Rochester Institute of Technology
RIT Scholar Works

Theses

1987

A simulation of a distributed file system

Alan Stanley

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation
Stanley, Alan, "A simulation of a distributed file system" (1987). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Rochester Institute of Technology
School of Computer Science and Technology

A Simulation of a Distributed File System

by
Alan L. Stanley

A thesis, submitted to
the faculty of the School of Computer Science and Tecnology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: ______________________________
Professor James E. Heliotis

____
Professor Andrew Kitchen

____
Professor Peter Lutz

A Simulation of a Distributed File System

Alan L. Stanley

Abstract

This thesis presents a simulation of a distributed file system. It is a simplified version of the distributed file system found in the LOCUS distributed operating system. The simulation models a network of multiuser computers of any configuration. The number of sites in the network can range from a minimum of three sites to a maximum of twenty. A simple database management system is supported that allows the creation of an indexed database for reading and updating records. The distributed file system supports a transaction mechanism, record level locking, file replication and update propagation, and network transparency. To test the effect of site failures and network partitioning on the distributed file system, a facility is provided to "crash", "reboot", and "jump to" random sites in the network.

November 23, 1987

Computing Review Subject Codes

File Systems Management / Distributed File Systems (CR D.4.3)

Distributed Systems / Network Operating Systems (CR C.2.4)

Table of Contents

1. Introduction 1

  1.1. Distributed File System Definition 1

  1.2. Overview of Dissertation 1

2. Overview of the LOCUS Operating System's Distributed File System 2

3. Project Description 5

  3.1. Program Design 6

    3.1.1. Network and Site Specification 6

    3.1.2. Site Communication and Message Forwarding 8

    3.1.3. File Replication and Distribution 12

    3.1.4. File and Record Locking 15

    3.1.5. Transaction Mechanism 18

    3.1.6. User Site Program 20

    3.1.7. Storage Site Program 23

    3.1.8. Current Synchronization Site Program 24

  3.2. Network Testing Facility 26

    3.2.1. Demo User Site Specification 26

    3.2.2. Site Crashing Mechanism 28

    3.2.3. Site Rebooting Mechanism 29

4. Project Testing 29

    4.1. Test Plan and Procedures 29

    4.2. Problems Encountered and System Limitations 31

    4.3. Project Test Results 34

5. Conclusions 36

Appendix 37

    A. User's Guide 38

    B. Test Plan 48

    C. Simulation vs. LOCUS 56

    D. Glossary 58

References 59

1. Introduction

The proliferation of minicomputers, microcomputers, and personal workstations that have replaced the large mainframes, along with the many large databases around the nation, has required the development of distributed file systems. Early attempts at developing such systems resulted in slow, inefficient, and very user unfriendly systems. Information was passed between processors, each with its own operating system. Today there are only a handful of commercially available distributed file systems, while others are still in the development stages.

1.1. Distributed File System Definition

A distributed file system provides users with the ability to access files and the individual records within the files exclusively, atomically, and transparently. Exclusive access to one or more records within a file is typically achieved by a locking mechanism with record level granularity. Files can be locked either with a shared lock,which allows multiple readers and blocks writers, or with an exclusive lock, which blocks all other readers and writers.

Atomicity is typically achieved by the use of a transaction mechanism. All updates performed within a transaction are guaranteed to be applied or none of them are. For example, if a site failure in the network causes a transaction to be interrupted, all previous updates are discarded. Updates are only guaranteed to be applied when a transaction commit acknowledgement is received. Updates can also be forcibly discarded by an abort command.

The network is transparent to the users because the distributed file system does all the work. Users do not need to know where in the system a file is located. Users only become aware of the system when a site failure interrupts their processing.

1.2. Overview of Dissertation

The distributed file system simulation described in this paper is based on the distributed file system utility of the LOCUS distributed operating system. An overview of the distributed file system utility of LOCUS is presented in chapter 2. Chapter 3 describes the program design and implementation and the network testing facility. Chapter 4 describes the project testing, problems encountered, and the test results. The conclusions are in chapter 5.

2. Overview of the LOCUS Operating System's Distributed File System

The LOCUS operating system, a superset of UNIX™, was developed at the University of California at Los Angeles [POPEK 81], [WALKER 83], [WEINSTEIN 86]. LOCUS is designed to operate on a high band width, low delay, local area network. A token ring network, with a circulating token supported by hardware, is used to control access to the transmission mechanism.

For performance reasons, LOCUS is not based on the ISO model of layers for network software. The ISO model would have the network software implemented in the application layer [TANENBAUM 81], shown in figure 2.1. But, since it has been observed to cost up to 5,000 executed instructions just to move a small collection of data from one user program to the network, the LOCUS software support is located within the operating system at the lower network layer level [POPEK 81].

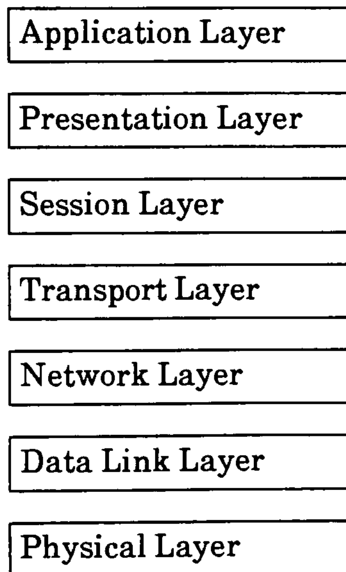


Figure 2.1 ISO/OSI Network Model

LOCUS has proven to be highly reliable [POPEK 81] and it supports network transparency of the system interface so that the user needs no knowledge of the network and its interface code

to communicate with foreign sites. LOCUS also supports transparent replication of files, record level locking, and transaction mechanisms such as commit, abort, and recovery. These attributes of LOCUS are found in its distributed file system and other attributes pertaining to the LOCUS operating system such as remote process creation and execution are not considered here.

Accessing a file in LOCUS involves the interaction of three logical sites: the user site (US), the storage site (SS), and the current synchronization site (CSS). The US is the site requesting a file, the SS is the site with a copy of the file and is selected by the CSS to supply it to the US. The CSS also enforces a global access policy.

When the CSS receives a request from the US to open a file, the CSS chooses the site with the most up-to-date copy to be the SS. Subsequent file requests by the US are handled by the SS. When the US requests the closing of a file, the SS sends this request to the CSS which closes the file and sends a response to the SS, which then passes it on to the US. Figures 2.2 and 2.3 show the LOCUS open and close protocols. The transaction mechanisms of LOCUS [WEINSTEIN 86] are fully implemented on the original base system as described in [POPEK 81] and [WALKER 83]. Transactions in LOCUS are simple-nested. The fully-nested transaction mechanism of [MUELLER 83] is not implemented because of its high implementation and performance costs.

A simple-nested transaction is encompassed by a BeginTrans, EndTrans call pair. All file operations by the process between this pair become permanent unless an AbortTrans call is sent or the SS fails: in which case all updates are discarded. But if the SS fails after the commit signal has been sent to the US by an EndTrans call, the changes are saved until the site (SS) comes back on line, at which time they are committed to the transaction file.

A file locking mechanism is provided by LOCUS with a record level locking granularity. LOCUS enforces a two phase locking mechanism. Records may be locked with either a shared or an exclusive lock. Shared locks allow multiple readers and exclude writers. Exclusive locks are required for writing and provide the lock holder with exclusive access to the locked file region.

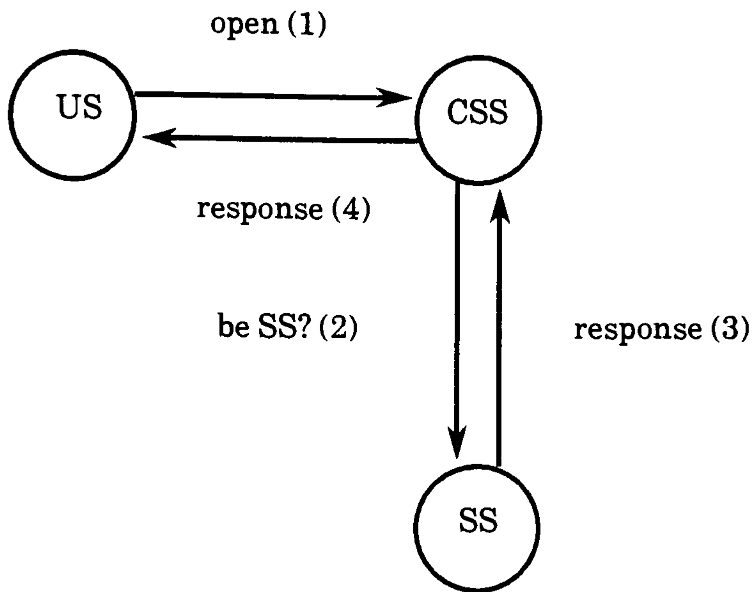


Figure 2.2 Open Protocol

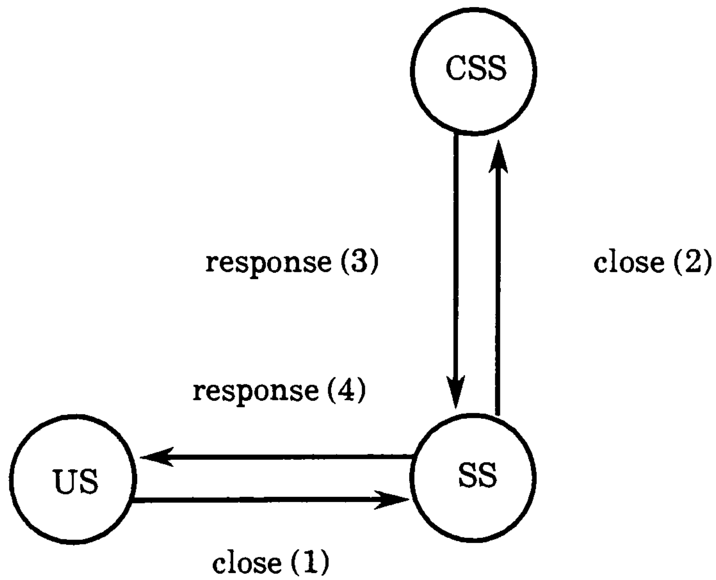


Figure 2.3 Close Protocol

Other distributed operating system designs as supersets of UNIX™ include the Newcastle Connection [BROWNBRIDGE 82], developed at the University of Newcastle-upon-Tyne, England, COCANET UNIX™, developed at the University of California at Berkeley [ROWE 81], S/F-UNIX™, developed at Bell Labs [LUDERER 81], and the Berkeley Remote Mount currently in development at the University of California at Berkeley [HUNTER 85].

3. Project Description

This project is a simulation of a distributed file system (dfs) residing on a local area network. The program was written in the C programming language in a System V UNIX™ environment on a Pyramid 90X computer. The network sites are simulated as processes which are forked from the *demo* program according to the specifications of the network configuration file. Each site process is asynchronous and runs concurrently with the other sites in the network. Communication between the sites is achieved by the use of UNIX™ FIFOs (named pipes).

The LOCUS open and close protocols are implemented as well as other database operations using the US, SS, CSS logical site concept. As with LOCUS, the user needs no knowledge of file location or file replication. Also, the integrity of the system is maintained by the use of record level file locking and by a transaction mechanism. Transaction nesting, however, is not implemented in the simulation.

Recovery from site failures is provided in the simulation. Alternate communication paths are used to reroute messages that are interrupted by site failures. A site failure at a site that is the CSS results in the drafting of another site to become the new CSS when the failure is discovered. Under most circumstances, the file replication mechanism allows the system to access alternate storage sites for a particular file group when one or more of its file images is lost because of a site failure (s).

If a site failure causes the network to become partitioned into two or more subnets, those sites that cannot access the CSS will still be allowed read access to files in their subnet. Each of these subnets designates a site to be a **CSS_rdonly** (CSS read-only). The **CSS_rdonly**

allows files to opened for reading only and only shared locks are allowed on the records within the file.

A testing facility is also provided to evaluate the performance of the simulation and to study the effect of site failures on the simulated distributed file system.

LOCUS provided the protocols and mechanisms used in the simulation. The protocols and mechanisms were used as guidelines in the implementaion since many of the implementation details were not provided by the literature on LOCUS. The network testing facility was designed for the simulation and is not a part of LOCUS.

Refer to Appendix C for a complete summary of the differences between LOCUS and the simulation.

3.1. Program Design

3.1.1. Network and Site Specification

The demo program accepts a previously formatted network configuration file as an argument when executed. If no file is specified, a network configuration file is created. A network configuration file consists of structures of type SITE as shown in figure 3.1. The site field of the SITE structure can hold any integer from 0 to 99 as a name for the site. The **linknum** field holds the number of sites linked to the specified site. The **links** array contains the name of each linked site. A maximum of twenty sites is allowed in a network and each site may have a maximum of nineteen sites linked to it.

```
typedef struct {
    int site;
    int linknum;
    int links[MAXLINKS];
} SITE;
```

Figure 3.1 SITE Data Structure

After the network configuration has been determined, a new asynchronous process is forked and the *site* program is executed. Each new site process is made asynchronously by a double fork algorithm. An asynchronous process is a process that can execute its program and die at

any time without having to be waited for. A process can die without being waited for only if its parent process has previously died. Since the demo process needs to continue executing after creating a site process, an intermediate process has to be forked. The only purpose of the intermediate process is to fork a new process, execute the site program, and then die. The demo process, after waiting only a short time for the intermediate process to die, continues executing, while the new site process is now asynchronous since its parent, now dead, was the intermediate process. Figure 3.2 shows the code for forking an asynchronous process.

```
switch (fork()) {
case -1:
    syserr("fork");
case 0:
    switch (fork()) {
        case -1:
            syserr("fork");
        case 0:
            execlp("./site", "site", "boot", sstr, demo, css, file, NULL);
        default:
            exit(0);
    }
}
wait(NULL);
```

Figure 3.2 Asynchronous Process Creation

Each site process creates at least two asynchronous processes before creating its FIFO for receiving messages. The *forward* program, described more fully in the next section, is forked by the site process to pass messages on to the next site in the communication path field of the message structure when the message is not intended for this particular site. The forward process runs concurrently with the *site* process and does not terminate unless the site crashes.

The forward program is run as a separate process to free the site program from processing and forwarding messages, thereby increasing the response time of the simulation. As the number of processes and the number of sites in the network increases, so does the number of messages that have to be forwarded. This would greatly slow the system down if the forwarding program was not run as a separate process since the site process performs many functions for the processes that are run at the site and for the messages that are intended for the site.

The other asynchronous process that is forked is the *storflist* program. The purpose of this program is to send the names of the files, their last modification times, and their permissions to the CSS. The storflist process terminates when all file names have been sent to the CSS. The storflist process will be executed again when this site discovers that a new CSS has been elected (due to a site crash).

The demo program prompts for the name of the site to be used for the demonstration and also for a site to become the current synchronization site. If a given site has been chosen to be the demo site and/or the CSS, another asynchronous process (es) has to be forked and executed by the site. The user site program, *uspgm* (see section 3.1.6), executes the interactive utility that is used to access the system files. The current synchronization site program, *csspgm* (see section 3.1.8), executes the utility that synchronizes user site (US) system file accesses.

The final phase of the site program is the message receiving loop. The site creates a FIFO when the *receive* function is executed. All FIFO names are made up of the word "fifo" and a long integer **key**. **Keys** associated with a particular site contain the site name. For example, the site receiving **key** for each site is: site name + 100. The site forwarding **key** for each site is: site name + 200. Therefore, site 0 would have a a receiving **key** of 100 and a forwarding **key** of 200. A further explanation of the **keys** and message passing is found in section 3.1.2.

Operations to be performed by the site are contained in a switch statement. A single character found in the message structure field, **cmd**, determines the site command. When the site finishes executing the command, it returns to the top of the receiving loop and waits on the receiving end of the FIFO for more commands. The only exception to this is when the *die* command is received (~). The die command causes the site to kill any active processes that it created and then exit. This site command and the other options are described when discussed in subsequent sections.

3.1.2. Site Communication and Message Forwarding

Site processes communicate with one another via a FIFO. When either a reader or writer opens a FIFO, it blocks until the other end of the FIFO is opened. This allows the reader and the writer to synchronize themselves before the transmission of data begins. If blocking is not

desired, the FIFO can be set to not block when opened. This causes the process to return immediately if no other process has opened the other end of the FIFO.

A modified version of an implementation of message passing using FIFOs found in [ROCHKIND 85] is used. Rochkind creates a logical device which he calls a *queue*, to describe the place to which messages are sent. The queue is opened to deposit and retrieve data by a long integer called the **key**. Data to be added to the queue is added to the back of the queue while data to be read from the queue is taken from the front. Thus, the oldest data on the queue is read first.

Two primitives, *send* and *receive*, are used by Rochkind to describe message passing. If the queue fills up, the send primitive blocks until the queue has enough room for the message. The number of bytes read from or written to the queue is found in the nbytes field of the message primitives. Any left over bytes are left on the queue or discarded. Therefore, the sender and the receiver use the same size message on any given queue. The headers for the send and receive primitives are shown in figure 3.3.

```
BOOLEAN send (dstkey, buf, nbytes)  /* send message */
long dstkey;                        /* destination key */
char *buf;                          /* pointer to message data */
int nbytes;                         /* number of bytes in message */
/* returns TRUE on success or FALSE on error */

BOOLEAN receive (srckey, buf, nbytes)  /* receive message */
long srckey;                           /* source key */
char *buf;
int nbytes;
/* returns TRUE on success or FALSE on error */
```

Figure 3.3 Message Primitive Headers

The buf field of the message primitives is a pointer to a message data structure whose type may vary, so buf is declared as a character pointer. Two message types, DMESSAGE and SMESSAGE are used. DMESSAGE is used to send messages to the demo process, while SMESSAGE is used by the sites to communicate with each other and by the demo process to send messages to the sites. The two message types are shown in figure 3.4.

```
typedef struct {
    char cmd;                   /* demo command */
    int site;                   /* site name */
} DMESSAGE;


typedef struct {
    char cmd;                   /* site command */
    char data [MAXDATA];        /* message */
    long srckey;                /* key name of message source */
    long dstkey;                /* key name of message destination */
    int src;                    /* site name of message source */
    int dst;                    /* site name of message destination */
    KEYS fwd[MAXSITES];         /* network communication path */
} SMESSAGE;


typedef struct {
    long prev;                  /* key name of previous site in path */
    long next;                  /* key name of next site in path */
} KEYS;
```

Figure 3.4 Message Data Structures

The DMESSAGE message type is used to "suspend" file processing in order to test the effect a site crash, site reboot, or a new demo site has on the network. The site field of DMESSAGE contains the name of the site that is to be affected by the action of the suspend command. Section 3.2 describes these network testing features.

The SMESSAGE message type is used by the sites to send many different commands and data to one another. It may also be used by a program running at the site to retrieve network related information such as communication paths to other sites. Most of these commands are discussed in later sections of this paper. The SMESSAGE data structure itself and message forwarding is what is emphasized in this section.

When one site wants to send a message to another site in the network, a communication path must be constructed. The function *compath* is called to determine the shortest reachable path to the destination site. This function is recursive and it uses a depth first search algorithm to find the shortest reachable path.

The function compath, whose header is shown in figure 3.5, requires six inputs. The first, **network**, is a data structure that contains every site in the network and their links. The next input, **dead**, is a list of sites in the network that have "crashed" The **dead** list is checked before a new site is added to the path and if found to be "dead", it is not added to the path. The input, **start**, holds the sender site, while **dest**, holds the receiving site. The next input, **srchpath**, holds a candidate for the shortest reachable path, while **shortpath**, holds the current shortest path.

```
void compath (network, dead, start, dest, srchpath, shortpath)
SITE network[];           /* network site structure */
int dead[];               /* dead site list */
int start;                /* sender site */
int dest;                 /* receiver site */
char *srchpath;           /* pointer to candidate path */
char shortpath[];         /* current shortest path */
```

Figure 3.5 Communication Path Function Header

The search algorithm for finding the shortest reachable path is shown as follows:

```
add start to srchpath;
for each site from start to dest {
    call compath function for each link not in dead list and not visited;
    add new site to srchpath;
    if srchpath finished and (shortpath is empty or srchpath < shortpath)
        copy srchpath to shortpath;
    else if srchpath > shortpath
        exit from this compath call;
}
```

If a communication path can be determined, it must be converted and loaded into the **fwd** field of the SMESSAGE data structure. The **fwd** field is an array of structures of type KEYS The KEYS structure contains two long integer fields, **prev** and **next**, which contain the keys to the previous and the next site in the communication path. The **start** and **dest** sites are

loaded with 99, to indicate the beginning and the end of the communication path, in **fwd[0].prev** and **fwd[nsites].next.** The sites in between have keys in the 200 to 299 range to indicate that the message is to be sent to the forwarding process of each site in the communication path and not to the site (site receiving keys range from 100 to 199).

The forward process for each site waits on its queue for messages to pass along. When a message is received, the forward process finds its key (**fwd[n]**) in the **fwd** path list and sends it to the 'next' site in the **fwd** path. Depending on the value in the **cmd** field, 'next' may mean either **fwd[++n].next** or **fwd[--n].prev**. **Fwd[++n].next** is the next site for all commands except for the crash command ('!') and the response command ('@'). The response command message is sent back to the source site indicating the success or failure of the original message command. So the forward process sends the response message to the queue whose key is in **fwd[--n].prev**.

The crash command indicates that a site in the communication path has crashed. The forward process times out if it is blocked from sending to the **fwd[++n].next** queue after some maximum tries. When a site crashes, all queues associated with it are removed with the *rmqueue* function. Rmqueue removes the queue (FIFO) whose key is found in the input key variable.

The function *openfifo*, allows a process to have up to seven FIFOs open for reading or writing and keeps track of which file descriptor is associated with what FIFO (queue key). A clock counter is incremented every time openfifo is called and is stored with each FIFO's file descriptor and key whenever a certain FIFO is used. If an eighth FIFO is needed, the FIFO's file descriptor that was least-recently-used (the one with the lowest clock time) is closed and the new FIFO's key and file descriptor replaces the old one.

The number of FIFOs a process is allowed to have open at any one time is limited to seven because the total number of file descriptors that a UNIX™ process is allowed to have open at any one time is twenty. A site with many links, requiring a FIFO for each, could run out of file descriptors if several files are opened for processing. The number seven was arbitrarily picked and could be changed for efficiency reasons. Seven proved adequate for this simulation since testing was limited by the UNIX™ operating system as explained in section 4.3.

3.1.3 File Replication and Distribution

File replication in a network decreases the effect of site crashes on the system. As the number of file replicates increases, file accessing efficiency increases. Many well distributed file replicates decreases the distance between message source and destination. Communication path calculations are shorter and the chance that the communication path would have to be recalculated due to a site failure within the path decreases with a shorter path. Therefore, as far as network communication is concerned, file accessing would be most efficient if every site had a copy of every file in the system.

A file copy at every site would require tremendous amounts of memory at each site in a large network with many files. A lot of time would be required of the current synchronization site (CSS) to propagate copies of newly created files and to propagate file transactions. If file copies are distributed to nearly every site, rather than all sites, a large amount of memory would be needed on the CSS to store a site file list table to keep track of file locations.

Conversely, too few file copies causes inefficient file accessing. Longer communication paths result in a greater chance that the message will have to be rerouted or that the file is inaccessible. File inaccessibility will occur if all sites with the required file are down or if site failures cause the network to be partitioned into two or more smaller networks. Too few file replicates could cause a sub-network to not have any copies of some files.

No attempt is made in this simulation to determine the most efficient file replication scheme. For every file that is created, two replicates are created and distributed in the network for a total of three file *images*. Three file images are sufficient for this simulation since the number of sites in the network simulation is limited (see the subsection on problems encountered in the conclusions section).

Figure 3.6 shows a sample network and some files a, b, c, d, and e. Replicates are designated, for example, by a ' and a".

File distribution in this simulation is determined by the CSS when the newly created file is requested to be closed by the user site (US) through the storage site (SS) as described by the close protocol. The first message that the CSS receives is to close the file ('c' command). This command causes the CSS to decrement the **ofiles** field for this site in its process status table.

site	files
0	d
1	b,e
2	
3	a,b',d'
4	c
5	a,e'
6	
7	b",c'
8	a",c"
9	d",e"

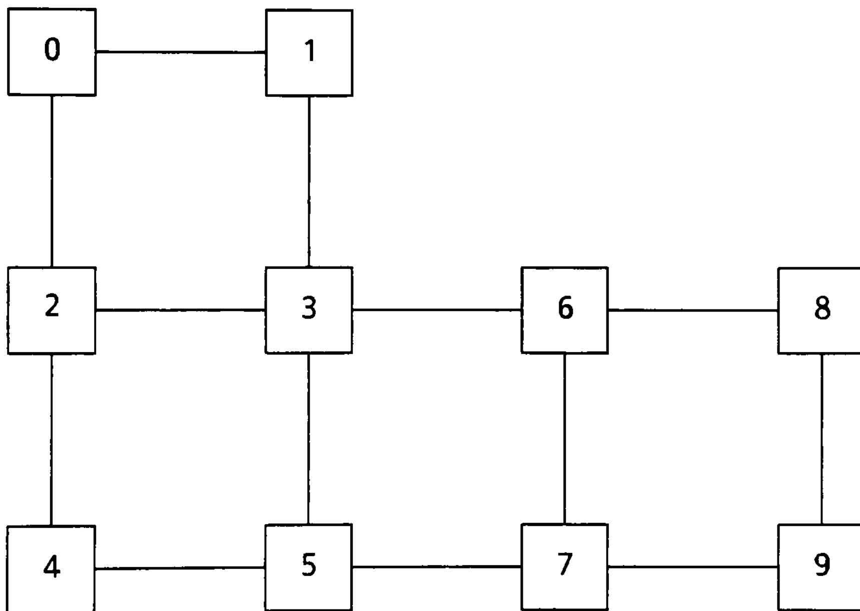


Figure 3.6 Simulated Network Example

The CSS keeps track of the number of files that a user process on a site has open because each user process in this simulated network is allowed to have only five files open at the same time. The CSS sends a response back to the SS, which causes the SS to transmit the newly created file to the CSS. When the entire file is received, the CSS picks the sites to hold the two file copies of the new file. Picking the sites to become storage sites involves finding the two sites in the **network** structure that are within two positions plus and minus from the US. The CSS propagates the new file copy to the replicate sites and stores the new file's name and the sites that store a copy of it in its file list table (see section 3.1.8. for a further explanation).

3.1.4. File and Record Locking

A locking mechanism is supported to provide protection of files during file accessing and to ensure the integrity of the distributed file system. Before any file can be accessed, the record or records needed must be locked with the appropriate lock at each file image location. A record level locking granularity is provided so that any number of records from one to n records can be locked. Two types of locks are provided to make file accessing serializable, a shared lock and an exclusive lock.

A file access (or transaction) is serializable if the effect of several concurrent file accesses has the same effect on the file as would the same file accesses done one at a time in any order. A shared lock on a record or records within a file allows multiple readers of the same record block and keeps all writers from accessing any of the records within the locked block of records. This means that any number of shared locks and no exclusive locks on all or part of a shared-lock record block can be acquired by other users. The owner of the shared lock is allowed to read the records but not update or add records.

An exclusive lock on a block of records does not allow other users to acquire either an exclusive lock or a shared lock on any record or records within this block. An exclusive lock allows the user to read the records in the record block and to update and add records. Thus exclusive access to an entire file can be achieved by an exclusive lock on every record in the file.

An unlock mechanism is also provided to remove shared and exclusive locks from records when the owner no longer needs to access the records.

To acquire a lock on a previously opened file, the US sends a lock request to the SS. The SS calls the *lockmgr* function, which determines if the lock is to be allowed at this site. If other users have a lock on this copy that would conflict with any part of the requested lock, the lock request is denied. If there are no conflicting locks, the SS sends the lock request to the CSS.

The CSS looks in its file list table for the sites that store the other two file images. The CSS then sends the lock request to these other storage sites. Any locks at either site that conflict with the lock request will cause the lock request to be denied. Otherwise, the CSS grants the lock and sends a message to the SS, which sends it on to the US.

Figure 3.7 shows a graphical depiction of the lock protocol described here.

The lock manager function, *lockmgr*, implements record locking with a table of **LOCK** structures found in each record of a file. The **LOCK** structure is made up of the process id (**pid**) of the owner of the lock and the type of lock (**lock**) on the record, represented as the single character- 's' (shared lock), 'u' (unlock), or 'x' (exclusive lock).

A table of **LOCK** structures is needed because multiple readers are allowed to share the same record for reading. Each process that wishes to share lock a record has its pid number added to the **LOCK** table provided there is no exclusive lock in existence on the desired record. When a process unlocks a record, its pid and lock are removed from the **LOCK** table.

An exclusive lock will be allowed on a record if there are no locks with pids that do not match the pid of the process requesting the lock. Only one exclusive lock is allowed per record and it is stored as the first entry into the **LOCK** table.

An existing lock on a record may be upgraded from a share lock to an exclusive lock if the pid of the process equals the pid stored in the **LOCK** table and this process's lock is the only share lock on the record. Any record within the requested record block with multiple readers will cause a lock upgrade to be denied.

The unlock lock request affects only those records whose **LOCK pid** matches the pid of the process requesting the lock. Therefore, an unlock of a whole file removes only the unlocking process's locks and does not affect the locks of other processes. It is not an error for a process to

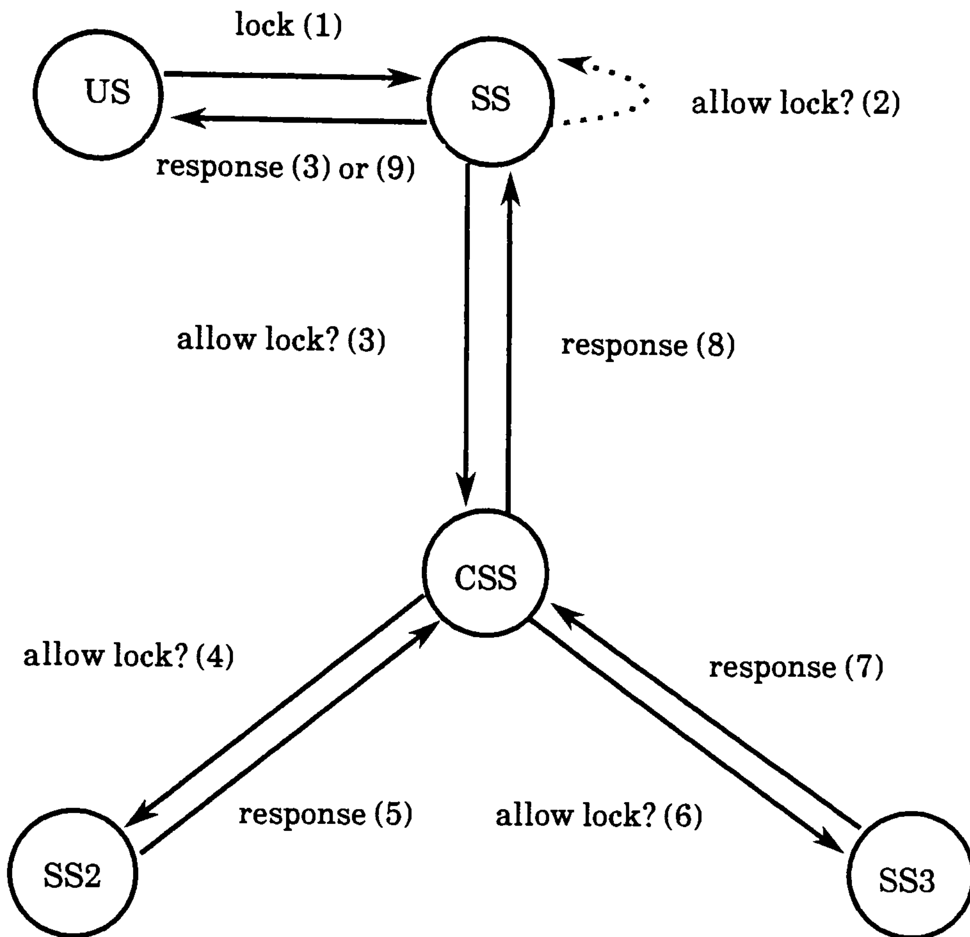


Figure 3.7 Lock Protocol

request that the whole file be unlocked even if the process does not have the whole file locked. This relieves the user of the chore of keeping track of which records in the file it has locked.

Since there are multiple file images in the distributed file system, the lockmgr sets the desired lock on the local copy if it is allowed while it awaits the outcome of the lock request from the CSS. This is done to prevent some other process at the local site from acquiring a local lock while this SS process is waiting for its response from the CSS. If the CSS denies the lock request, lockmgr clears the local locks and returns to the calling process, indicating that the lock was denied. If the SS receives a positive response from the CSS, lockmgr sends a message to the calling process indicating that the lock has been allowed.

The **LOCK** table in the record data structure is used for a quick test to see if the user has the appropriate lock for the database request. A request to read the next record in the file or to read a record by index requires that the process owns a share lock in the record's **LOCK** table. Deleting or changing an existing record, requires that the process owns an exclusive lock in the record's **LOCK** table. Adding a new record to a file requires that no other process has the entire file locked. If the process does not own an appropriate lock on the desired record, the database request is denied. Figure 3.8 shows the RECORD data structure.

```
typedef struct {
    char index[MAXINDEX];
    char rcd[MAXREC];
    char delete;
    LOCK netlocks[MAXLOCKS];
} RECORD;

typdef struct {
    int pid;
    char lock;
} LOCK
```

Figure 3.8 RECORD Data Structure

3.1.5. Transaction Mechanism

The transaction mechanism provided is atomic. Therefore, all changed records in a file within a transaction must be guaranteed to occur on every file image or all changes must be discarded. There can be no partial transaction updates since this would leave the file in an inconsistent state. The user must not be left guessing as to how many of the updates were actually committed to the file.

To start a transaction, a *begin transaction* command must be sent to the SS that has the file to be updated (a user can have up to five files open but only one file at a time may be designated the transaction file). The SS creates a temporary file to contain the changes to the file during the transaction. The changes will only become permanent if an *end transaction* command is received at a later date from the user and the commit is successful.

There are two ways in which a user's updates can be discarded. A user may cause the changes that were made during a transaction to be discarded by issuing an *abort transaction*

command. The abort command is sent to the SS, which upon receiving the command, discards the temporary file that held the transaction updates. The user is now free to send another begin transaction command to the same SS or go on and do something else.

A user's updates may also be discarded as the result of a site failure during the processing of an end transaction command. When a SS receives an end transaction command, the SS sends a copy of the transaction file with all the changes to the CSS. If the CSS does not receive the entire transaction file and respond to the SS due to a site failure, the SS will abort the transaction by discarding the updates and notifying the user of the aborted transaction. Otherwise, the updates are committed and the user is sent a commit message. A site failure may also cause the end transaction command to never reach the SS or the SS itself may crash before it can finish processing the end transaction command.

When the CSS receives the transaction file copy, it must propagate the changes to the other two file image sites. If one or both of the sites have not been designated as file image storage sites, then a site(s) has to be chosen. After the file has been sent to the other file image sites, the CSS waits for a response, containing that file image's new modification time, from these sites as an indication that the transaction file was received and stored.

A timeout by the CSS in trying to send to a file image storage site results in the modification time of that site 's file image to be set to zero in the CSS's file list table and the transaction file that the CSS received from the SS to be retained. No file image will be able to be accessed until it has been updated with the changes in the transaction file. When the site that had crashed comes back on-line, the CSS sends the transaction file to the rebooted site and waits for the file images's new modification time to be returned. If the file transfer is successful and all storage sites have a file modification time greater than zero, the transaction file is discarded. Otherwise, the transaction file is retained.

The transaction file retained by the CSS may be appended before the CSS is able to propagate the file to a storage site that has not yet come back on-line. This is allowed since the only storage site that would be allowed to be accessed would be the one that is up-to-date and has a modification time greater than zero. This site could be either the SS that sent the transaction file to the CSS or a storage site containing a successfully updated version of the same file. The

other file images of a file would be updated with the transaction file changes as soon as they become accessible to the CSS.

A problem occurs if the CSS crashes. The new CSS, which is chosen by finding the site in the **network** structure that is two sites away, will have no knowledge of the file system until the other sites in the network are made aware that a new CSS has been elected and have sent their file lists to the CSS. If the CSS does not receive a file list from every site in the network, the CSS will have an incomplete knowledge of the file modification times of some file image groups. The integrity of the file system can be adversely affected if an old copy is allowed to be updated before it has received the previous transaction updates. The new CSS would not have a copy of the transaction file in our example and cannot get one if the site at which the transaction occurred has since crashed. Also, the most up-to-date copy can not be determined if one or more file image sites are down when a new CSS is elected.

Since the integrity of the file system must be ensured, file access must be restricted if a new CSS is elected and the CSS does not have the modification times for each file image. Clearly, this is inefficient since access will be denied to some files that were never changed at any of the locations if one or more of the storage sites are down when a new CSS is elected. Normality does not return to the system until all sites of each file image group come back on-line.

All record locks that are needed in a file must be acquired before a transaction can begin. Otherwise, deadlock can occur if a lock is allowed to be acquired on the transaction file after the begin transaction command has been executed. All locks on a file acquired before a transaction remain after the execution of the end transaction command.

3.1.6. User Site Program

The user site program consists mainly of a user interface to the distributed file system and it is executed by the sites in the network. Each site in the network allows multiple users and each user site program that is executed returns its process id to its home site. The home site stores the process id of a new user site program in a process id table.

The **key** for the message queue of a user site program is determined by multiplying the home site number by one hundred, adding a process code of 20,000 to it, and adding the offset into

the process id table. This **key** is sent to the user site program, along with the home site number, when it is executed.

The user interface accepts one letter commands from the user and then prompts the user for additional information as required. A question mark is typed by the user to see a menu of the available commands (see figure 3.9). The user is prompted for more information when it is required by a user interface command. Upon completion of the execution of the command, the user will receive at least one of the following replies: OK, NOTFOUND, or ERROR.

a	abort transaction
b	begin transaction
c	close file
d	delete record
e	end transaction
l	lock file or record
m	make (create) file
n	read next record
o	open file
p	put record
q	quit
r	read record by index
s	suspend mode
t	rewind to top of file

Figure 3.9 User Interface Commands

The OK reply is sent to the user when a user interface command executes successfully. A NOTFOUND reply is usually returned if a requested record or file does not exist. A reply of FAILED usually means that the command failed because of a site crash or the user attempted to lock a transaction file, acquire a lock on a file where existing locks blocked it, or perform a file update or read without the proper locks. The NOTFOUND and FAILED replies usually are preceeded by a more specific error message concerning the reason for the failure of the command.

Many of these commands have already been discussed and so will not be dealt with here. The *quit* and *suspend mode* commands are only covered briefly here and are explained more fully in section 3.2. The commands that are left, record reading and updating commands and a file creation command, are presented here.

The *make* or *create file* command allows a user to open a new file locally. The user is prompted for the name of the file and access permissions. The US then sends a request to create a new file to the CSS. As long as the CSS can be reached, the user process at that US has not exceeded the five open file limit, and another file with the same name does not already exist, the user receives an OK reply. The locking mechanism is not used nor is it needed for newly created files since the user has exclusive access to a new file until it is closed.

The user interface, or *dfs* (distributed file system), prompts the user for the record definition when the make file command is invoked. The user is prompted for the names of the fields in a record and it must be defined with the first field as the index of the record. A maximum of eleven fields may be defined (including the **index** field) and the length of the **index** field is limited to ten characters, while the rest of the fields may contain up to fifteen characters each. The **index** field is used to directly access a record within a file for reading or updating.

The *read next record* and the *read record by index* commands are used to view the records in an opened file. Only those records within a file that have been locked with at least a share lock may be read. The read next record command reads the record starting at the current position and changes the current position pointer to the beginning of the next record in the file. If the end of the file is reached, the read next record command causes the file to automatically rewind to the top of the file and the first record in the file is read. The file can also be rewound manually by calling the *rewind to top of file* command.

The read record by index command allows a user to directly acccess a record if the user knows the index of the record. A NOTFOUND reply is returned to the user if the index is incorrect or if the record does not exist.

The *put record* command is used to add a new record to a file or to update an existing record. The user is prompted by the dfs for the index and the fields of the record to be added or updated. A new record is appended to the end of the file. An existing record must have an exclusive lock on it and put simply over-writes the old record. The file is checked for the existence of a record with the same index first, so multiple records with the same index cannot occur.

The *delete record* command also requires that the record to be deleted have an exclusive lock on it. The dfs prompts the user for the index of the record. If successful, the record is marked for deletion. Actual record deletion does not occur until the file is closed, but the deleted record is treated as not existing and cannot be accessed again. The dfs removes marked records by rewriting only the unmarked records to the file. If no record can be found with the given index, the NOTFOUND reply is returned to the user.

The *quit* command is used to exit from the demo user site program at a particular site. This command also allows another site to be specified as the demo user site.

The *suspend mode* command allows the user to suspend record processing to change the state of the network. As explained in section 3.2, any site in the network may be "crashed", "jumped to" or "rebooted" while in suspend mode. When this command has completed, the user is returned to the user interface.

3.1.7 Storage Site Program

The storage site program is executed by a site when that site is picked to be the SS for a US by the CSS. The SS is chosen based on the most up-to-date file image and the reachability of the storage sites. A site in a network is allowed to execute multiple storage site programs. The following information is included in the argument list sent to the storage site program when it is executed: the home site number, the name of the file requested by the US and the key for the storage site program's message queue. Each storage site program returns its process id to the home site.

The home site stores the storage site program's process id in a storage site pid table. The home site uses the offset into this table to calculate the storage site's message queue key in the same way as the message queue key is determined for each user site program, with the exception that a storage site code of 10,000 is used instead of a process code.

Once a US has successfully opened or created a file, the US no longer comunicates with the CSS regarding this file. All file requests are now sent from the US to the SS. Any file requests which require the services of the CSS, such as record locking or closing a file, have to be sent to the SS for processing before the file request is sent to the CSS.

The storage site program runs until the US sends a close file command to the SS. As long as the file to be closed is not a transaction file, the SS sends the close file request to the CSS (transaction files must first be ended with the end transaction command). Whether or not the SS is able to send the close command and any nontransaction changes to the CSS, the storage site program exits after returning an ERROR message to the US if the close could not be sent to the CSS or after returning an OK message to the US if the close was successfully sent to the CSS and an OK reply was received from the CSS.

It should be noted that nontransaction updates are not guaranteed and may leave a file group in an inconsistent state.

3.1.8 Current Synchronization Site Program

The function of the CSS is to synchronize file accesses on a network-wide basis, enforce a global locking scheme, and to propagate new files and existing file updates to the sites in the network. The file accessing commands that the CSS handles, as described earlier, are the open file, the make file, and the close file commands. File and record locking was described in section 3.1.4. and file propagation was discussed in section 3.1.3., so these topics are not covered in this section.

The initial CSS is chosen at the beginning of the demo program. Each site in the network is given, as an argument, the name of the initial CSS when the demo creates the site process. The site whose name matches the designated CSS, executes the current synchronization site program as an asynchronous process.

This simulation allows only one CSS in the network, at any given time, to provide both read and write access to the files in the network. However, because of network partitions and the need to provide read access to those sites in subnets without access to the CSS, multiple read-only CSSs can occur. A special flag is set to designate a CSS as read-only. This flag is checked each time a request is received to open or set locks on a file. As long as this read-only flag is set, files will only be allowed to be opened for reading and share locked.

A read-only CSS can become a "full service" CSS if the site is chosen later when the network has been restored. This is done by resetting the read-only flag. All files currently opened for

reading can be promoted to updating mode as long as exclusive locks can be acquired on the desired record(s).

The CSS initializes and maintains two tables; a site status table and a file location table (shown in figures 3.10 and 3.11 respectively). The current synchronization site program receives the names of the sites in the network as an argument and loads the site names into the site status table, CSTAT. Each site's queue is calculated (site + 100) and is also stored in CSTAT. Within CSTAT is a table to hold the queue key of the user processes at each site and an open files counter. A maximum of forty-eight user processes are allowed at each site in the network and each user process is allowed to have a maximum of five files open at any one time. The user process information is loaded into the CSTAT table when open file requests are received and updated when close file requests are received.

```
typedef struct {
    int site;
    long skey;
    PSTAT *psf[MAXPROC];
} CSTAT;

typedef struct {
    long pkey;
    int ofiles;
} PSTAT;
```

Figure 3.10 The Site Status Table, CSTAT

```
typedef struct {
    char file[MAXFILE];
    SITEMOD ssites[MAXFSS];
} SFLIST;

typdef struct {
    int site;
    long lastmod;
    char perm[MAXPERM];
} SITEMOD;
```

Figure 3.11 The File Location Table, SFLIST

The file location table, SFLIST, contains the name of every file in the network, the names of the sites that store the file images of a given file, the latest modification time of each file

image, and the permissions, read, write, and/or execute, of each file image. During the initial startup of the simulation, each site in the network, except the CSS, sends a list of the files, their last modification times, and their permissions to the CSS. The current synchronization site program interrogates its home site's file storage space and loads the file information into SFLIST. A maximum of fifty database files are allowed to be stored in the network. This means, at three images per file, the maximum number of file images allowed is one hundred and fifty. New files are added to SFLIST when they are successfully created and file modification times are updated when files are successfully updated.

The SFLIST table is dumped to a file, sysf, after each site's file list is received by the CSS and whenever a new file is added to the network and the SFLIST table. The sysf file is used to restore the SFLIST table after a site crash and a system reboot occurs at the CSS and no other site has been elected to be the new CSS. If this happens, the new current synchronization site program running at the rebooted site (the same site as the old CSS) loads its SFLIST table from the sysf file. A further explanation of the effect that a site crash at the CSS has on the network is explained in section 3.2.2.

3.2. Network Testing Facility

A facility to change the state of the network and to test the effect such changes have on the network, as well as the distributed operating system, is provided. The state of the network may be changed in three different ways; changing the demo user site, crashing a site, or rebooting a site.

3.2.1 Demo User Site Specification

The demo user site is specified at the beginning of the demo program execution, but it may be changed at any time during the demonstration of the simulation. This is done in one of three ways; executing the user interface quit command, "crashing" the site at which the demo user program is running, or "jumping" to another site in the network without exiting from the current user site program.

When executed, the quit command first attempts to close all files that the demo user site has open. If unable to reach any SS due to a network partition, that SS will remain idle, waiting

on its message receiving queue. Each "orphaned" SS remains in this state until either its home site crashes or the simulation demonstration terminates.

After sending close commands to each SS that is serving the demo US, the user program sends a quit command to the demo process and then the user program terminates its execution. The demo process prompts the user for his intentions regarding the continuation or termination of the simulation demonstration. If the user replies that a new demo US is desired, the user is prompted for the name of the new demo site. The demo process then sends a command to the site that has been chosen to be the new demo US to execute the user site program. Otherwise, all processes are terminated and the simulation demonstration ends.

The simulation also allows the user to specify a new demo user site if the home site of the US process crashes. Before a site crashes, it checks to see if it has executed a user site program. If the crashing site is also the demo US and if the crashing site has not received an "enddemo" flag, the crashing site sends a quit command to the demo process. As explained above, the simulation user has a choice of either continuing the demonstration by choosing another demo US or discontinuing the simulation demonstration. The site crashing testing mechanism is explained in greater detail in section 3.2.2.

The last method of changing the demo user site is by using the suspend command to jump to another site in the network. When the jump command is chosen at the suspend command line, the current demo user site waits on a special suspend message queue. This message queue uses a base value of 1100 and the site number is added to the base value to come up with its **key**. Thus the current demo US is effectively "suspended" until it is revived by sending a message to its suspend queue from the demo process.

After receiving the jump command, the suspend process sends the command in a message to the demo process and then exits. The demo process then inserts the current demo US into its **stopped** array table. Next, the name of the site to jump to is prompted for by the demo process. If the new site is listed in the **stopped** array table, the new site is restarted by sending a message to its suspend queue. If the new site cannot be found in the **stopped** array table, a "be demo" command is sent to the chosen site which causes it to execute a demo user site program.

3.2.2 Site Crashing Mechanism

The site crashing mechanism is accessed by executing the suspend program. The suspend process is created whenever the user issues the suspend command, 's', at the dfs interface command line.

It has already been seen that the suspend process accepts the jump command at its command line, but it also accepts the crash and the reboot commands. If the crash command is chosen, the crash command is sent to the demo process which then prompts the user for the site name to be crashed.

Upon receiving the site to be crashed, the demo process searches its site process id table, SPID, for the site. The SPID table holds the name, the process id, and the message queue key of each site in the network. If the site that is requested to be crashed is not found in the SPID table, an error message is printed and control returns to the user dos interface. Figure 3.12 shows the SPID table.

```
typedef struct {
    int pid;
    long key;
    char user[SSIZE];
} SPID;
```

Figure 3.12 The Site Process ID Table, SPID

If the site that is requested to be crashed is found in the SPID table, the demo process sends a *die* command message to the site and the demo removes the pid for that site from its SPID table. The demo process does not wait for a reply to its message, and waits on its message receiving queue for further commands from the sites in the network.

The site that receives the die command must terminate all processes that it has executed before the site process itself can terminate. The site process sends terminate signals to every storage site process, current synchronization site process, and user site process it has created and that is still running. Message queues used by the site and the processes that it created are also removed from the system before the site exits.

3.2.3. Site Rebooting Mechanism

As stated in the previous section, the site rebooting mechanism is accessed by executing the suspend program. The suspend program sends a reboot message to the demo process which looks for the site in its SPID table. If the site name is found, and as long as there is not a process already running with the same site name, a new site program is executed for that site. The new site process is sent a **reboot** flag as one of its arguments and the demo process stores the pid of the new site process in its SPID table.

A site process that is executed as a "rebooted" site, must find out where the current synchronization site is in the network. This is accomplished by sending *get css* command messages to each linked site in succession until one of them responds. If the CSS site name that is returned to the rebooted site is the same as the name of the rebooted site, then the rebooted site must create a new CSS process. If none of the linked sites responds to the get css command, then the rebooted site has been partitioned from the rest of the network.

The last possible result from petitioning the linked sites is that one of the links sends the name of the CSS to the rebooted site and the CSS site name is not the same as the rebooted site. If this is the case, the rebooted site sends a list of the files that it stores to the CSS.

4. Project Testing

This section describes the testing of the distributed file system simulation, the system limitations and problems encountered, and the solutions to these problems.

4.1. Test Plan and Procedures

The test plan consisted of two phases. During the first phase, the state of the network in a simulation did not change since no site crashes or site reboots are performed in this phase and the demo user site remains at the originally designated demo site. A small network of four or five sites was used in different network configurations. Larger networks were tested after the smaller networks tested successfully.

Several user site processes were to be run throughout the network to test the synchronization of file accesses and the locking mechanism. These other user site processes would simulate file accessing by opening files and setting different types of locks on records within the opened files. After sleeping for awhile, the "simulated" user processes would release their locks, close the files that they opened, and terminate.

Another method of testing the network during phase one involved changing the locations of the demo user site and the CSS for different simulation runs. The demo US and the CSS were made to run on separate sites that are either close to one another or far apart. The demo user site process and the CSS process were also run on the same site. If the demo US and the CSS process are run on the same site as the most up-to-date file image of a certain file group, then it would be possible to test the effect of having all three logical sites existing at one site in the network.

Testing of the dfs interface commands, except the suspend command, was done in phase one. Both nontransaction updates and transaction updates were tested. Initially, the make file command was given a heavy workout in order to create enough files for simulation testing. The CSS process also got a lot of use since the creation of files requires file images to be created and propagated to other sites in the network.

Phase two of the test plan involved testing the simulation while site crashing, site rebooting, and demo US respecification was going on. The site crashing mechanism provided by the suspend command was used to test the simulation under adverse situations. Crashing a site or sites between a source and an intended destination before sending a message, was used to test the ability of the simulation to find alternate paths to a desired destination and to examine the simulation's ability to determine that a destination site is unreachable due to a partitioning of the network. Likewise, a site crash at a destination such as a SS or a CSS, was used to test the system response to a destination site failure. A site failure at a CSS also tests the mechanism used to elect and set up a new CSS.

Site rebooting tests the ability of sites in the network to recover from a site failure. A successful recovery of most sites involves determining what site in the network is the CSS and sending a list of its stored files to the CSS. Other times the newly rebooted site must create a

CSS process if it had been the CSS before it crashed and no other site in the network was elected to be the new CSS while the old CSS was down.

Demo US respecification allows the continuation of the simulation even if the home site of the demo user site process has crashed. Demo US respecification also provides a tool to examine the effect that network partitioning has on the system. For example, if the network becomes partitioned into two or more subnets and the demo US still has access to its SS and/or the CSS, the demo US could be respecified at a site in a different subnet. Another subnet may not have access to the CSS; thus creating a new scenario in a particular simulation run.

The jump command allows multiple demo user processes to be run by a user, one at a time, while the original demo US and the other demo user sites are suspended. This gives the simulation the ability to run parallel background operations to test the integrity of the system.

After overcoming the problems and system limitations mentioned in the next section, the distributed file system simulation tested successfully and provided a useful tool for observing the effect of site failures on different network configurations of processing sites.

4.2. Problems Encountered and System Limitations

A major problem was discovered at the beginning of the testing phase. The first test of the simulation used a network of ten sites. Each site in the network needs a site process and a forward process at all times to simulate a site. A CSS process and a demo US process need to be created at some site(s) in the network. Simulated US processes may be run on some sites in the network to test the ability of the dfs to handle multiple file accesses and enforce the global locking policy. SS processes are created whenever files are successfully opened and SS processes continue running until the file is closed. Other processes such as the storflist process and the suspend process, come and go throughout the execution of the simulation.

A network of size ten, a demo process, a CSS, a demo US process, and ten storflist processes, requires a minimum of thirty-three processes. Add to this the number of processes needed for the simulated user sites and SS processes. This brings the total number of processes that would be running on and off to around forty.

Unfortunately the operating system upon which this simulation executes, does not allow a user to execute more than twenty-five processes at a time. The number of processes that a single user in a UNIX™ environment is allowed to execute at one time is hardware dependent and thus cannot be changed. This limitation necessitated changes in the testing procedures.

The first such change was a major reduction in the maximum network size that could be tested. Although the distributed file system simulation design allows as many as twenty sites in a network, a network of five sites was used to test the simulation. A five site network was used to allow those processes that are needed to simulate a network to execute, to allow a maximum of five open files for the demo US, and to allow other UNIX™ processes, such as the process status command, ps, to run on another terminal in order to monitor the execution of the simulation processes.

Shrinking the size of the network limits the number of different networks that can be configured. A network of twenty sites would allow several subnets of two and three sites in the event of network partitioning. A five site network, would allow, at most, only two two-site subnets with the failure of a site in the network.

The simulated US processes that were to be executed to test file accessing and locking were not used in order to stay within the system process limit. Instead, records within the network files were locked by using the jump command to create a new demo US and suspend the old demo US. This allowed the lock mechanism to be tested as to how well it maintained the integrity of the system. Only two or three demo user sites were run, however, to stay within the UNIX™ imposed process limit. Also, the multiple demos only opened two or three files each since every open file requires a SS process to access it.

Another problem encountered, was the problem with the UNIX™ unlink system call. According to The UNIX System [BOURNE 83], Advanced UNIX Programming [ROCHKIND 85], and The UNIX™ System User's Manual [AT&T 86], the unlink system call is executed to remove a file. If there are any processes with open file descriptors on the file to be unlinked, unlink waits until all file descriptors have been closed before removing the file. There is no indication in any of the aforementioned books that unlink handles FIFOs in a different manner. Yet it was discovered that when unlink is called to remove a FIFO, the FIFO is

removed even if there are open file descriptors on the FIFO. The UNIX™ system call, fstat, was used to check the validity of the file descriptor. Unfortunately, any file descriptors that were open on a FIFO when it was unlinked are still considered valid by fstat.

If a write to a nonexistent FIFO was attempted with an fstat-verified file descriptor, the Pyramid computer would often times crash. Sometimes, for reasons unknown, the Pyramid computer would react in a way that would seem more logical; that is, it wouild kill the offending process instead of allowing it to perform an operation that would cause the Pyramid system to crash.

Although it soon became apparent that this simulation program was causing the Pyramid system to crash, it was quite difficult to find the problem and correct it. The solution to this problem was to use the demo process as a simulation system monitor. Whenever a site in the network crashes, the demo process sends a "dead site" message to every receiving message queue and every forwarding queue of every site that is linked to the crashed site. Both the forward process and the site process for each link closes every file descriptor associated with the dead site. Each site process that is linked to the dead site also sends a dead site message to every US process that it has executed and is still running.

This change to the system eliminated Pyramid failures, but it did not affect any of the site's knowledge about the state of the network. The sites that are linked to a dead site do not remember anything about a dead site message that it receives after the processing of the message has completed. Therefore, the sites that are linked to the dead site continue to send messages to the dead site whenever it is the destination of the message or the message is merely passing through the dead site. A timeout during an attempt to send a message to a dead site signals the message source that the destination or a site in the communication path has crashed.

Timing problems between the processes in the simulation also were discovered during testing. Some of the timing problems resulted from faulty logic in the design of the simulation. Most of the timing problems, however, were caused by the nature of the simulation; multiple asynchronous processes running concurrently. For example, during the simulation start up, each site in the network creates the storflist process to send the list of files that it stores to the CSS. If the US process sends an open file request to the CSS before the CSS has received all of

the sites' file lists, the CSS may send a NOTFOUND reply back to the US. This also occurs during site reboots.

The solution to this timing problem was to make the affected process sleep a while. The amount of time needed to wait for other processes was determined by trial-and-error. Monitoring the simulation processes as they executed with the process status utility running on another terminal helped to determine how long the US process needed to wait before it could begin processing the dfs interface commands.

The last problem encountered was discovered while testing the simulation's file locking mechanism. Originally, the simulation's file locking mechanism incorporated the UNIX™ system call, fcntl (file control). The fcntl function provides advisory locking commands that were added following UNIX™ System V releases 1.0 and 2.0. According to The UNIX System User's Manual [AT&T 86], if a process closes a file, any locks that the process owned are removed from the file. Unfortunately, all locks are removed from a file when it is closed by a process, including the locks held by other processes that still have the file open. Because of this bug, the UNIX™ file locking mechanism had to be removed from the simulation and replaced by the original locking mechanism described earlier in section 3.1.4.

4.3. Project Test Results

Testing of the distributed file system simulation revealed two problems in its design. The first problem was found in the original design of the transaction mechanism's end transaction phase. After the CSS successfully received the entire transaction updates from the SS for a particular transaction file, the CSS would send an OK reply back to the SS and attempt to transmit the updates to the home sites of the other two file images. If one or both of these sites were down or could not be reached due to a network partition, the assumption was that this would not matter since the transaction file would be the most up-to-date and therefore would always be picked by the CSS to satisfy an open file request.

However, during the testing phase, two scenarios were found that would introduce inconsistency to the system using the design described above. If sometime after the CSS successfully receives the file updates and returns an OK reply to the transaction file SS, the home site of the transaction file SS crashes and stays down, the most up-to-date file will no

longer be available. Suppose that neither of the other two file images were updated because they were not on-line. Now suppose that one or the other of the unchanged file image sites reboots. This site will now contain the most up-to-date file image for this file group. Any file updates to this site's file image would be inconsistent with the file image that was updated earlier with the transaction mechanism. The transaction mechanism no longer can guarantee file updates.

In the second scenario, the same circumstances occur except that one or both of the sites that are suppose to receive the transaction file updates is partitioned rather than down. Suppose also that a US was in the same network partition as its SS, performing its own transaction updates. If the network becomes reconnected while the home site of the first SS is still off-line, inconsistent file images will again result.

The solution to this was to mark those site's file images that have not been updated with transaction updates in the sysf table of the CSS. The CSS was changed to not allow users to access any file images that were marked as not having been updated. File access would remain prohibited until the file with the transaction updates comes back on-line. Also, any file image that is marked would cause the CSS to abort any existing US transactions in a network partition when an attempt is made to commit the changes with the end transaction command. Although these changes decrease the efficiency of file accessing, the separate file images of each file group remain in a consistent state.

The second design problem pertains to the CSS drafting protocol. In the original design, if the home site of the CSS crashes, a new CSS is drafted when another site discovers that the CSS has crashed. This works fine as long as the home site of the old CSS does not come back on-line before some other site in the network is drafted to be the new CSS. If this does occur, the rebooted site creates another CSS, but it never receives the file lists of the other sites since the other sites in the network were never aware that the CSS had crashed.

The solution to this problem was to dump the sysf table to a sysf file after every new database file is entered into the table. This allows a site that is to remain the CSS after crashing and then rebooting to load its sysf table from the sysf file and to continue on as if a site failure had never occurred.

5. Conclusions

Although this simulation is limited by the number of processes that a single UNIX™ user is allowed to execute at any one time, it still has its uses as a learning tool. The simulation and ideas presented in this thesis can be used as guidelines for an actual distributed operating system implementation. Different scenarios involving the crashing and rebooting of the three logical sites, the US, CSS, and the SS, could be run using the simulation to see how they interact and adjust to changing network conditions. The locking and transaction mechanisms may also be helpful in the actual implementation of these mechanisms.

The use of FIFOs for inter-process communication may be useful in other applications that need such a mechanism. This would include any operating system applications that use multiple processes running concurrently.

The transaction and file mechanisms, could both be used as a basis for future work in both distributed and centralized file systems. For example, the transaction mechanism could be improved to allow, in a distributed system that provides file replication, different versions of the same file to be reconciled without introducing inconsistency into the system.

Appendix

A. User's Guide

A.1. Startup

A.2. Distributed File System User Interface

B. Test Plan

B.1. Network Site Crashing Scenarios

B.2. File Reading and Updating Test

B.2.1. File Accessing without Site Failures

B.2.2. File Accessing with Site Failures

B.3. Site Rebooting Test

B.4. File/Record Locking Test

C. Glossary

A. User's Guide

A.1. Startup

The user begins the simulation by typing *demo* at the UNIX™ command line. The demo program accepts one argument; the network configuration file. If no argument is given to the demo program or if the file given as an input does not exist, the user is prompted to construct a network configuration file. Two configuration files, con1 and con2, shown in figure 6.1 and 6.2 are provided.

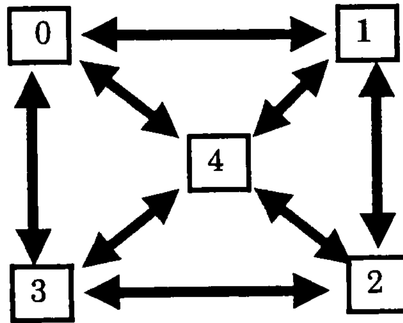


Figure 6.1 Con1

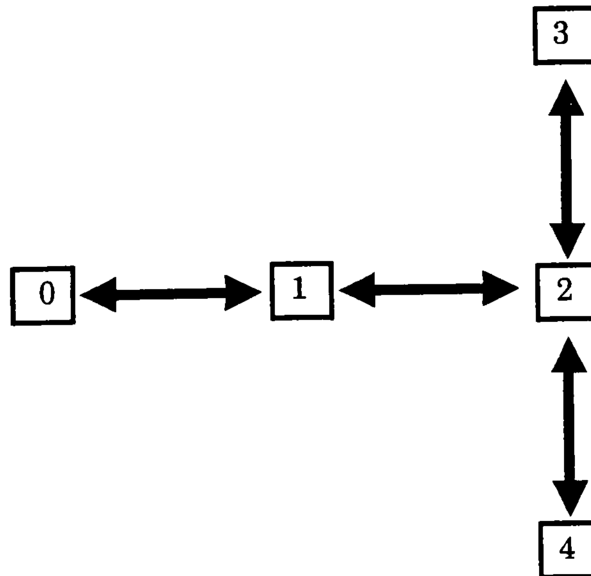


Figure 6.2 Con2

The construction of a network configuration file begins with the user naming the file. The user is next prompted for the number of sites in the network. The user then is instructed to enter the names of each site and the number and names of other sites that are linked to it. A predetermined number of maximum sites and maximum links is checked for violations of these limits when the user inputs the network configuration file information.

The name of the network configuration file is confirmed by the simulation by echoing it to the user when it has been determined to exist or when it has been constructed by the user. The user is now prompted for the initial demo user site and for the current synchronization site.

A.2. The Distributed File System User Interface

Once the simulation startup has completed, the user should see the following prompt on the screen:

Command (? for help) ->

The user types one letter commands at this command line. The help facility lists the commands and their meanings when *?* is typed (see figure 3.9).

A sample run follows with user input in italics, the initial demo user site at site 0, the current synchronization site at site 1, and con2 used as the description of the network. In the first section of the sample run the user creates a file with the 'm' (make) command and creates a three field record type. The first field of the record is used as the index to a record.

Command (? for help) -> *m*

File to create -> *tch_cour*

Enter number of fields -> *3*

Enter field name -> *Soc Sec*

Enter field name -> *Name*

Enter field name -> *Courses*

OK

The following section shows two records being added to the tch_cour file with the 'p' (put) command. Each field of a record to be added is prompted for by the system with the name of each field that is described in the record type.

```
Command (? for help) -> p
tch_cour is the current file
Enter file (<CR> for current) -> <CR>
Soc Sec -> 000000000
Name -> Samuel Quentin
Courses -> 007, 711, 747
OK
Command (? for help) ->p
tch_cour is the current file
Enter file (<CR> for current) -> <CR>
Soc Sec -> 222334444
Name -> Paul Bradley
Courses -> 776, 876, 976
OK
```

Next, the user opens tch_addr for writing with the 'o' command and share locks a block of five records starting at the record whose index is 222334444, with the 'l' command. The storage site chosen by the CSS to be the SS is site 4.

```
Command (? for help) -> o
Enter file -> tch_addr
Enter mode (r, w, x) -> w
OK
Command (? for help) -> l
Enter file -> tch_addr
Enter lock type (s, x, u) -> s
Enter starting index or 'all' -> 222334444
Enter number of records to lock -> 5
OK
```

The next section shows the two types of commands that can be used to read a record. The first one is the 'r' (read by index) command which prompts the user for the index of the desired record. The 'n' (read next) command is used to read the record pointed to by the file pointer.

```
Command (? for help) -> r
```

tch_addr is the current file

Enter file (<CR> for current) -> *<CR>*

Enter index -> *222334444*

Soc Sec:	222334444
Name:	Paul Bradley
Street:	25 Kernel Circle
City:	Avon
State:	NY

OK

Command (? for help) -> *n*

tch_addr is the current file

Enter file (<CR> for current) -> *<CR>*

Soc Sec:	111223333
Name:	Mary Morton
Street:	1 Boolean Blvd.
City:	Rochester
State:	NY

OK

The following section demonstrates the 's' (suspend) command. While the current demo user site is suspended, the user creates another user demo site and "jumps" to it with the 'j' command.

Command (? for help) -> *s*

S_Command (? for help) -> *?*

c	crash site
j	jump to new site
r	reboot site

S_Command (? for help) -> *j*

Enter jump site -> *3*

OK

At the new demo user site, the user opens the tch_addr file for writing and attempts to exclusive lock three records starting at the record whose index is 222334444. This lock is

denied by the system since it would violate the record locking protocol described in section 3.1.4 of this paper. The user next attempts a share lock on the same three records, which is allowed by the system as per the aforementioned record locking protocol.

Command (? for help) -> *o*

Enter file -> *tch_addr*

Enter mode (r, w, x) -> *w*

OK

Command (? for help) -> *l*

Enter file -> *tch_addr*

Enter lock type (s, x, u) -> *x*

Enter starting index or 'all' -> *222334444*

Enter number of records to lock -> *3*

Unable to lock tch_addr starting at 222334444

FAILED

Command (? for help) -> *l*

Enter file -> *tch_addr*

Enter lock type (s, x, u) -> *s*

Enter starting index or 'all' -> *222334444*

Enter number of records to lock -> *3*

OK

In the next section of the sample run, the user crashes site 2. This action causes the network to become partitioned and the tch_addr SS to become inaccessible. Therefore, the read by index command fails.

Command (? for help) -> *s*

S_Command (? for help) -> *c*

Enter site to crash -> *2*

OK

Command (? for help) -> *r*

tch_addr is the current file

Enter file (<CR> for current) -> *<CR>*

Enter index -> *222334444*

Network partition!

Unable to access record 222334444

FAILED

After the failure of the read by index command, the user reboots site 2. The user is now allowed to access the tch__addr SS at site 4. The user exclusive locks the record whose index is 888776666 and deletes this record from the file. A read by index for 888776666 is tried to confirm that this record has been deleted.

Command (? for help) -> *s*

S_Command (? for help) -> *r*

Enter site to reboot -> *2*

OK

Command (? for help) -> *l*

Enter file -> *tch_addr*

Enter lock type (s, x, u) -> *x*

Enter starting index or 'all' -> *888776666*

Enter number of records to lock -> *1*

OK

Command (? for help) -> *d*

Index -> *888776666*

tch_addr is the current file

Enter file (<CR> for current) -> *<CR>*

Command (? for help) -> *r*

tch_addr is the current file

Enter file (<CR> for current) -> *<CR>*

Enter index -> *888776666*

NOTFOUND

The user next reads a record by index, closes the tch__addr file, and quits demo user site 3. Demo user site 0 is reactivated and the files, tch__addr and tch__cour are closed. Lastly, the user ends the simulation run by not requesting a new demo user site.

Command (? for help) -> *r*

tch_addr is the current file

Enter file (<CR> for current) -> *<CR>*

Enter index -> *000000000*

Soc Sec: 000000000

Name: Samuel Quentin

Street: 53 Floppy Drive

City: Pittsford

State: NY

OK

Command (? for help) -> *c*

File to close -> *tch_addr*

OK

Command (? for help) -> *q*

New demo site? (y/n) -> *y*

Enter demo site -> *0*

OK

Command (? for help) -> *c*

File to close -> *tch_addr*

OK

Command (? for help) -> *c*

File to close -> *tch_cour*

OK

Command (? for help) -> *q*

New demo site? (y/n) -> *n*

The user in the sample run closed all files that it used without unlocking the records that the user had previously locked. This is allowed since during a close operation on a file, all locks by the calling process are removed automatically. The user, however, may manually unlock the records that it used in part or in whole. If the user cannot remember which records were locked, the user may unlock the whole file. In this case, the simulation checks each record in the file for a lock by the calling process and removes only those locks. Locks that are held by other processes are not changed and are simply skipped over.

Three transaction commands are available to the user: 'a' (abort transaction), 'b' (begin transaction), and 'e' (end transaction). Before invoking the 'b' command, the user must open and lock the file that is to be the transaction file. No locks will be allowed to be acquired after the 'b' command has been called and only the transaction file can be updated. Other files may however be read during a transaction. The 'a' command is used to discard transaction updates and to quit the transaction. The 'e' command is used to commit file updates and to exit from a transaction.

The following section shows the file tch_addr being opened and share locked for reading and tch_cour being opened and exclusive locked for writing.

```
Command (? for help) -> o
Enter file -> tch_addr
Enter mode (r, w, x) -> r
OK
Command (? for help) -> l
Enter file -> tch_addr
Enter lock type (s, x, u) -> s
Enter starting index or 'all' -> 222334444
Enter number of records to lock -> 5
OK
Command (? for help) -> o
Enter file -> tch_cour
Enter mode (r, w, x) -> w
OK
Command (? for help) -> l
Enter file -> tch_cour
Enter lock type (s, x, u) -> x
Enter starting index or 'all' -> 222334444
Enter number of records to lock -> 4
OK
```

The next section shows the user designating tch__cour to be a transaction file. Then a record from tch__addr is read, a record from tch__cour is changed, and the transaction is aborted.

Command (? for help) -> *b*

Enter transaction file -> *tch__cour*

OK

Command (? for help) -> *r*

tch__cour is the current file

Enter file (<CR> for current) -> *tch__addr*

Enter index -> *222334444*

Soc Sec:	222334444
Name:	Paul Bradley
Street:	25 Kernel Circle
City:	Avon
State:	NY

OK

Command (? for help) ->*p*

Soc Sec -> *222334444*

Name -> *Paul Bradley*

Courses -> *770 771*

OK

Command (? for help) ->*a*

OK

The file tch__cour is again made a transaction file by the user. The user next attempts to set a lock within a transaction and the lock is denied. Finally, the user makes a change to a record in the file and commits the change by ending the transaction.

Command (? for help) -> *b*

Enter transaction file -> *tch__cour*

OK

Command (? for help) -> *l*

Enter file -> *tch__cour*

Enter lock type (s, x, u) -> *x*

Enter starting index or 'all' -> *978756368*

Enter number of records to lock -> *1*

No locks allowed within a transaction!

FAILED

Command (? for help) -> *p*

Soc Sec -> *222334444*

Name -> *Paul Bradley*

Courses -> *770, 771, 772*

OK

Command (? for help) -> *e*

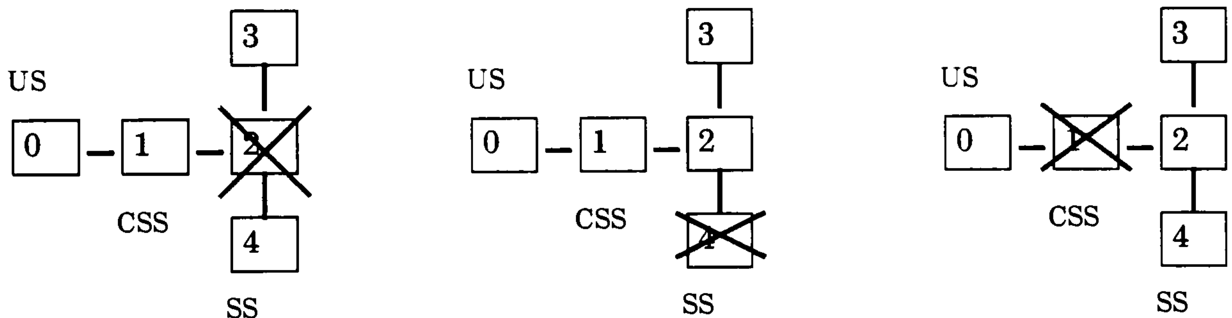
OK

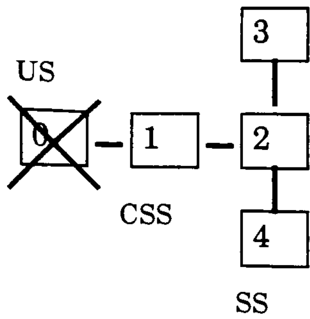
B. Test Plan

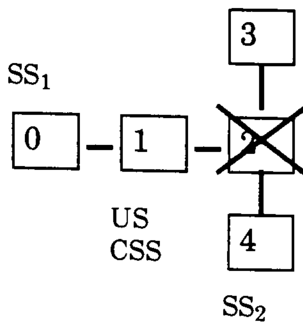
During each of the simulation runs in the test plan, a copy of the run was made. For each test run, a listing of the processes as they come and go was also obtained. The test runs that affected the records within a file(s) also contain a listing of the file(s) and their copies in the network.

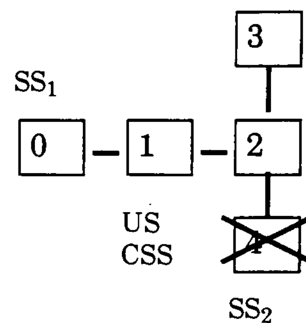
B.1. Network Site Crashing Scenarios

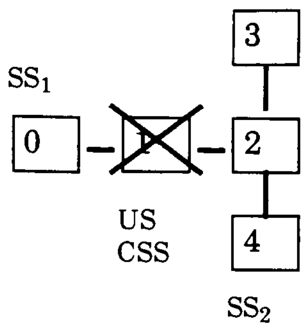
The following thirty-one network site crashing scenarios shows a five site network with various combinations of the three logical sites, US, SS, CSS, at different network sites. For each scenario, the simulation is run with the configuration shown except that all sites are on-line initially. During each simulation run, a file at the site that has been designated to be the SS(s) is locked and read to show that the file can be accessed. Next, the site(s) that is marked with an 'X' is crashed and the file at the SS is again accessed to show the effect of the site crash(es) on file accessing and site communication. During most of the simulation runs, the crashed site(s) is rebooted and oftentimes the rebooted site(s) is tested for its ability to continue site communication and file accessing.

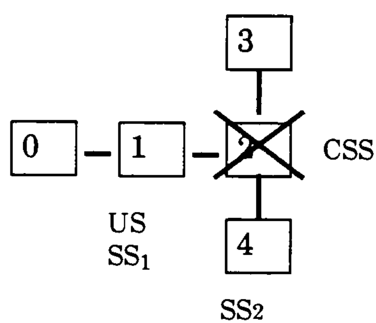


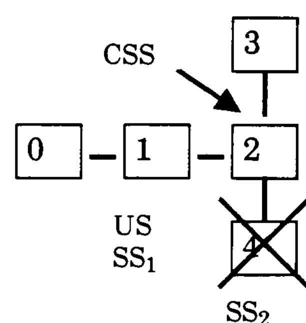

US
0
1
2
3
4
CSS
SS

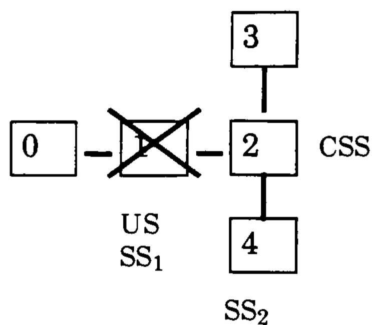

SS1
0
1
3
4
US
CSS
SS2

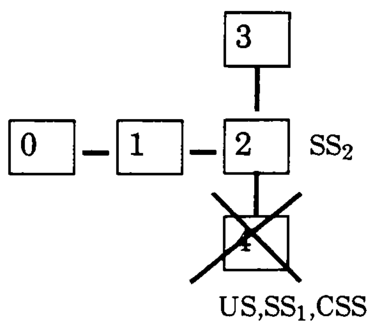

SS1
0
1
2
3
US
CSS
SS2

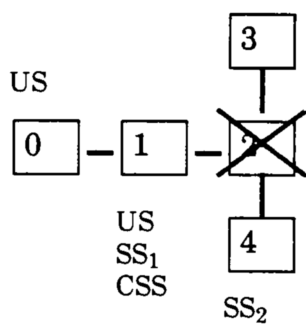

SS1
0
2
3
4
US
CSS
SS2

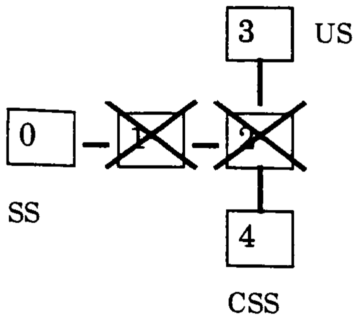

0
1
3
4
CSS
US
SS1
SS2

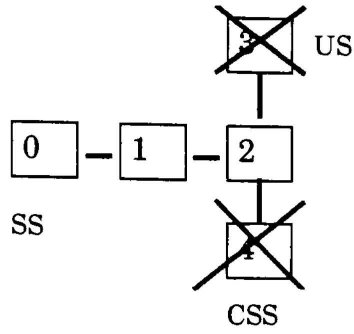

CSS
0
1
2
3
US
SS1
SS2

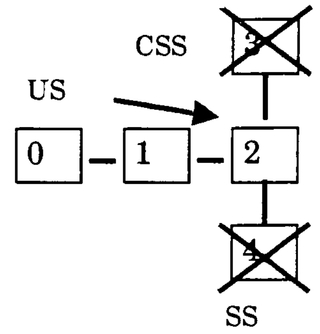

0
2
3
4
CSS
US
SS1
SS2

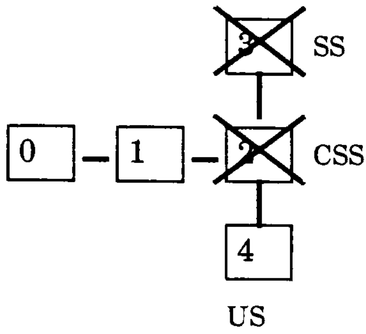

0
1
2
3
SS2
US,SS1,CSS

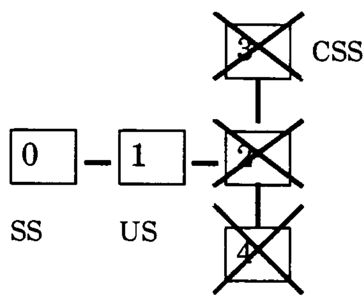

US
0
1
3
4
US
SS1
CSS
SS2

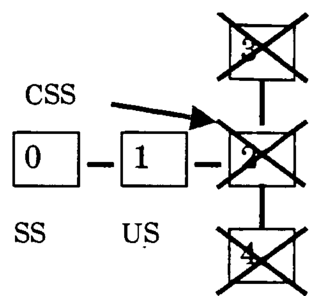

3
US
0
1
2
SS
4
CSS

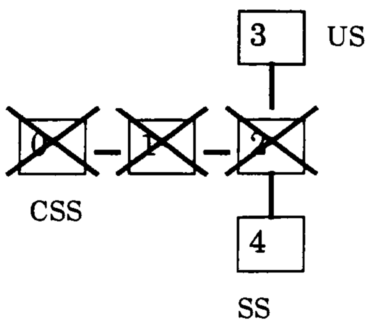

3
US
0
1
2
SS
4
CSS

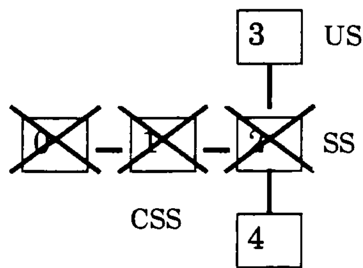

CSS
3
US
0
1
2
4
SS

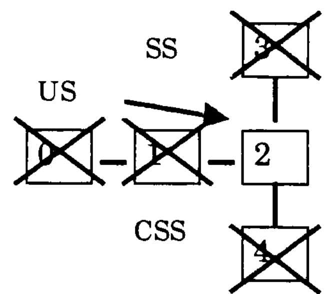

3
SS
0
1
2
CSS
4
US

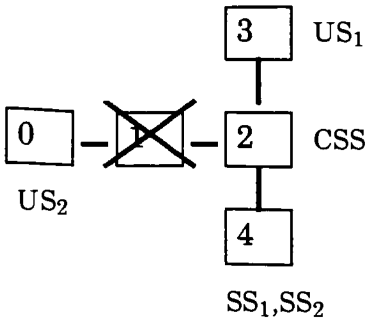

3
CSS
0
1
2
SS
US
4

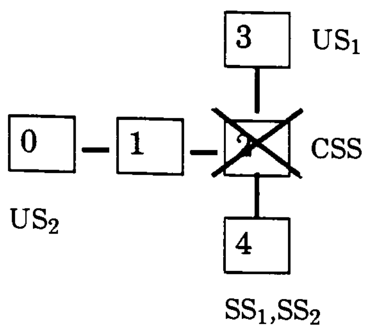

3
CSS
0
1
2
SS
US
4

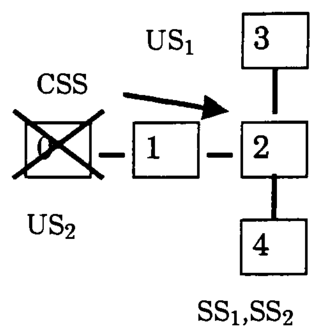

3
US
0
1
2
CSS
4
SS

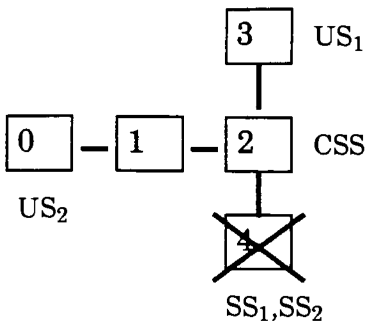

3
US
0
1
2
SS
CSS
4

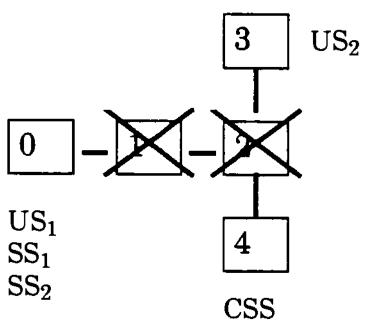

SS
3
US
0
1
2
CSS
4

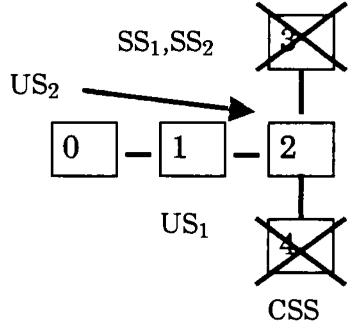

3
US1
0
1
2
CSS
US2
4
SS1,SS2

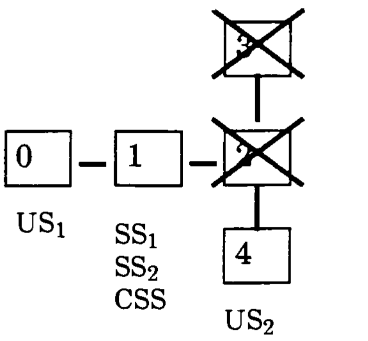

3
US1
0
1
2
CSS
US2
4
SS1,SS2

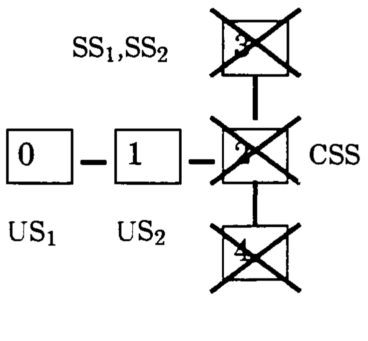

US1
3
CSS
0
1
2
US2
4
SS1,SS2

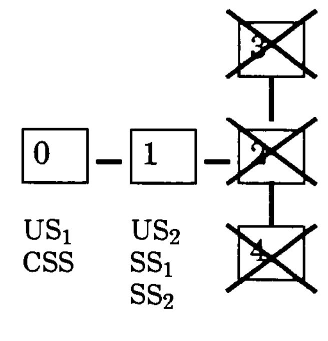

3
US1
0
1
2
CSS
US2
4
SS1,SS2

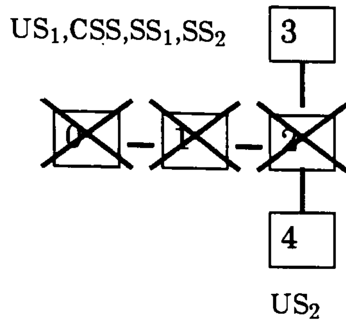

3
US2
0
1
2
US1
SS1
SS2
4
CSS


SS1,SS2
3
US2
0
1
2
US1
4
CSS


3
0
1
2
US1
SS1
SS2
CSS
4
US2


SS1,SS2
3
0
1
2
CSS
US1
US2
4


3
0
1
2
US1
CSS
US2
SS1
SS2
4



B.2. File Reading and Updating Test

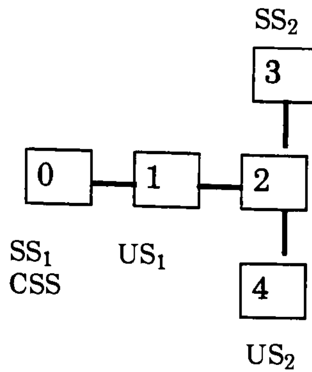
B.2.1. File Accessing w/o Site Failures

The test performed with the following network configuration shows the ability of users to perform transaction and non-transaction file reads and updates with no site crashes. The simulation run with this configuration began with the site designated as US_1 as the initial demo user site. While at US_1, a new file was created and records were added to, deleted from, changed, and read from the new file. US_1 also opened files served by SS_1 and SS_2, set locks on them, and designated SS_1 to be a transaction file and SS_2 to be a non-transaction file. The demo user site was next changed to US_2.

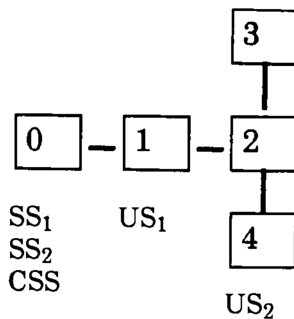
While at US_2, the same files were opened and locked, but SS_2 was designated to be a transaction file and SS_1 to be a non-transaction file. File operations were performed on both files and the changes made to the transaction file were aborted before the files were closed. The files were closed on US_2 after the demo user site was changed again to US_1. The transaction and non-transaction files at US_1 were also updated and read from, but the transaction was committed rather than aborted.

B.2.2. File Accessing w/ Site Failures

A simulation run showing the effect of site crashes on transaction and non-transaction file updates was performed on the following network configuration. As in the previous section, a new file was created at US_1 and both US_1 and US_2 opened and locked two different files and

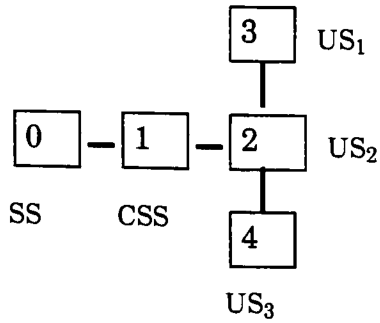


designated the opposite files to be the transaction and non-transaction files. Site crashes occurred at sites 1 and 2.



B.3. Site Rebooting Test

A test to show the effect of site rebooting on file accessing and file locks was performed on the following network configuration. Each user site, US_1, US_2, and US_3, opened and set locks on the same file served by SS. After each site demonstrated its ability to access the file, site zero was crashed. Next, site zero was rebooted and the same file was opened and locked by each of the user sites. Sites two and three were then crashed and rebooted and US_3 quit its operations. Finally, a new US_1 at site three again opened, locked, and accessed the same file served by SS_1.



B.4. File/Record Locking Test

The following table depicts the file/record locking test. Each column in the table is a record within a file and each row of the table is a user site lock request from either US_1, US_2, or US_3. A lock request is either a share lock (S), an unlock (U), or an exclusive lock (X). Lock requests are chronological from the first row to the last. The subscript 'all' means that the lock request was allowed, while the subscript 'den' means that the lock was denied.

	Rec1	Rec2	Rec3	Rec4	Rec5	Rec6	Rec7	Rec8	Rec9	Rec10	Rec11	Rec12
US_1	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}	S_{all}
US_1	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}	X_{all}
US_2				S_{den}	S_{den}	S_{den}						
US_2	X_{den}	X_{den}	X_{den}									
US_1			U_{all}	U_{all}	U_{all}	U_{all}	U_{all}					
US_2		S_{den}	S_{den}	S_{den}								
US_2				S_{all}	S_{all}							
US_3				X_{den}	X_{den}	X_{den}	X_{den}	X_{den}	X_{den}			
US_3						X_{all}	X_{all}					
US_1				S_{all}	S_{all}							
US_3				S_{all}	S_{all}							
US_1	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}	U_{all}
US_2	X_{all}	X_{all}	X_{all}									
US_3				U_{all}	U_{all}				X_{all}	X_{all}		
US_3								S_{all}				
US_2				X_{all}	X_{all}	S_{den}	S_{den}	S_{den}				

User Site File/Record Locking Request Table

C. Simulation vs. LOCUS

Simulation

-> partially connected, partially connected network

-> no global naming mechanism; every new file name is checked for its existence in the current file system

-> only the site that is designated to be the CSS contains system file information

-> one CSS is designated for the entire network to control access to all file groups; multiple CSSs can only occur when the network is partitioned into multiple networks

-> only reading of files will be allowed in subnets that do not contain a "full service" CSS; a read only CSS is designated in each of these deficient subnets

-> allows record-level granularity in locking files

LOCUS

-> token ring network

-> global naming mechanism; provides a mapping facility between user defined global names and system internal global names.

-> concept of logical file groups (replicates of a file) with information on each file group stored at each site in the network

-> one CSS for any given file group; a CSS may synchronize access for more than one file group and multiple CSSs are allowed

-> file updating allowed during network partitions; file images are equated when access to all files in the file group has been restored

-> allows byte-level granularity in locking files

Simulation

-> the transaction mechanism does not allow nesting and is limited in its scope

-> only distributed file system features are implemented

LOCUS

-> provides a full implementation of a simple-nested transaction mechanism

-> provides other mechanisms such as remote creation and execution to constitute a distributed operating system

D. Glossary

current synchronization site (CSS) : a site in a network that synchronizes file accesses and chooses the storage site for a user site. The CSS also enforces a global locking policy for the files in the network.

distributed file system (dfs) : mechanism used to coordinate file operations in a network of independent processors.

file image : one of several copies of a file.

file replication : process by which one or more copies of a file are stored at various locations in a network.

network : interconnection of processing sites; allowing communication between sites.

network partition : creation of two or more subnetworks due to site or communication link failures.

network transparency : under normal circumstances, no knowledge of the network is needed to interface with foreign sites.

replication transparency : file replication and storage location is hidden from the user under normal circumstances.

storage site (SS) : site in the network that provides a file copy to the user site for accessing.

update propagation : process by which the CSS transmits file modifications to the other file images in the network.

user site (US) : site in a network requesting a file.

References

[AT&T 86]

AT&T Information Systems, The UNIX™ System User's Manual, AT&T, 1986.

[BOURNE 83]

J.R. Bourne, The Unix System, Bell Telephone Laboratories, Inc., 1983.

[BRERETON 82]

Pearl Brereton, *Detection and Resolution of Inconsistencies among Distributed Replicates of Files*, Operating Systems Review, Vol. 16, No. 4, October 1982.

[BROWNBRIDGE 82]

D. R. Brownbridge, L. F. Marshall, and B. Randall, *The Newcastle Connection or UNIXes of the World Unite!*, Software - Practice and Experience, 1982.

[DANIELS 86]

Dean Daniels and Alfred Z. Spector, *An Algorithm for Replicated Directories*, Operating System Review, Vol. 20, No. 1, January 1986.

[DATE 83]

C. J. Date, An Introduction to Database Systems, Vol. II, IBM Corporation, 1983, pp. 291 - 332.

[DENNING 83]

Peter J. Denning and Robert L. Brown, *Should Distributed Systems Be Hidden?*, International Workshop on Computer Systems Organization, Sheraton New Orleans Hotel, New Orleans, March 1983.

[HUNTER 85]

E. Hunter, *Adding Remote File Access to Berkeley Unix 4.2BSD Through Remote Mount*, Report No. UCB/CSD 85/224, PROGRES Report, No. 85.3, Feb. 1985.

[KERNIGHAN 78]

Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, Bell Telephone Laboratories, Murray Hill, New Jersey, 1978.

[KLEINROCK 85]

Leonard Kleinrock, *Distributed Systems*, Communications of the ACM, Vol. 28, No. 11, November 1985.

[LUDERER 81]

G. W. R. Luderer, H. Che, J. P. Haggerty, P. A. Kirlis, and W. T. Marshall, *A Distributed UNIX System Based on a Virtual Circuit Switch*, Proceedings of the Eighth Symposium on Operating Systems Principles, Asilomar, California, December 14-16, 1981.

[MUELLER 83]

Erik T. Mueller, Johanna D. Moore, and Gerald J. Popek, *A Nested Transaction Mechanism for LOCUS*, Proceedings of the Ninth Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, October 1983.

[NESSETT 82]

D. M. Nessett, *Identifier Protection in a Distributed Operating System*, Operating Systems, Review, Vol. 16, No. 1, January 1982.

[PETERSSON 85]

James L. Petersson and Abraham Silberschatz, Operating Systems Concepts, Second Edition, University of Texas @ Austin, 1985, pp. 457-503.

[POPEK 81]

G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel, *LOCUS: A Network Transparent*, High Reliability Distributed System, Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December, 1981.

[ROCHKIND 85]

Mark J. Rochkind, Advanced UNIX Programming, Advanced Programming Institute, Ltd., Boulder, Colorodo, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[ROWE 82]

L. A. Rowe and K. P Birman, *A Local Network Based on the UNIX Operating System*, IEEE Transactions on Software Engineering, Vol. 8, No. 9, March, 1982.

[SOBELL 85]

Mark G. Sobell, A Practical Guide to UNIX System V, Benjamin/Cummings Publishing Co., Inc., Menlo Park, California, 1985.

[SPECTOR 83]

Alfred Z. Spector and Peter M. Schwarz, *Transactions: A Construct for Reliable Distributed Computing*, Operating Systems Review, Vol. 17, No. 2, April, 1983.

[TANENBAUM 81]

Andrew S. Tanenbaum, Computer Networks, Vrije Universiteit, Amsterdam, The Netherlands, 1981, pp. 440-483.

[WALKER 83]

Bruce Walker, Gerald Popek, Robert English, Charles Kline, Greg Thiel, Proceedings of the Ninth Symposium on Operating Systems Principles, Bretton Wood, New Hampshire, October 10-13, 1983.

[Weber 84]

Weber Systems, Inc. Staff, C Language User's Handbook, Weber Systems Inc., 1984.

[WEINSTEIN 86]

Matthew J. Weinstein, Thomas W. Page Jr., Brian K. Livezey, and Gerald W. Popek, *Transactions and Synchronization in a Distributed Operating System*, Operating Systems Review, Vol. 20, No. 1, January 1986.