempt to unlock it. However, SURFS "thinks" of each replicated subdirectory as a whole; there is no way to unlock parts of the subdirectory. Thus using rep-write to unlock a file in a replicated directory actually unlocks all the files in that directory.

Both rep-lock and rep-unlock may be used with replicated directories, however you must use the name of the subdirectory, not the name of a file in it.

Dismounting a Replicated File

Rep-dismount is used when you no longer want to replicate a file. SURFS's version of rm refuses to remove replicated files, and to avoid problems you should not attempt to remove them using /bin/rm. Replicated files must be removed one at a time and all the nodes that contain a copy of the replicated file must be up when rep-dismount is executed.

As with rep-lock and rep-unlock, rep-dismount should be called with the base name (string on the right of the last "/") of the replicated file to remove. If rep-dismount succeeds, all the copies of the replicated file are removed, except for the copy that is on the node that executes it. Rep-dismount fails when it is executed on a witness

(see below). If it is executed on a node with an out of date copy, it updates the copy before dismounting the file.

Using Witnesses

If you have a limited disk quota on any of the nodes, or if a node does not know some of the other nodes, but the other nodes know it (i.e. the other nodes can execute the rsh command, but that node cannot), it may be a good idea to make that node a witness for the replication of the files.

Nodes that are witnesses contain no copy of the file, but can "testify" to the current state of the file. In other words a witness only keeps the version number of the replicated file. Since witnesses do not contain a copy of the file, SURFS does not allow the user to update a replicated file from a witness, and obviously out of date nodes cannot get the most updated copy of a replicated file from a witness. For these reasons the order in which you give the node names to rep-mount is very important. When rep-unlock is executed in an out of date node, it will look for the most updated copy by seeking the version number of the other nodes. The order in which it asks is defined by the order in which you gave the node names to rep-mount, therefore the names of the witnesses must be the last arguments to rep-mount.

SURFS learns that a node is a witness if you tell rep-mount to put the copy of the replicated file in a non existent subdirectory called "/witness". A witness does not need to know any of the other nodes, but all the nodes that contain a copy of a replicated file, must know the witnesses.

Command Summary

1. chmod: Equivalent to the Unix chmod command (with the System V "-R" and "-f" options). Protects replicated files from being illegally locked or unlocked.

2. mv: Equivalent to the Unix mv command. Protects replicated files from being illegally moved or renamed. You cannot change the relative name of a replicated file, however you can move them to another subdirectory using rep-mv.

3. rm: Equivalent to the Unix rm command. Protects replicated files from being illegally removed. Rep-dismount must be used before removing a replicated file.

4. rep-abort: NOT TO BE EXECUTED BY THE USER. Removes all information about a replicated file from the "~/.repinfo" subdirectory without asking for permission or printing error messages. Receives one or more arguments, the relative name of the replicated file, and the list of nodes where ~/.repinfo must be updated. If no node names are given, it will try to remove the local copy of the file. Otherwise it will try to remove the remote copies of the file, leaving the local copy untouched.

5. rep-dismount: Receives the base name of the replicated file to be dismounted. Updates the "~/.repinfo" directory of the affected nodes, and leaves the most updated copy on the node that executes it.

6. rep-hostname: This program is provided by the user. When no arguments are given it should return the nickname of the node that executes it. Otherwise, it should "translate" the given nickname (its only argument) to a valid host name to be used by the rsh command. If given the name of a node that uses a shared file server (such as Sun's NFS), it should return the name of the file server.

7. rep-info: When executed with no arguments it writes to the standard output the absolute names of all the copies of the replicated files available at the local node. If one or more arguments are given, it takes them as the names of replicated files and prints some information about each one.

8. rep-install: Creates the "~/.repinfo" subdirectory, and checks the "path" variable, receives no arguments.

9. rep-lock: Not usually executed by the user since files may transparently be locked by rep-write. Receives the base name of the file to be locked, it asks the other nodes to execute rep-update, and locks the file.

10. rep-mount: Starts the replication of a file. Receives the file name, and a list of node nickname, login name pairs each separated by a /, where the copies of the replicated file are to be placed. It prompts the user for other information and prints explicit error messages if it fails.

11. rep-mv: Used to move replicated files from one subdirectory to another, receives the name of the replicated file to be moved and the name of the subdirectory where it is to be placed. Replicated files cannot be moved into replicated subdirectories. Replicated files must be moved one at a time.

12. rep-pwd: Usually a symbolic link to /bin/pwd. It should be a program equivalent to the Unix pwd command, however it should return the same valid path (for the same file) when executed in any of the machines that share a file server.

13. rep-unlock: Not usually executed by the user since rep-write transparently unlocks the files. Usage: rep-unlock [-w] [-n] file. If the -w option is used, it believes that you are not going to update the file until rep-unlock is done. If the -n option is used "fast updates" cannot go undetected, reducing the probability of reaching inconsistency.

14. rep-update: NOT TO BE EXECUTED BY THE USER (however it can be used to do a manual reconfiguration of an inconsistent replicated file, see the comments of the program). Receives three arguments, the name of the node (host.user) with the most updated copy, the relative name of the replicated file to update, and the version number of the must updated copy. It refuses to update the node if the third argument is less than local version number of the replicated file. This program uses rcp to copy plain files, and a combination of tar and rsh to copy subdirectories. Only the updated portions of the replicated subdirectories are sent through the network. It may receive two options: "-force" "-nolocking". If the "-f" option is used it does not check the version numbers i.e. it may overwrite the local copy with an old one. Rep-update assumes that the file is unlocked before it is executed, therefore it locks the file unless the "-n" option is used.

15. rep-verify: This program is designed to be executed by rep-mount. It receives the name of a file and the name of a subdirectory, and writes to the standard output the reasons why that file cannot be replicated if the copy is to be added to that subdirectory. It may be useful to execute in combination with the rsh command before mounting a file.

16. rep-write: usage: rep-write [-w] [-n] command args. Provides transparent access to the replicated files. The options are passed to rep-unlock. Read the section "Access to replicated files".

Technical Information

Although you should not be concerned about the contents of the ~/.repinfo, it may be helpful to know this in case you find an error i

n the programs or if you wish to modify them. This subdirectory contains a text file called LOCALFILES. Each line of this file is the absolute name of the local copy of a

replicated file. ~/.repinfo also contains four subdirectories: VOTES, UPDATES, LINKS and TEMPFILES.

The VOTES subdirectory contains one file for each replicated file. Each line of these files has two words, the first word is a VALID node name and login name joined by a dot, the second word always begins with a 0 and is an integer number indicating the number of votes assigned to that node. The name of each file in VOTES is the same as the base name of the corresponding replicated file.

UPDATES also contains one file for each replicated file, the name of each of these files matches this pattern *.%.*. Here the string on the left of the ".%." sequence corresponds to the replicated file's base name, and the string on the right is the file's version number. The contents of these files are irrelevant for replicated plain files, and in the case of replicated subdirectories, each line contains the name of the files and or subdirectories (relative to the replicated subdirectory) that were updated from the previous to the current version. The user's execute access of the files in the UPDATES subdirectory is used as a flag to indicate if the file is locked or unlocked.

The LINKS subdirectory contains one symbolic link for each replicated file, the name of the link is equal to the file's base name.

TEMPFILES also has files that match *.%.*; these files usually contain error messages, and the string on the right of the ".%." indicates how they were created. TEMPFILES should be empty if all the replicated files have been and are inactive.

Questions and Answers

1. How should the votes be assigned? The number of votes should be a function of the node's failure rate, its links failure rates and the expected update rate to that particular replicated file in that node. If you are not familiar with replicated file theory, and cannot get help from someone else, here are some heuristics. Assign each node the same number of votes if the demand for the file is evenly distributed. If you expect the file to be updated at the same node most of the time, give that node an advantage (more votes). In all cases, the total number of votes should be an odd number; if not give one site one more vote.

2. SURFS is doing "funny" things? Although it has been tested, the software may still have some bugs. Also strange states may be reached if any of programs gets killed.

Probably the best strategy is to manually reconfigure ~/.repinfo at each node, if its contents are not as described above. Rep-info may be very useful to check what is going on.

3. If the nodes on the network are already replicating some files, how can a new node be added? Follow the installation procedure at that node. Unfortunately, there is no way to tell SURFS that there are already some files to replicate; you may add new replicated files to that node, but you cannot add a copy of an existing replicated file to that node. The only way around this is to dismount the file from the other nodes (do not attempt to dismount two files at the same time!!!), and mount it again on this and the other nodes. You may also manually reconfigure ~/.repinfo at all the nodes . . . if you are brave.

4. How can a node be dismounted? First you must dismount all the replicated files that are available at this node. Be careful when you execute rep-dismount, remember that only one copy of the replicated file is left on the node that executes rep-dismount. Once the node does not contain any copies of the replicated files, you may safely remove ~/.repinfo and SURFS's software using the rm command.

5. Can I replicate my home directory? Unfortunately no. Any directory that contains ~/.repinfo cannot be replicated.

6. How can I make rep-update resolve symbolic links in a replicated subdirectory? Modify the call to the tar command in that script; tar has an option that makes it follow symbolic links. However, if you do not modify rm, mv and/or chmod the files pointed to by the links will not be protected.

7. What happens if I logout immediately after rep-write is done? Nothing, if you check the last few lines of rep-lock, you will see that the output is redirected to /dev/null, thus you cannot know if the remote nodes were able to update themselves. You may modify this, but if you do each "rsh" will not die until rep-update is done at the remote nodes. This will leave your machine with two (or more) extra processes for each node until rep-lock is done, which may annoy the system's manager and/or affect the machine response time.

8. What about partially connected networks? SURFS is able to work on partially connected networks. Not all the nodes must "know" the rest, for example, if you want to install SURFS on six nodes (A, B, C, D, E and F) and suppose that A only knows B and C, i.e. there is no way to execute the rsh command to D, E or F. In this case SURFS allows you to replicate a different set of files on each of the two fully connected subnetworks, A-B-C and B-C-D-E-F at the same time. You can also define your own subnetworks, for

example, each one of B-C-D, E-F, B-E-F, etc. may also share a different set of replicated files. Thus different sets of files may be shared by different sets of users; the only restriction is that the absolute names of the replicated files at each node must be unique, whether or not they are being replicated on the same subnetwork (see the section about witnesses).

Known Bugs

Mounting and/or dismounting two or more files at the same time anywhere on the network may destroy the contents of ~/.repinfo/LOCALFILES.

SURFS is confused by ".cshrc" files that generate some output, it usually thinks of this as an rsh failure. It may also be confused if this file creates a strange environment for the rsh command.

The SURFS version of chmod does not accept absolute modes if any of the arguments is a replicated file.

If rep-mount fails, a file may be left on the ~/.repinfo/TEMPFILES of the other nodes.

A new version of rmdir is not provided. Hopefully replicated directories will never be empty.

The tar command combined with the rsh command are used to transmit directories. Tar is executed by the remote nodes on the most updated node, thus there may be as many executions of tar as the number of nodes, which creates a bottleneck and may affect the response time. However rep-lock does not start rep-update on the other nodes, strictly on parallel, and the speed of the links and the response time of the nodes is usually different, therefore the other nodes will start the execution of the tar command at different times. This problem may be fixed by replacing the contents of the files in ~/.repinfo/UPDATES with the output of tar.

The following is not an actual bug, but it may confuse the user. If a node fails while rep-mount is running and you do not get an abort message, the copies of the node(s) that failed will be empty until they are repaired and the file is updated.

VITA

Raymundo Diaz Rodriguez was born in Monterrey, Nuevo Leon, Mexico on September 17, 1964. He attended elementary schools in that city and was graduated from the Universidad de Monterrey's High School in 1981. Immediately after, he entered Instituto Tecnologico y de Estudios Superiores de Monterrey, where he received the degree of Ingeniero en Sistemas Electronicos (Electronic Systems Engineer), in 1986.

During his college years he was employed by the computer center of his university, as a programmer assistant to develop and maintain software for a computer assisted registration system based on a network of microcomputers. Also during these years, he took an intensive English course at the EF Institute, in Cambridge England.

In 1987 he accepted a teaching and research assistantship at the Instituto Tecnologico y de Estudios Superiores de Monterrey, and began study toward a Master's degree. Later, on the same year, he turned down that offer, and accepted a teaching assistantship position at the University of Tennessee, Knoxville, and continued studying toward a Master's degree in Computer Science. This degree was awarded in August 1989.