1989

The DFS distributed file system: Design and implementation

Ananth K. Rao

Rochester Institute of Technology
School of Computer Science and Technology

The DFS Distributed File System: Design and Implementation

by

Ananth K. Rao

A thesis, submitted to
the Faculty of the School of Computer Science and Technology,
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 Anderson

February 21, 1989

Permission to Reproduce this Thesis

The DFS Distributed File System: Design and Implementation

I Ananth K. Rao, hereby grant permission to the Wallace Memorial Library of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: Feb 21st 1989

Acknowledgements

This thesis is dedicated to my parents for having given me the opportunity to study in this country.

I would like to thank my thesis advisor Dr. Heliotis for his help during the course of this thesis.

I would also like to thank P. Rajaram for his guidance during the initial phases of this thesis. Thanks are also due to Catherine Henderson whose generosity during this difficult period, I will never forget.

My special appreciation goes to my wife Rajini, without whose constant encouragement, I would have given up a long time ago.

The DFS Distributed File System: Design and Implementation

Ananth K. Rao

Abstract

This thesis describes the design of an operating system independent distributed file system (DFS) and details the implementation, on a cooperating set of server computers interconnected by means of a communications network. The DFS provides the mechanism by which the file systems of these physically dispersed units are integrated into a single logical unit. Users and application programs thus have the illusion that their files are on a single computer system, even though in reality they may be physically distributed around the network. This location transparency frees users from having to remember details such as the current location of the file and also affords considerable mobility, allowing access to files from any workstation. In addition, automatic storage replication and an atomic transaction mechanism provides high reliability and improved availability in the distributed environment in the face of site failure.

Computing Review Subject Codes

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

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

TABLE OF CONTENTS

1.0 Introduction

Distributed computing today is made possible by the price - performance revolution in hardware technology on the one hand and the development of effective interconnection structures on the other. Some of the logical criteria [LELA 85] which defines a distributed computing environment could be :

* multiplicity of general purpose resource components
* interconnected computing processing elements having apparent or hidden levels of control.
* system transparency
* system wide executive control performed by several processes without hierarchical relationship and which do not have a coherent view of the entire system.
* processes have disjoint address spaces and communicate via explicit message passing.

The desire to share these physically distributed hardware and software resources among users is evident in the emergence of low cost local area network implementations (eg. Ethernet) and standard communications protocols (eg. TCP/IP). One of the major components of such a distributed computing environment is a distributed file system which constitutes the underlying mechanism for sharing information. The file system plays a central role in distributed system architecture both because file system activity typically dominates in most operating systems and so high performance is critical and because generalized name services provided at the file system level are used by so many other parts of the system.

This thesis describes the implementation of an independent distributed file system. It is so described because it is not embedded within an operating system and because it is implemented on a cooperating set of server computers connected by a communications network, thus creating the illusion of a single logical system for the creation, deletion and accessing of data.

For the purposes of this thesis, a distributed system is considered to consist of a multiplicity of physically dispersed homogeneous AT&T 3B1 workstations interconnected by means of a local area network (Ethernet) and running the UNIX (System V) operating system. The distributed file system (DFS) provides the mechanism by which the file systems of physically dispersed units are integrated

into a single logical unit. Users and application programs thus have the illusion that their files are on a single computer system even though in reality they may be physically distributed around the network. The global name space under which the file system operates allows users to access files in a transparent fashion, freeing the user from having to remember details such as the current location of the file or the site where it was created. It also allows considerable user mobility, allowing the user to access any file from any workstation. In addition, an automatic storage replication mechanism provides high reliability and improved availability in the distributed environment in the face of site crashes.

Section two reviews some existing distributed file systems. Section three lists the design goals and assumptions and section four describes the architecture while section five details the implementation. Section six describes project test results while section seven lists the current state of the implementation and describes possible future enhancements and finally section eight contains concluding remarks.

2.0 Review of existing distributed file systems

This section briefly reviews some existing distributed file systems, some of the concepts of which have been incorporated in the design of the DFS.

Distributed file systems can be divided into three categories based on the model used to provide service [FORD 85]. These are: super root systems, remote mount systems and global file name space systems.

2.1 Super root systems:

Systems such as the Newcastle Connection [8ROW 82] and COCANET [LUDE 81] fall into the category of super root systems. In these systems, a special file is used to extend the root directory of existing Unix file systems, allowing potential access to the entire remote file system. Naming transparency does not exist in these systems as users need to know on what system their files exist and need to specify a complete path name with respect to the super root directory. These systems do however support file access transparency.

The Newcastle Connection consists of an additional layer of software in each of the component Unix systems, sitting between the resident kernel and the rest of the operating system. From above, the layer is functionally indistinguishable from the kernel and from below it appears to be a normal application level user process. The system uses run-time mapping tables to map file names to their locations and uses a remote procedure call mechanism (RPC) to access remote files. This software layer thus performs the role of 'glueing' together the component file systems to form what appears to be a single structure.

2.2 Remote mount systems:

The Sun Network file system (NFS) is an example of a remote mount system. It is an operating system independent service which allows users to mount directories (including root directories) across the network and then use those directories as if they were local [WALS 85]. The system refers to a replicated read only data base called the Yellowpages [SAND 85] to access network data

independent of the relative locations of the client and server. NFS is built on top of an RPC mechanism to help simplify protocol definition, implementation and maintenance. In addition it makes use of an External Data Representation (XDR) specification to describe protocols in a machine and system independent fashion by converting between local machine representation to a machine independent representation.

The NFS uses two file system abstractions; the *virtual file system interface (VFS)* and the *vnode interface* to implement the file system. Since two inodes (in a Unix context) on two machines in a network environment having the same number could cause name conflicts, the NFS uses an abstraction of inodes called *vnodes* which guarantees that each vnode number is unique. The VFS interface is built on top of the vnode mechanism and separates file system operations from the semantics of their implementation.

The NFS uses a 'stateless' protocol in its client server model thus making crash recovery very easy. When a server crashes, the client re-sends the request until a response is received. The server maintains no state information and does no crash recovery. This simplifies the protocols immensely since when state is maintained on the server, the server would have to detect client crashes so it can discard state it is holding on behalf of the client while the client must detect server crashes so that it can rebuild the servers state.

2.3 Global name space systems:

Systems that fall into the global file name space category are Locus and the Apollo Domain file system.

2.3.1 The Locus Distributed Operating System:

The Locus distributed operating system, developed at the University of California at Los Angeles is a distributed version of the Unix operating system with extensive changes to the kernel to aid in distributed operation and support for high reliability and network transparency. To maximize performance, all Locus software is situated within the operating system nucleus thus avoiding much of the overhead associated with "abstract networks" of layered software [POPE 81].

The file system is the heart of the operating system. It presents a single tree structured naming hierarchy to users and applications. This path name tree is made up of collections of file groups (analogous to the Unix file system). Connections among these wholly self contained subtrees compose a 'supertree' where nodes are file groups. Thus there is only one logical root for the tree in the entire network. The file system provides a very high degree of network transparency while at the same time supporting high performance and automatic replication of storage. Locus achieves location transparency through a global file name space. Two types of global names exist; user defined global names and system internal global names which are used for internal references to a file. In addition a number of high reliability functions such as nested transactions and record level locking are also provided [WALK 83a].

Locus follows the *integrated model* of distributed systems wherein each machine in the network runs the same software. Individual machines in a Locus network cooperate to create the illusion of a single machine. This greatly simplifies the system since only one implementation is required [WEIN 86]. Each site in the Locus network is a complete system and is capable of functioning alone or as part of the network [POPE 85]. Locus applications never need to reference a specific site or node in the network. In this way, Locus achieves a high degree of network transparency.

2.3.2 The Apollo Domain Distributed File System

The Apollo Domain system is a distributed processing system designed for general purpose and interactive graphics applications. It is a collection of personal workstations and server computers interconnected by a high speed token ring network. The Domain architecture creates an "integrated distributed environment": it is distributed because users have their own workstations and it is integrated because the system provides mechanisms, and the user can select policies, that permit a high degree of cooperation and sharing [NEL5 84].

The Domain distributed file system fosters cooperation and sharing by allowing users and programs to name and access all system entities in the same way, regardless of their location in the network. Consequently all users and application programs view the system as an integrated whole rather than as a collection of individual nodes. All files in the Domain system are identified by their

globally unique identifiers (UID's). Mapping between human readable string names for these objects to their UID's is maintained by a name server.

3.0. System design goals and assumptions

The principal motivation for the development of a distributed file system is the need to provide *information sharing*. This motivates the following system goal which will be defined, along with a brief explanation on how the DFS will realize them.

* Location transparency
* Consistent naming
* Data consistency
* Reliability

Location transparency allows a user or administrator to move file system objects from one network node to another without having to find or alter all name and program references to that object. Therefore the human readable name for an object must not bind the specified object to a particular node. Location transparency results in being able to open a file in precisely the same manner independently of whether the file is local or remote i.e. issue the same system call, with the same parameters in the same order etc. If *open* 'file name' is used to access local files, it is also used to access remote files. The network thus becomes invisible . Location transparency is achieved in the DFS by having the user view the entire distributed file system as one logical hierarchically structured tree. Users and application programs are thus unaware of the underlying network and that their files are physically distributed over several sites.

Consistent naming means that every program and user in the network uses exactly the same name text for a file system object, regardless of where it exists. The system must thus guarantee that a given name will access a particular file from anywhere in the network. This is achieved by site independent naming of files. Every file has a unique global logical name and the system maintains a mapping between these names and physical file names.

To increase *availability* and *reliability* of files, provision is made in the DFS to replicate files at several sites. In this way, if a site fails it is likely that other sites will continue to operate and provide access to the file. Availability and reliability of a file can be made arbitrarily high by increasing the

order of replication. Since data is replicated at one or more sites, the system ensures *data consistency* by keeping these file images consistent. Consistency is maintained by means of two version numbers, an original version number and a current version number associated with each file [BRER 82]. The original version number keeps track of partitioned updates while the current version number is incremented every time the file is updated. The original version number is unused in the current implementation as partition of the network into independently operating subnetworks is assumed not to occur. Thus the problems related to partitioned update are not considered. This version number can be used to accommodate partitioned update if it is to be implemented at a later date and section 5.5 on file consistency describes a possible protocol that may be used.

The DFS protocols assume that the underlying network is fully connected. If site A can communicate with site B and site B with site C, then site A is assumed to be able to communicate with site C. The converse of this assumption of transitivity states that if site B is not accessible to site A then no other site can communicate with it either. This assumption is crucial to recovery protocols wherein if a site is detected as having crashed by any other site, then it is assumed that none of the remaining sites in the network can communicate with the crashed site. Section 5.9 on error recovery discusses this protocol in more detail.

Replication at the file and directory level is provided to increase file availability and reliability in the face of node failures. Reliability is augmented by a simple atomic transaction mechanism which allows all operations bracketed within the 'begin trans' and 'end trans' calls to execute atomically. This transaction mechanism supports failure atomicity, wherein if a transaction completes successfully, the results of its operations will never subsequently be lost and if it is aborted, no action is taken. Thus, either all or none of a transactions operations are performed. The transaction mechanism is built on top of a locking mechanism and the system automatically locks data base files in the appropriate mode when they are opened. Deadlock problems are avoided by ensuring that processes never wait for files to be unlocked, and an error message is returned if are not available.

Protection has not been considered an issue in this thesis. All files are created readable and writeable by every one. Authentication can be implemented at a later date by making a few changes to the name server date base format and associating each file's record in it with a protection code. This code can then be checked on file access to see whether the user has valid permission to read or write the file. In the current implementation as long as the named file and its parent exist in the name server, file access can take place and no other directory searching is done.

4.0. DFS Architecture:

In order for a distributed system to be convenient to use, it is important that there be a global way to name files, independent of where they are stored. This is accomplished in the DFS by organizing the files into a single logical file system (LFS). The LFS is a single tree where internal nodes are directories and leaf nodes are ordinary text files. Thus the users view of the file system is analogous to a single centralized Unix environment with globally unique names in a single, uniform hierarchical name space. There is only one logical root for the tree in the entire network. As in Unix, files are named by giving a full pathname or a relative pathname with respect to a current working directory.

Files and directories in the logical tree are mapped onto multiple physical file systems [ANDR 87] at different sites. These physical files are stored at fixed locations in the Unix hierarchy under system maintained physical file names (PFN). Figure 4.1 depicts this mapping.

All file names are thus global and users have access to their files only through the logical name space. The system maintains a mapping from logical file names to physical file names through a name server. The name server is represented as a regular typed replicated DFS file. The name server is assumed to exist at all potential current synchronization site locations and the system takes responsibility for keeping the several images consistent.

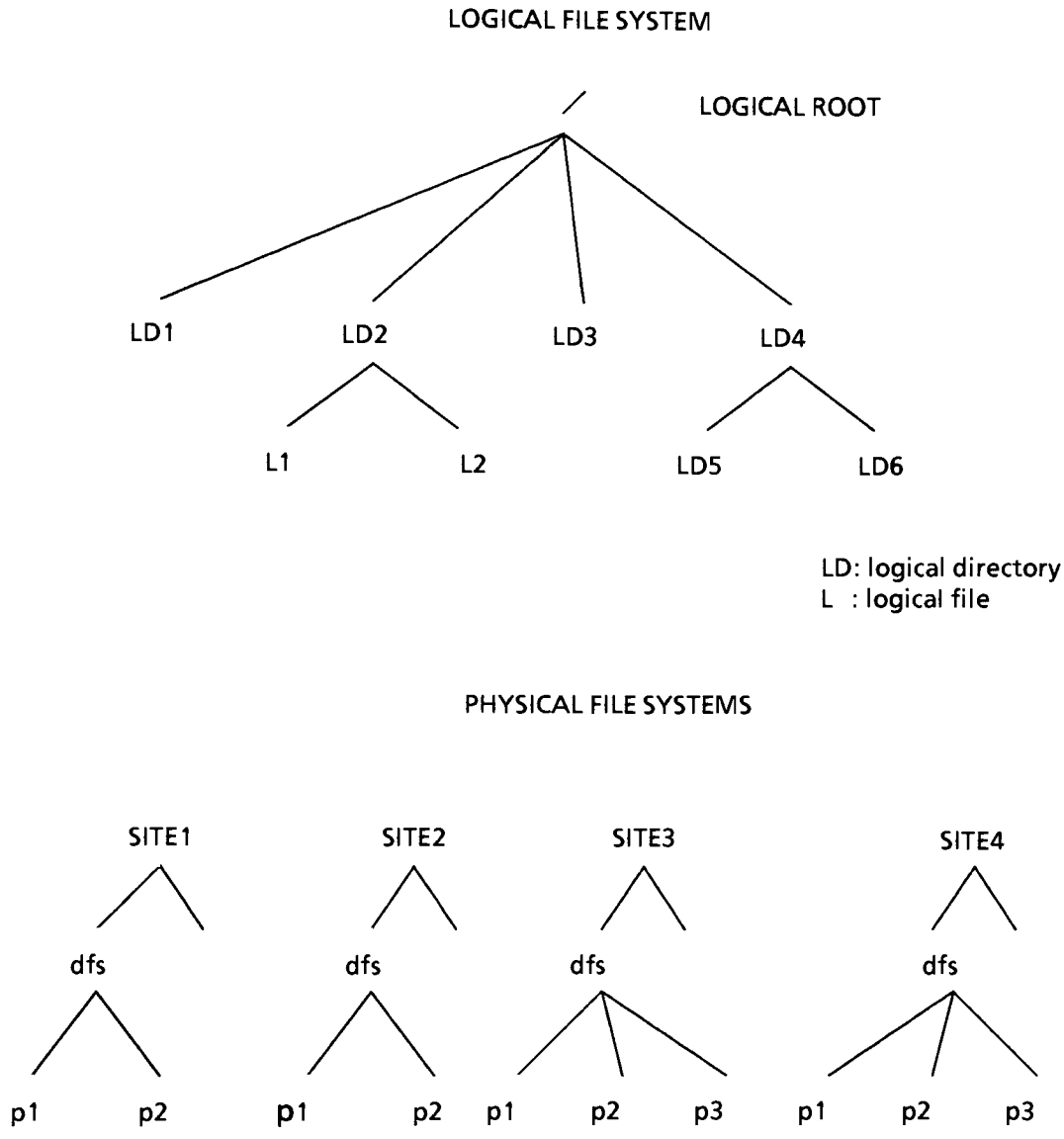
4.1. Logical Sites for file system activity:

File system activities are based on the Locus concept of using sites (US), storage sites (SS) and current synchronization sites (CSS). These logical sites are defined to be:

a. Using site: This is the site which issues commands to either the CSS or the SS.

b. Storage site: This is the site at which a copy of the requested file is stored and

c. Current synchronization site: This logical site enforces a global access synchronization policy for the set of replicated files and selects SS's for each 'open' request. A given physical site can be the CSS for any number of using sites but there is only one CSS in any given partition at any given instant of time.



Logical - physical filename mapping

LD1 -> <p1@site1, p2@site2> (2 images)

LD2 -> < p2@site1, p3@site3> (2 images)

L1 -> < p3@site4> (1 image)

Fig. 4.1. DFS File name mapping

Thus there are three independent roles in which a site can operate. The DFS is designed so that every site can act as a full function node so any site can be the US, CSS or the SS or any combination of the three as shown in Figure 4.2.

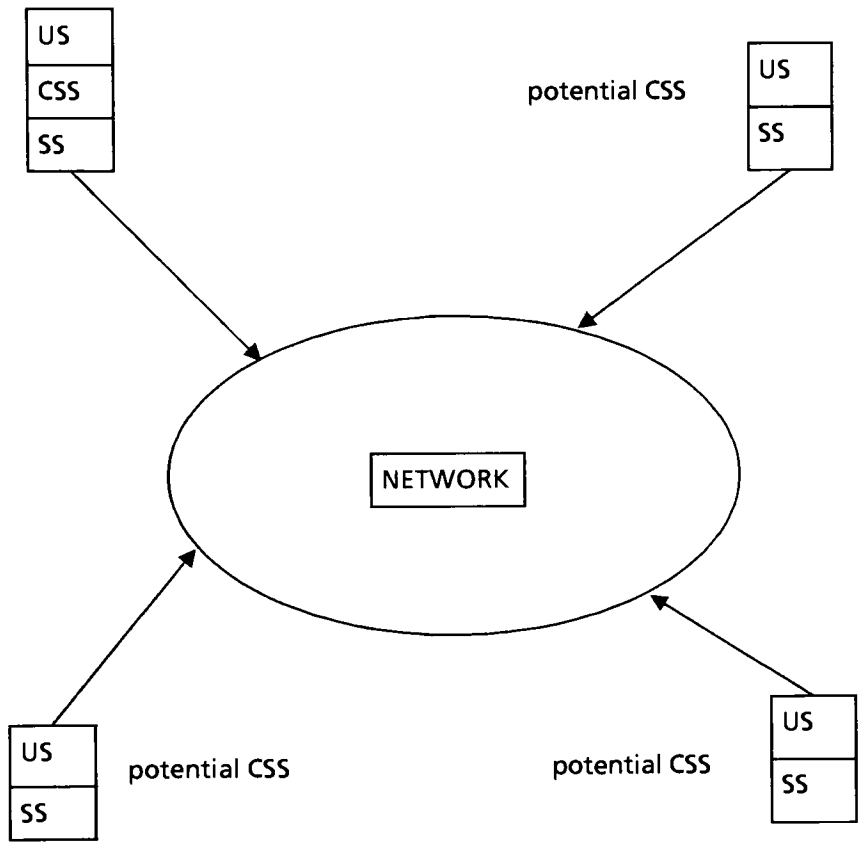


Fig. 4.2 DFS logical sites

5.0 DFS implementation

Major issues common to most distributed file systems are:

* User interface
* Naming
* Inter process communication
* File access mechanism
* File consistency
* Synchronization
* Locking
* Atomic transactions
* Error recovery

This section discusses these issues and describes the DFS implementation with respect to these issues.

5.1 User interface:

The DFS presents users with two command interfaces: the *shell interface* and the *data base interface.*

5.1.1. The shell interface:

The shell interface is a Unix like command interface that allows manipulation of files (unrestricted streams of bytes) within the hierarchical logical file structure. In general terms a command is expressed as

command arg1 [arg2, arg3 ... argn]

Two modes of operations are supported in this interface: default mode and explicit mode. The default mode is the normal mode of operation wherein the system makes use of a version number associated with each file to access the most up to date file image. This mode allows complete location transparency thus relieving the user of having to remember where files are located.

To afford more flexibility over file manipulation, the explicit mode of operation allows users to override this version number mechanism and specify the site at which a file image is located in the command line. Explicit mode is specified by :

command arg1@site1 [arg2@site2, arg3@site3 ...]

The system can be queried at any time to find out where the files images are located. This mode is useful if the file replication mechanism is to be avoided. Files created in default mode are created at more than one randomly selected site whereas in explicit mode the file is only created at the location specified by the user. The explicit mode can also be used to copy images from one location to another. This is useful in the event that file images become inconsistent due to one or more sites crashing while the file is being updated Under normal circumstances, the system attempts to make the images consistent during the next file access, but this can also be done manually using the explicit mode of operation. A complete list of commands available to users appears in Appendix A.

5.1.2. The data base interface:

The data base interface is also a command interface much like the shell interface and allows access to special files called data base files. Data base files created in this interface are typed DFS files with all access taking place one record at a time. Commands to create, read, write, remove and rewind a data base file are supported.

This interface has no concept of current working directories or different modes of operation. Default mode is the only mode of operation and all data base files are created in the root directory. In addition an atomic transaction mechanism has been implemented that allows a series of reads and writes to be considered as one atomic action. Transactions are bracketed by 'begin trans' and 'end trans' or 'abort' system calls. All writes to the database are made to a temporary file and written to the data base only in the event of a 'end trans' call and discarded otherwise.

The transaction mechanism is enforced in this interface and it is considered an error if the 'begin trans' system call ('bt' command) does not precede each database session. The transaction has to be restarted if more operations on the data base are to follow. Transactions are discussed in more

detail in section 5.8. A complete list of commands available to users in this interface appears in Appendix B.

5.2 File name mapping:

An identifier or a name is a string of symbols, usually bits or characters, used to designate or refer to an object. These names are used for a wide variety of purposes like referencing, locating, allocating, error controlling, synchronizing and sharing of objects or resources and exist in different forms at all levels of any system [WATS 85].

If an identified object is to be manipulated or accessed, the identifier must be 'mapped' using an appropriate mapping function and context into another identifier and ultimately into the object. A name service provides the convenience of a runtime mapping from string names to data. An important use of such a service is to determine address information, for example mapping a host name to an IP address. Performing this kind of address look up at run time enables the name server to provide a level of indirection that is crucial to the efficient management of distributed systems. Without such a run time mapping, changes to the system topology would require programs with hardwired addresses to be recompiled, thus severely restricting the scope for systems to evolve.

These name to data mappings may consist of a logically centralized service eg. ClearingHouse [OPEN 83] or a replicated data base e.g. Yellowpages [WALS 85]. The DFS follows the second approach for file name mapping. To attain location transparency, one must avoid having resource location to be part of the resource name. This is done by dynamically mapping global logical user defined file names to multiple physical file names.

Mapping of logical to physical file names is maintained in a DFS regular replicated file called the name server. Since this file contains system information, it is typed and contains only fixed length records (one record for each file created within the system). Each record in the name server maintains information about a DFS file such as the file type (whether it is a file or a directory) , the number of images, the files logical name, the locations of the file images and the version numbers of the file images. Since all access to any file must go through the CSS, after opening a file, the file

descriptors obtained at the US and the CSS uniquely specify the file's global low level name and is used for most of the low level communication about open files.

MAXCOPY images of the name server are created at system boot time at potential CSS locations. Since access to the name server is only through the CSS, each potential CSS will know the locations of the name server images. Reads from the name server may take place from any image with the highest version number. However, since the name server is heavily used, writes are made to the local image when possible thus avoiding network traffic and speeding up access. An error during the write results in failure of the system call and no attempt is made to contact any other image. This enables the CSS to always store the most recently updated image and enables it to keep it consistent. Other operations on the name server include deleting records.

In addition to the name server, the CSS also maintains a cache of the most recently read record in main memory for fast access and in the event the name server becomes inaccessible after a file has been opened.

This cache is in the form of a shared memory segment, since it will be referenced by future calls. On an open, once the name server has been read, and the record cached, no further services of the name server are required until the file is closed. Calls that refer to this cache are the ones associated with locking and transactions. On a close, the cache is deallocated, the version number incremented (if the file was opened for writing) and the record written back to the data base.

Since the name server is a regular replicated file, it may fall out of consistency due to site failure. Consistency of the name server is maintained by a 'watch dog' program running in the background, by checking the version vector associated with the name server at regular intervals. The first record in the name server data base stores version number information about the remaining name server images and this record is read when it is required to be made consistent. If the versions are found to be inconsistent, a copy of the local file image is sent out to the other lower version numbered image sites and consistency is restored.

5.3 Inter process communication:

Machines converse in the DFS by sending messages using an underlying protocol for reliable transmission. The DFS makes use of the Transport Level Interface (TLI) [AT & T 87]mechanism of the Transmission Control Protocol / Internet Protocol (TCP/IP) suite of protocols [POST 81a] and [POST 81b] implemented on the AT&T 3B1 machines, to allow processes on different machines to communicate with each other. The TLI layer provides the basic service of reliable end to end data transfer needed by applications. The kernel structure consists of three parts: the transport interface layer, the protocol layer and the device layer as shown in Figure 5.3. The transport level interface

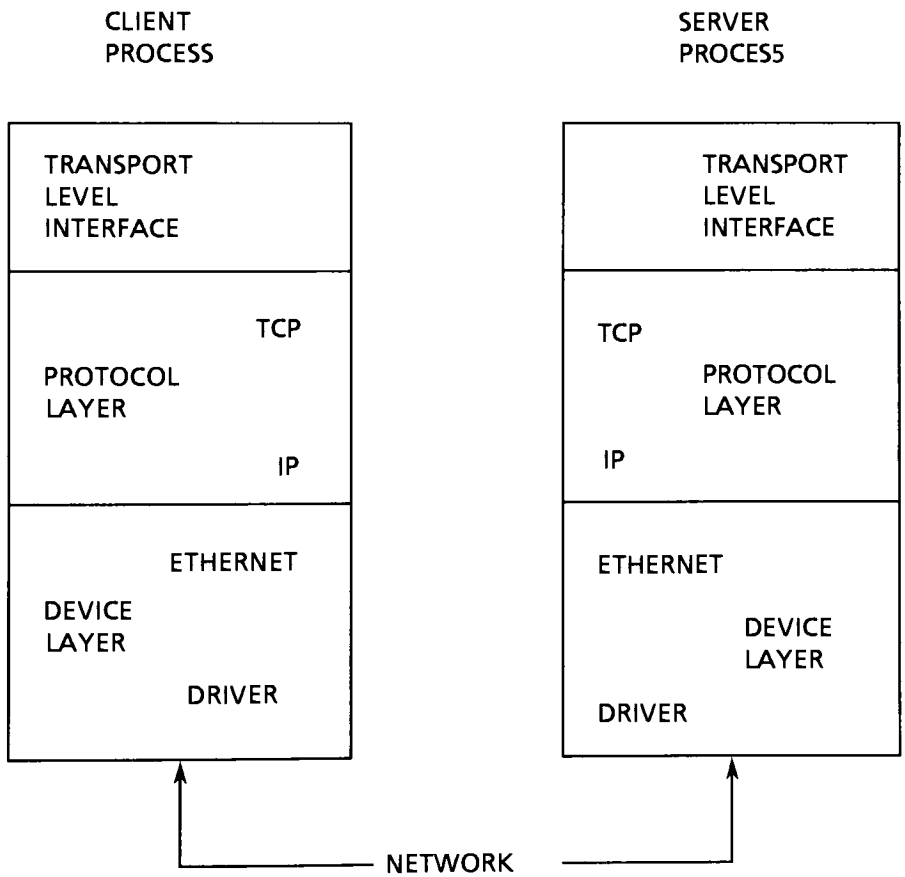


Fig. 5.3 Transport interface model

provides the system call interface between application programs and the lower layers. The protocol layer contains the protocol modules used for communications and the device layer contains the device drivers that control the network devices.

All network messages in the system require an acknowledgement response message from the serving site. The response message, in addition to telling the requesting site that the call was successful, also returns additional information that is produced as a result of the call. Moreover extensive file transfer is carried out by the system in keeping the name server and user files consistent. The connection mode of communication, being circuit oriented, seemed particularly attractive for this kind of data stream interaction and would enable data to be transmitted over the connection in a reliable and sequenced manner. Hence the connection mode service of the transport level interface was chosen for use in the DFS so that higher level system routines are assured of reliable communication.

Every host that wishes to offer service to users (remote or local) has a process server (PS) through which all services must be requested [TANE 81]. Whenever the process server is idle, it listens on a well known address. Potential users of any service must begin by establishing a connection with the process server. Once the connection has been established, the user sends the PS a message telling it which program it wishes to run (either the CSS or the SS). The process server then chooses an idle address and spawns a new process, passing it the parameters of the call, terminates the connection and goes back to listening on its well known address. The new process then executes either the CSS or SS programs, executes the appropriate system call and sends its reply back to the calling process.

Addresses in the Internet domain are composed of two parts, the host network address (identifying the machine) and a port number which is simply a sixteen bit integer that allows multiple simultaneous conversations to occur on the same machine to machine link. These two parts are commonly referred to as a socket and uniquely identifies an end point for communication.

In the current prototype implementation, all process servers are assigned one unique port number which is hardcoded in the program. In an operational implementation, the requesting site or US would look up a (Unix) system data base usually found in /etc/services for the port number of

the process server. All requesting sites can also read a (DFS) system file to find out the current location of the CSS. This string name is used to query the Unix system and the appropriate network address of the process server is thus obtained. The two parts of the Internet address are now known and this enables the requesting site to construct the complete address of the process server and thus establish a connection.

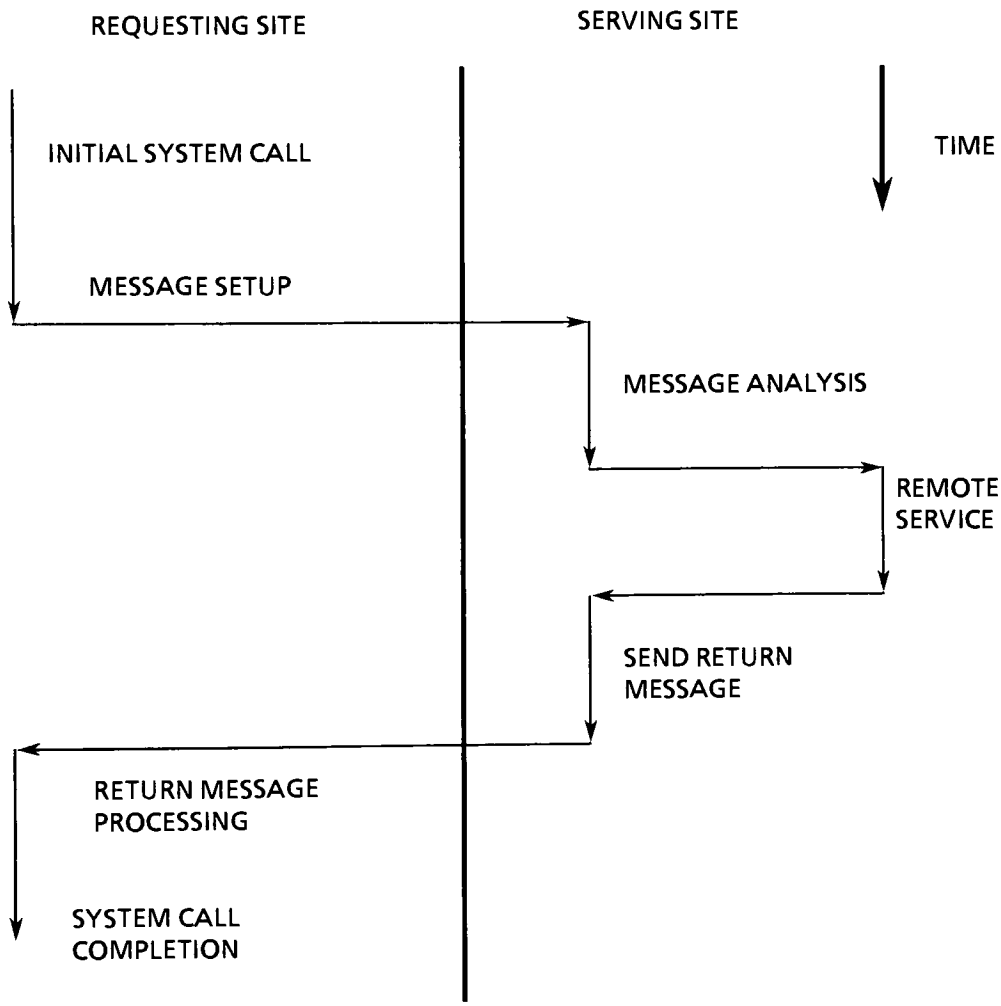


Fig. 5.4 Communication sequence

To obtain service, the requesting process packages up a request message on behalf of the user and sends it to the process server and then blocks, awaiting the outcome to be communicated in the opposite direction by means of a response message. This one to one response interchange is the fundamental access mechanism. The model is of the form shown in Figure 5.4.

All send and receive messages are of fixed length with each message containing control information which specifies the type of message and related system call parameters. These message formats [BACH 86] are shown in Figure 5.5.

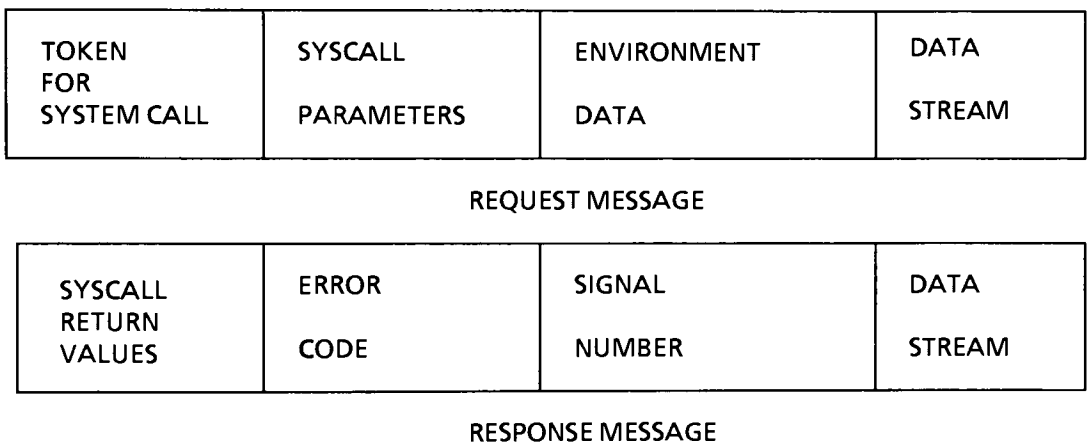


Fig. 5.5 Message formats

The request message consists of a token that specifies the logical site to which the message is destined (CSS or SS), a token that specifies the system call the process server should execute on behalf of the client, parameters of the system call and environmental data such as user id, current directory, user hosts address etc. The remainder of the messages contains space for 512 bytes of user data.

The responding site waits for requests from the requesting site. When it receives a request, it decodes the message, determines what system call it should invoke, executes the system call and encodes results of the call into a response message. The response message contains the return values to be returned to the calling process as a result of the system call, an error code which tells the

requesting site whether the call completed successfully and a signal number which passes the requesting site additional information on the call and a fixed length (S12 byte) data array. The requesting site then decodes the response and returns the results to the user.

5.4 File access:

All files which are to be read or written, locked or unlocked need to be opened before any operations can proceed. Since there are three logical sites taking part during file access, state information on the call needs to be maintained at all three sites. In fact this is one of the inherent problems associated with the integrated model of distributed systems as error recovery becomes all the more difficult since this state information needs to be reconstructed in the event of site failure.

The data structures at the US and CSS keeping track of open files are very similar to those used by Unix. In Unix, four levels of data structures are used to record information on open files. Each application process refers to open files using a small non-negative integer called a file descriptor. The file descriptor is used as an index into the first data structure, the user file descriptor table. The user file descriptor table is used to map process file descriptors into entries in a global file table which is the second data structure level. The global file table holds the open mode (read or write) and a pointer to a table of inodes which is the focus of all activity relating to the file. Multiple file table entries could refer to the same inode. The inode stores all file attributes including page pointers, file protection modes, directory link counts etc. The final data structure is the buffer cache which stores modified pages. The file system data structures in Unix are shown in Figure S.S.

The user file descriptor and global file tables are similarly implemented in the DFS. At the US, the file table points to a table of 'unodes' while at the CSS, the file table points to a table of 'cnodes'. Both these tables store information that is relevant to the logical site. The file descriptors obtained at the US and the CSS uniquely specify the file and is used as the low level name of the open file for most internal references to that file.

To read a file, users can issue the 'open' command at the shell interface level. The shell packages up the request and sends the request to the US. The US allocates a slot in its file descriptor table, its

file table and its unode table, thus obtaining a local file descriptor (lfd). A message is then sent to the CSS and the US blocks until it receives a response from the CSS. The CSS location is obtained by examining a system file (csslist) which contains a list of potential CSS locations. The first entry is the current CSS. If unode information is already available at the US (if the file was recently opened), the last modification time of the file is included in its message to the CSS. If the file has not been modified since it was last accessed, the cached copy of the file is used for subsequent reads.

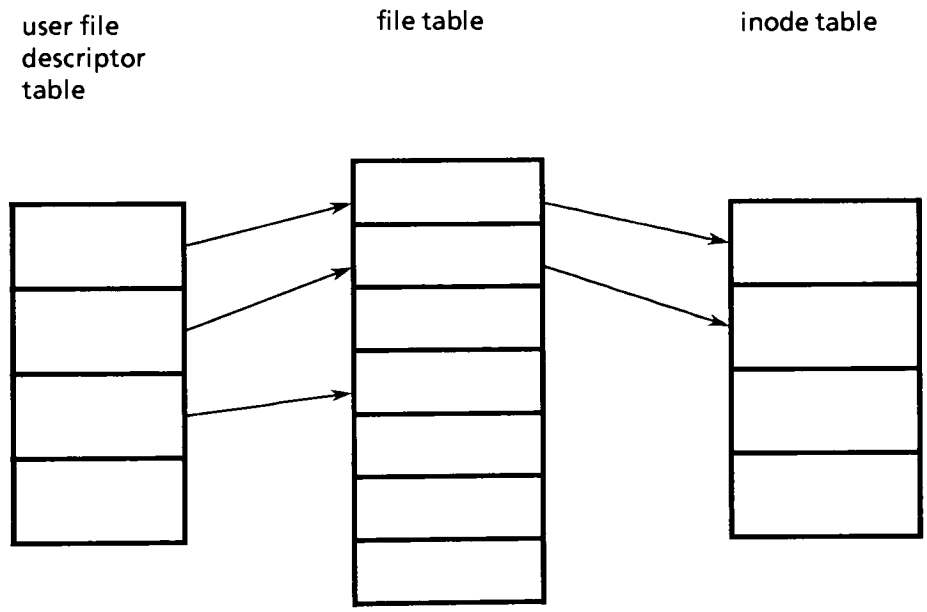


Fig 5.5 Data structures in Unix

Just as at the US, the CSS first allocates slots in its file descriptor table, file table and cnode tables, thus obtaining another file descriptor for the file (cfd). It then reads the name server and obtains the locations of the file images. The version vector associated with the file is examined and if any inconsistency is detected, an attempt is made to make the images consistent. A message is then sent to each image site requesting the SS to open the file, including the <lfd,cfd> pair in its message.

The SS maps the logical file name to the <lfd,cfd> pair, checks to see whether the file is locked in a conflicting mode by any other process and replies appropriately to the CSS.

On receipt of replies from all the image sites, the CSS then completes the information in its cnode table and returns a list of successfully opened image sites to the US.

The US in turn completes the information at its unode and returns the local file descriptor to the user. This file descriptor is used in all future references to the file. The open protocol is shown in figure 5.7.

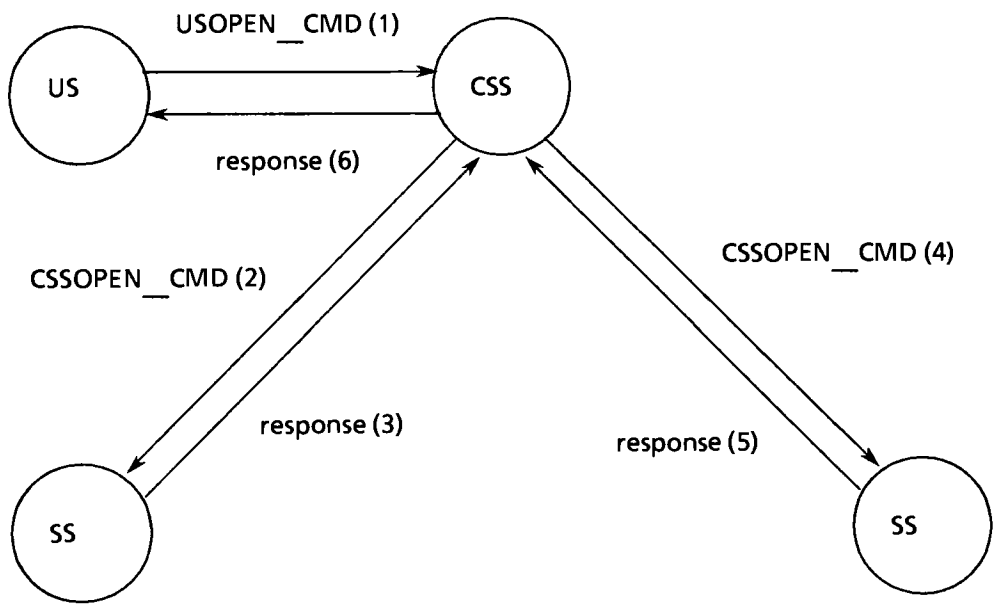


Fig. 5.7 Open protocol

After the file is opened, the user can issue read and write commands. These are directed to any one of the opened images. The read protocol is shown in figure 5.8. Reads and writes refer to the site vector to obtain site locations. In the event of a site crashing while the the US is reading or writing, it simply locates an alternate SS (provided one exists) and restarts the operation. On a write, the SS that was successfully written to is marked as the primary site. All subsequent reads are made only from the primary site. After one of the sites has been designated as the primary, the system call fails if the primary fails.

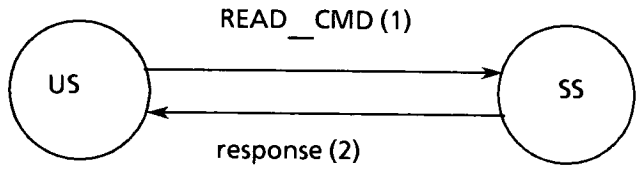


Fig. 5.8 Read protocol

Once the file is done issuing reads and writes, the 'close' command is sent to the primary site. The close call deallocates data structures at the SS and the request is then sent to the CSS. The CSS deallocates data structures and sends the request to the remaining file images. The close protocol is shown in figure 5.9.

The 'create' command works very similarly to the open call except for the fact that newly created files are made temporary by attaching a ".tmp" extension to its physical name. These temporary files are produced by processes creating files and then crashing and can be cleaned up by a garbage collection routine as part of the system administrators duties. All writes to a newly created file are committed on the close call.

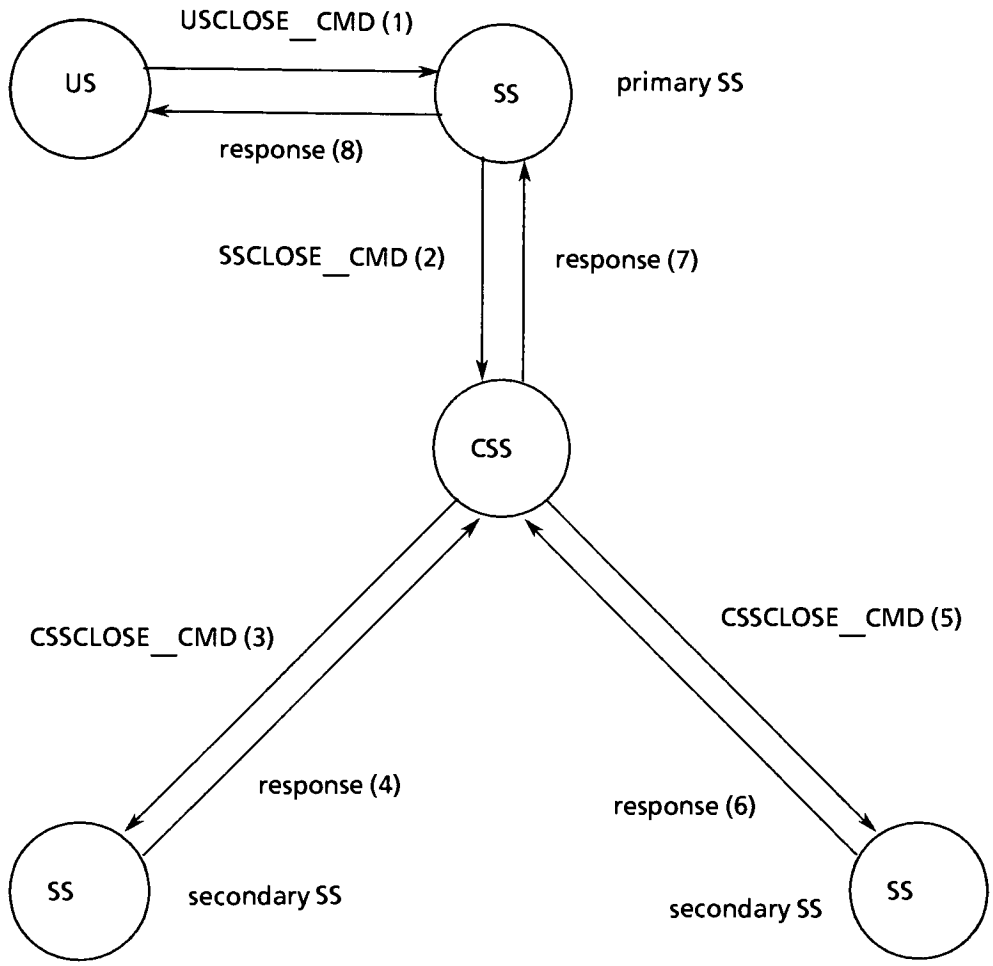


Fig. 5.9 Close protocol

5.5 File consistency:

Since there are several images of the same file, it is important that these images remain consistent across site failure. Inconsistencies in the images arise when one or more image sites crash after the file has been opened for update or during update. It is up to the system to keep these images consistent.

For the purposes of the consistency algorithm, all files may be in one of two states : If the CSS accessing a particular (possibly replicated) file set is in contact with at least a majority of the file images, then the files are said to be in the STABLE state. When the CSS accessing a file set is in contact with less than a majority of the files then the files it cannot contact are said to be in the RECLUSE state.

File image consistency in the DFS is maintained by a simple version number mechanism consisting of an <original version number, current version number> tuple. The *original version number* keeps track of partitioned updates. The original version number is initialized to zero at file creation time and indicates that either no updating has taken place while the file was in the RECLUSE state or any partitioned updates have either been reconciled or resolved with a majority of the file copies. In the event of a partition, if the original number is zero, it is set equal to the current version number. All files in the DFS have the *current version number* initialized to unity at file creation time and this number is incremented every time the file is updated.

Thus each file image has its own version vector with the name server keeping track of the version vector of each file. On each file access (open), the name server is first read and the version vector of the file is obtained. The system then opens the images with the highest version number, thus presenting the user with the most up to date version of the file.

The consistency algorithm is illustrated by an example: Assume that a file is triplicated so that any two copies form a majority. Denote these copies by A, B and C. The version vector associated with these three images is A < 0,2>, B <0,2> and C <0,2> after one update. Now assume C is detached and goes into the RECLUSE state.

Two cases arise:

i. If A is updated, then this results in a version vector of A<0,3>, B <0,3> and C <0,2>. This presents no conflict and C can be made consistent when it comes back on line using the file images at A or B.

ii. If both A and C are updated independent of each other it results in a version vector of A<0,3>, B <0,3> and C <2,3>. Since the original version number of C is greater than zero but less than the current version number of A and B, both partitions have been updated and a conflict arises. Conflicts can then be resolved by the user.

The original version number is unused in the current version of the implementation since it is assumed that the network does not partition into independently operating subnetworks.

5.5.1 Resolution of inconsistency:

An attempt is made to resolve any inconsistency in user's files at the time of access. When the file is opened for reading or writing, the current version number associated with the file is checked. If any inconsistencies exist, a copy of the image with the highest version number is sent to the remaining images. If any of the images are un-reachable, the system will attempt to make the files consistent on the next access. The same mechanism is used to keep directory files consistent since they are treated in the same way as regular files. Directory files are made consistent during the 'ls' command.

The name server is also a replicated DFS file, and also has to be kept consistent. Since it is a system file, and since a consistent name server is crucial to the correct working of the system in the event of CSS failure, it is made consistent in background independent of user access. The 'watch dog' process checks the version vector associated with the name server at regular intervals and sends a copy of the highest version numbered image to the other sites.

5.6. Synchronization:

Synchronization in the face of multi-user operations is handled in the DFS at two levels: the system level and the application level.

5.6.1 System level synchronization:

Each US executes only one command at a time and hence the issue of synchronization does not arise. There could however be more than one user attempting to get service from the US potentially at the same time. The shell and data base interfaces communicate with the US through a named pipe (FIFO) and the semantics of the Unix pipe mechanism takes care of synchronization here by queuing simultaneous requests at the writing end of the pipe. The shell / dbms attempts to open the pipe for writing, a finite number of times in non blocking mode. If the open fails either because the US is busy servicing other client requests or if the pipe is full, the call returns with an error. Thus they never block indefinitely waiting for service. Once a write has been made to the pipe however, the calling process blocks on a read waiting for the US to return. The shell / dbms includes its return address in its message to the US so that the US knows to whom it must send its reply. An alarm signal (currently set at 120 secs) prevents the calling process from waiting indefinitely in the event of SS or CSS crashes.

However, since more than one user could potentially be accessing the CSS at the same time, the process server forks off a CSS process for each request it receives and goes back to listening on its well known address. Since some of the system calls need to refer to state information produced by a previous call, all data structures take the form of shared memory segments. These segments are protected by semaphores. The semaphore protecting a segment is required to be obtained before access to the data structure can take place. Processes block on a queue until the semaphore can be obtained, again relying on an alarm signal to prevent any deadlock or indefinite waiting caused by a process acquiring a semaphore and then crashing. Once a slot has been obtained in the data structure, it is marked as 'used' until the 'close' call is issued and the data structure deallocated. Subsequent accesses to the data structure by different processes thus do not disturb the contents of the slot while it is being used.

At the SS, state information is substantially less than at the CSS since it only keeps track of objects that are local to it and hence data is simply written to a file. Here also, since there could be multiple

SS processes running concurrently, the data structures are protected by semaphores similar to that at the CSS.

5.6.2 Application level synchronization:

A synchronization mechanism is also required by user level processes since more than one user could be accessing a file or data base at the same time. The DFS provides a distributed implementation of file locking with file level granularity to allow concurrent access to files. In standard Unix, multiple processes are allowed to have the same file open concurrently. These processes can issue read and write system calls with the system only guaranteeing that each successive operation sees the effects of the ones that precede it. In other words a read on a file could see intermediate data if another process is writing to the same file at the same time. This lack of synchronization is implemented in the DFS as the default locking policy and is called Unix mode.

Users can issue 'shared' or 'exclusive' lock commands on files after opening them. The semantics of the locking mechanism allow multiple readers or a single writer. Although locking may be seen to be advisory (ie. checked only during the lock system call) it is in reality checked on all reads and writes to prevent processes from accessing a file in the event that some other process has set an exclusive lock on it or a previously set lock has timed out.

While locking at the shell interface is left to the user, it is mandatory at the dbms interface level with the system automatically locking files when they are opened, transparently to the user. Section 5.7 discusses the locking implementation in more detail. In the data base environment, the atomic transaction mechanism built on top of the concurrency control mechanism (locking) provides failure atomicity to data base files. Section 5.8 on transactions discusses this mechanism in more detail.

5.7 Locking:

The DFS provides a distributed implementation of file locking with file level granularity to allow concurrent access to files [LEVI 87]. Locking resolves contention among multiple processes wishing to access the same file at the same time.

The mechanism that provides concurrency control in the DFS is referred to as the distributed lock manager. The lock manager offers the means by which a process may obtain access to and control over an object so as to block other processes that wish to use the object in an incompatible way. An instance of the lock manager exists on every network node and each lock manager serves to control concurrency for the objects local to it. Each lock manager keeps its own lock data base that records all locks on local objects, with the CSS keeping track of locks on replicated object images.

The lock manager facility supports two locking modes: SHARED locks (multiple readers) and EXCLUSIVE locks (single writer). Other operations include UNLOCK and operations necessary to inquire about a particular lock on an object currently locked at a particular node. Special commands at the shell interface allow users to set shared or exclusve locks on files. In the data base interface, locking is automatic and is the platform on which the transaction mechanism is built.

The lock synchronization rules [WALK 83B] are shown in figure 5.10. The lock manager enforces compatible use of an object by refusing to grant conflicting lock requests. Locks are either granted immediately or refused. Processes never wait for locks to become available, so there is no possibility of deadlock (though indefinite postponement is possible). While this may be suited to human time span locking uses such as file editing, it may not be ideally suited for data base types of transactions.

		Process A		
		Unix mode	Shared lock	Exclusive lock
Process B	Unix mode	read/write	read	no access
	Shared lock	read	read	no access
	Exclusive lock	no access	no access	no access

Fig. 5.10 Lock synchronization rules

To acquire a lock on a previously opened file, the US sends a lock request to the primary SS. The SS consults its lock manager as to whether the lock is to be granted. If the file is already locked in a conflicting mode by another process, it denies the lock request and replies to the US. Otherwise it sets the lock on the local object and sends the lock request to the CSS. The CSS obtains the locations

of the remaining images by reading the cached copy of the name server record associated with the file. It then sends the request to the remaining sites and awaits the response from them. If all sites agree to grant the lock request, the CSS grants the lock request and replies appropriately to the SS. If however any on the sites refuse the request, the lock request is denied and a response is sent back to the SS. The SS then unlocks the file it had previously locked and replies to the US. Figure S.11 shows the locking protocol structure.

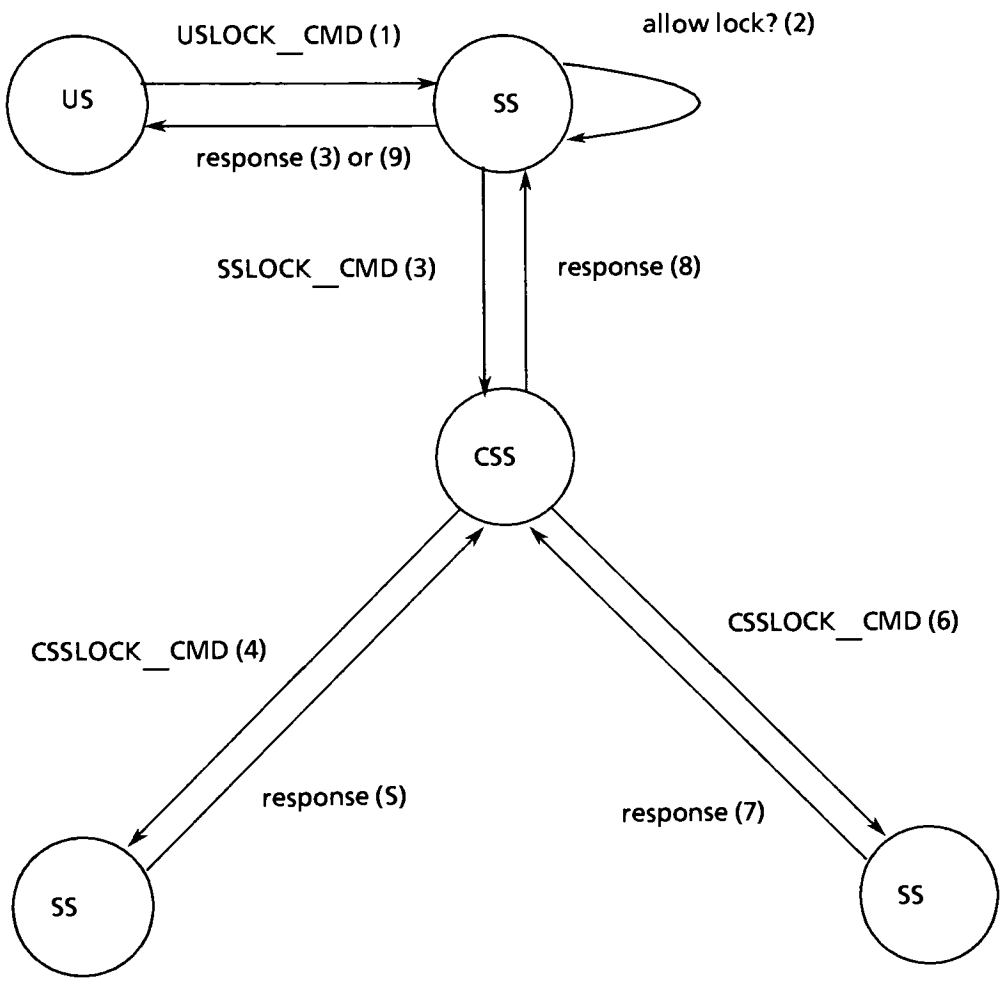


Fig. S.11 Lock protocol

Each SS that stores a copy of the file keeps track of open/ locked files by means of a 'lock list' table. Each record in the table holds the file pointer which keeps track of the number of bytes read

or written, the physical file name and an array of 'lock nodes'. An array of lock nodes is needed because multiple processes may have the same file locked at the same time. Each process that wishes to lock the file has its 'uid' added to the lock node, provided there is no conflict in the lock mode. When the process unlocks the file, its uid is removed from the lock node.

An existing lock can be upgraded from a shared lock to an exclusive lock provided the users uid matches that existing for the shared lock and provided no other processes have shared locks on the file.

A file could alse be permanently locked by a process that locks the file and then crashes. To prevent this from happening, all locks are time stamped and if a timer expires after a given interval, the file is automatically unlocked and the next reference to the file returns an error message.

5.8 Atomic transactions:

The DFS is not by itself a data base system; it provides no built in access mechanisms for files other than random access to sequences of bytes. Since the DFS is intended to support applications dealing with data bases, it provides mechanisms that they can use. The atomic property for file actions is the most important of these mechanisms. A sequence of reads and writes on some set of files (not necessarily residing at the same site) can be performed as an indivisible atomic operation. Such an action is called an atomic transaction. Atomic transactions are useful in distributed systems as a means of providing operations on a collection of computers even in the face of adverse circumstances such as concurrent access to the data involved in the actions, and crashes of some computers involved due to hardware failures [LAMP 85].

Consider the problem of crash recovery in a data base storage system, which is constructed from a number of independent computers. The portion of the system that is running on some individual computer may crash and then be restarted by some crash recovery procedure. This may result in the loss of some information present just before the crash. The loss of this information may in turn lead to an inconsistent state for the information permanently stored in the system.

Atomic transactions are meant for just these kinds of problems and enable the stored data to remain consistent. A transaction is a sequence of reads and write commands sent by a client to the file system. The write commands may depend on the result of previous read commands in the same transaction. The system guarantees that after recovery from a system crash, for each transaction, either all of the write commands will have been executed or none will have been. In addition, transactions appear indivisible with respect to other transactions which may be executing concurrently ie. there exists some serial order of execution which would give the same results. This defines the atomic property of transactions.

A transaction can be made atomic by performing it in two phases; first record the information necessary to do the writes in a set of intentions without changing the data stored by the system and second, when the transaction is committed, do the actual writes changing the stored data. If a crash occurs after the transaction commits but before all the changes to the stored data are done, the second phase is restarted. The restart is done as often as necessary to ensure that all changes have been made.

The atomic transaction mechanism centers around the concept of an atomic restartable action. An atomic action is characterized by :

```
procedure A      = begin
                     save_state;
                     R;
                     reset;
                   end;
```

If R is a restartable action then A is an atomic action as can be seen by the following case analysis. If A crashes before save_state, nothing has been done. If A crashes after reset, R has been done and will not be redone because the saved state has been reset. If A crashes between the save_state and reset, A will resume after save_state and restart R thus ensuring that A is completed successfully.

5.8.1 Transaction Implementation:

Transactions are bracketed by two special operations, 'begin_trans' and 'end trans' On a commit, all writes are committed to the data base. In addition an 'abort' call discards the writes and makes no changes to the data base. The following data structures are used in the implementation:

* A *transaction identifier (tid)*: a unique identifier
* An *intention* is a record containing:
  - n: the low level name of the file <lfd,cfd>
  - a: an action (Read or write)
  - P: a pointer to the data to be written

* A *transaction list* containing zero of more of the following:
  - t: a tid
  - ph: the phase of the transaction (nonexistent,running,committed,aborted)
  - i: a set of intentions

* A *lock node* containing:
  - t: a tid
  - ph: the phase of the transaction
  - l: the type of lock (shared,exclusive, unlocked)

*A *lock list* containing:
  - pfn: the physical file name
  - ln: a set of lock nodes

These data structures are distributed as follows
* For each file at the SS holding a particular file image
  - a lock list
  - a semaphore protecting the list
* At each CSS, the coordinator of the transaction
  - a transaction list
  - a semaphore protecting the transaction list

To start a transaction, one sends a 'begin transaction' request to the CSS. The CSS finds the next available slot in its transaction list, sets the transactions to 'running' and returns an identifier (tid) which is the index of its record in the transaction list. All further messages in the transaction will carry this identifier.

The client then opens the data base file for reading or writing. As a first step in accessing data, opening the file sets a lock on the file. Locking resolves contention among clients attempting to access the same data at the same time. Entries are thus made in the SS's lock list indicating that the

file has been locked. Reads and writes are now issued, each call being directed to the primary SS. The SS sends each write message to the CSS along with the record to be written without making any changes to the data base. The CSS records the write request in its intentions list and sets the 'action' field, so that it knows that the current operation is a write.

The client then issues an end trans or an abort command to the CSS. The commit results in the locks being retained until the transaction has completed successfully. The transaction is marked as committed and an attempt is made to write the records in its intentions list to the various images of the data base. This action is an atomic restartable action and is repeated until the writes are completed successfully. An abort command discards the intentions list and marks the transaction as 'aborted'.

Thus the phases of the transactions goes through exactly three steps: nonexistent, running and then either committed or aborted. Only setting it to running can remove it from the nonexistent phase and only changing it to committed or aborted can remove it from the intentions list.

5.9 Error recovery:

The philosophy behind the error recovery protocols is to take action only when needed. In other words no state information abut crashed sites is maintained, so service requesting sites find out that a site has crashed only after explicitly trying to send it a message and receiving no response. While this may not be the best approach for error recovery in distributed systems, it is certainly one of the simplest. This results in some loss in transparency but it considerably eases the design of the protocols.

Error recovery depends on the logical site that has crashed and the actions that were under way at the time the failure occurred.

At the US, during the course of an open, if it is discovered that the CSS has crashed, it runs a CSS recovery procedure which elects another CSS and continues with the call. If however the file has already been opened and the CSS subsequently crashes, calls that depend on a previously opened file will fail. This is because as part of the CSS recovery procedure, all files opened with the crashed

CSS are closed automatically. If a call discovers that an SS has crashed, an alternate site is obtained from the files site vector and the call continues transparently to the user.

At the SS, distinction has to be made whether the crashed site is the US or the CSS. In either case all open files that belong to the crashed site are unlocked and closed. No CSS recovery procedure is run at the SS since it is assumed to be a passive repository for user files and takes no action on objects other than those existing local to it.

At the CSS, if the crashed site is the SS, an alternate site is located (if one is present) and the call continues. In the case of an open the system checks the sites where the files images exist. In the case of a create, the CSS reads a system file to obtain a list of potential SS's. It then polls these sites and creates the file at sites that are up. If the crashed site is the US, all open files that belong to the US are unlocked and closed. This entails sending messages to the SS's which hold the files images.

All opens at the CSS are timestamped. If too much time elapses after the open, before any other operation, the file is automatically closed. This is necessary since sites could conceivably open a file and then crash, leaving the file permanently open. Closing of open files needs to be done at regular intervals by the system administrator. All locks at the SS are similarly time stamped. Provided they are not retained locks, they will be unlocked after the timer times out. Retained locks result from a transaction locking a file and then being unable to commit the transaction. The retained flag is only reset after the transaction has been committed. The complete error recovery scenario is summarized in Table I.

One problem faced is the problem of detecting workstation crashes since the transport provider has no way of detecting this case. While the transport provider can detect the case of the process server being down, processes rely on an alarm signal to prevent them from waiting indefinitely if a machine crashes. When the alarm goes off at any logical site, it is assumed that the responding site is down (though it may just be a very slow server) and an error recovery procedure is run. An optimum value for the alarm is chosen depending on average network response characteristics.

Crashing site	Failure action
a. CSS	
i. If detected by US	Attempt to talk to alternate CSS
ii. if detected by SS	Unlock all files currently open by CSS Close all files currently open by CSS
b. SS	
i. if detected by US	Attempt to reopen file at alternate site
ii. if detected by CSS	Get an alternate SS which stores the specified file
c. US	
i. if detected by CSS	Close all files open by US Send message to SS that hold the files images to unlock and close the file.
ii. if detected by SS	Unlock and close all files currently opened by US

Table I

... when the alarm goes off at any logical site, it is assumed that the responding site is

6.0 Project Testing

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

6.1 Test Plan

The test plan consisted of two phases. The first phase consisted of testing the system with all sites up and running. No sites were assumed to crash. A small network of three sites was used in the DFS network configuration to simulate the US, CSS and SS. All DFS commands were tested out with this configuration.

The second phase constituted testing under simulated site crashing conditions to check if error recovery mechanisms were in order. Sites were crashed both by bringing down the process server at remote sites as well as by rebooting sites. The CSS recovery procedure was also tested during this phase to see if the system would continue to operate transparently to the user.

6.2 Test Results

Even though high performance was not considered to be a major issue in this prototype implementation, slow response time seemed to be a major drawback of the system. This can be evinced from Table II which tabulates some performance figures.

The main reason for this delay are the 'fork' and 'exec' calls being carried out by the process server for each system call executed. These calls are comparatively expensive operations and utilized the most timed during execution of the DFS system call. One way to overcome this would be to have eliminated the process server completely and have both the CSS and SS processes listen on their own well known addresses. This would mean that the US would have to know both these addresses before hand.

For comparision, the table also indicates the case when resources are local as well as when they are remote. Marginal difference between these two cases was to be expected since the system sends messages across the network even when resources are local to it.

	Standard Unix	DFS (one image)	DFS (two images)	DFS (resources local)
1. create directory	0.48 S	20.41 S	26.56 S	20.88 S
2. create file	6.56 S	20.25 S	25.90 S	
3. read file (1K)	5.83 S	51.35 S		

Note: All times are real time

Table II DFS Timing Statistics

Timing problems between communicating processes were discovered during testing. These problems were attributed to the lack of synchronization between the transport providers of the TLI mechanism on the individual machines. For example, if two commands (like 'read' and 'close') were executed in rapid succession, it caused the two communicating sides to fall out of synchronization with unpredictable results. The solution to this problem was to make the affected process 'sleep' for a while, giving time for the two sides to synchronize their operations. The amount of time was largely determined by trial-and-error.

Random asynchronous events caused by processes being terminated while they were communicating, was also found to occur. It was finally discovered that the cause of this was a software termination signal (SIGTERM) that was being sent by the CSS/SS processes to the process server after they were done executing a system call. The signal was to be caught by the process server and would decrement a variable that was used to control the number of child processes that it had spawned. It was thought that this measure of control would be useful to prevent the process table from filling up in the event the process server spawned too many child processes. Even though the signal was being sent only to the process server, it would somehow terminate other processes that had chosen to ignore it. When the signal was changed to a user defined signal (SIGUSR1), the asynchronous events stopped, however this method of controlling the number of processes did not prove fool proof as the signal was frequently lost under heavy process server usage. The count of the number of processes was finally maintained by incrementing a variable and writing it to a file and having the child process decrement this variable when they were done.

Two user inputted IP addresses indicating the starting and ending addresses of the machines on the DFS network was the basis for SS site selection at the time of file creation. If most of the machines with addresses between these two addresses were not on the network, the system spent a lot of time polling these sites simply to find out whether they were operational, eventually timing out. It was found that performance was better if the machines on the DFS network had IP addresses as close together as possible. Time out also occurred when reading or writing large files since I/O is a relatively slow operation. With the current time out set for 120 seconds, file transfer of up to 16 K could be carried out safely.

7.0 DFS CURRENT STATE AND FUTURE DIRECTIONS

This section describes the current state of the implementation, the shortcomings of the system and some the areas of extension where future work is possible.

7.1 File images:

In the current implementation, the system create an extra image of all newly created files making a total of two images per file. The image sites are chosen at random from a list of sites that are on the network. The number of images produced can be arbitrarily increased to improve reliability and availability. At one extreme would be the case of fully replicating each file at every network site. This would entail very high overhead both in the form of increased memory requirements as well as network message involved in keeping the images consistent.

7.2 Locking:

Locking in the DFS is only at the file level. In the event of a high degree of sharing especially in the data base environment, this could lead to poor response. Record level locking would lead to much better performance.

7.3 Optimization:

Reducing the number of network messages would also lead to better performance. A detailed analysis of individual protocols associated with system calls would indicate potential areas where the number of messages could be reduced without losing functionality or reliability. One obvious area of optimization is in the case of local files. In the current implementation, the CSS is the only logical site which checks to see whether the site it is sending a message to is local to it. If it is local, only a procedure call is required. At the US and SS however, even if the responding site is local, a message is sent over the network.

To make this kind of optimization possible, all three logical site programs the US, CSS and SS would have to be stored in the form of a library of procedures, much like an RPC mechanism. The US program could then be linked with the CSS and SS and the CSS and SS programs linked with each other.

7.4 Authentication:

A simple authentication mechanism allows the system to distinguish between regular users and the system administrator. This could be extended to include a 'password' as in Unix to allow only legitimate users access to the system.

7.5 Protection:

Files in the DFS are created readable and writable by every one. In other words they are not protected against accidental or malicious damage to their files. A protection mechanism could very easily be incorporated into the DFS by associating each files slot in the name server with an additional protection field. In the current implementation, during file access, the system searches the name server for the named file and its parent. As long as both exist, file access can take place.

To incorporate the protection mechanism, each component of a files complete path name would have to be checked for access permission in the name server. If any of the components deny access, then access to the file would be denied. This would also mean incorporating additional system calls to set and change access permissions .

7.6 Rebuilding state:

No state information on open files is reconstructed in the event that the CSS crashes after a file has been opened. This results in a certain loss of transparency as calls that depend on a file previously opened for update will fail and the file would have to be reopened at the new CSS. To enable complete transparency, the US would have to send a list of open files to the new CSS as part of its CSS recovery procedure which would then allow operations to continue even after a CSS has crashed.

7.7 CSS usage:

Since the CSS is global to the entire network, it is very heavily used. One way to reduce load on the CSS is to have multiple CSS's operating at the same time. The global logical file tree could be partitioned into several subtrees and each CSS could handle operations on the objects that were in its own subtree. This would however, make the CSS recovery protocols more complex. Another issue to be considered is whether to partition the name server into a true distributed name server, with

each image only keeping track of objects local to its subtree or whether a replicated name server would be more advantageous.

7.8 Network partitions:

Partitioning of a network into independently operating subnetworks is assumed not to occur. Another related assumption is the fact that if a site does not receive a response from another site within a certain time frame, it assumes that the other site has crashed and invokes failure handling routines to clean up its internal data structures.

In general a communication failure between any two sites does not imply a failure of either site as failure could be caused by transmission problems or unforseen delays. This failure may not be detected by all sites but may only be detected by one of the sites involved in the communication. To allow the system to continue operating in the face of network partitions, a partition protocol may have to be run to find fully connected subnetworks and subsequently a merge protocol to merge several such subnetworks into a fuller partition.

8.0 Conclusions

Although limited by its speed, this prototype implementation can be used as a learning tool to help understand the complexities involved in distributed systems The problems associated with maintaining state information were mentioned and this is an important dimension to efficient automatic error recovery. A trade off has to be made between the complexity of the protocols and the degree of network transparency afforded by the system.

The locking and atomic transaction mechanisms may be useful in implementing distributed data bases and other applications that need such mechanisms. The flat transaction mechanism implemented here could be the basis for a nested transaction mechanism to limit the effect of failure within a transaction.

The 'integrated model' of distributed systems seemed to have fitted in well with the primary goal of user transparency and is a departure from the more common 'client - server' model. Since all sites run the same software, logical sites may be interchanged freely making for a more flexible model and simpler implementation.

Work still needs to be done to provide robust behavior in the face of all manner of errors, race conditions, partial failures, heterogeneous environments etc. but can be used as the platform for further research in the field.

APPENDIX A

DFS SHELL INTERFACE COMMAND REFERENCE:

1. CAT - Concatenate and Print

   Usage:

   i. Default form: cat file - displays latest version of file on terminal

   ii. Explicit form: cat file@site -displays file located at site on terminal.

2. CD - Change directory

   Usage:

   i. Default form: cd - change directory to home directory

   cd directoryname - change directory to directoryname

3. CP - make copy of file

   Usage:

   i. Default form: cp file1 file2 - make copy of file1 and call it file2

   ii. Explicit form: cp file1@site1 file2@site2 - make a copy of file1 located at site 1 at site2 and call it file2.

4. LS - list contents of directory

   Usage:

   i. Default form: ls - list contents of directory.

5. MKDIR - makes a new directory

   Usage:

   i. Default form: mkdir directoryname - makes a new directory at local site and two images at remote site.

   ii. Explicit form: mkdir dirname@site - makes a new directory at site.

6. RMDIR - remove directories

Usage:

i. Default form:	rmdir directory	- removes directory at local site
ii. Explicit form:	rmdir dirname@site	- removes directory at site.

7. MV - renames file

Usage:

i. Default form:	mv file1 file2	- changes name of file1 to file2
ii. Explicit form:	mv file1@site1 file2	- changes name of file1 located at site1 to file2.
	mv file1@site1 file2@site2	- moves file1 at site1 to site2 and changes name to file2.

8. RM - remove file

Usage:

i. Default form:	rm file	- remove all file images.
ii. Explicit form:	rm file@site	- remove file located at site.

9. PWD - prints working directory

Usage :

i. Default form:	pwd

10. RL - applied shared lock (read lock) to file

Usage:

i. Default form:	rl file	- apply share lock to all file images.

11. WL - applied exclusive lock (write lock) to file

Usage:

i. Default form:	wl file	- apply exclusive lock to all file images.

12. UL - unlock file

Usage:

i. Default form: ul file - unlock all file images (all modes).

13. SF - show file attributes

Usage:

i. Default form: sf file - show attributes of file displays
< no. of images and locations, lock
status and version numbers >

14. OPEN - open the named file in append mode

Usage:

1. Default mode: open file - open all images

ii. Explicit mode: open file@site - open only one image

15. CREATE - create the named file

Usage:

i. Default mode: create file - create two images of the file

ii. Explicit mode: create file@site - create only one image at 'site'

16. CLOSE - close the named file

Usage:

i. Default mode: close file - close all images of the file

ii. Explicit mode: close file@site - close one image, file must have been opened
or created in Explicit mode.

17. READ -read the named file

Usage:

i. Default mode: read file - read the file and display on standard output

ii. Explicit mode: read file@site - read the image located at site.

18. WRITE - write to the named file

Usage:

1. Default mode: write file - write to the named file

ii. Explicit mode: write file@site - write to the file at 'site'

19. FTOP - reposition file pointer to top of file

Usage:

i. Default mode: ftop file

20. FLUSH - flush the data structures

Usage:

i. Default form: flush - must be sytem administrator to execute

Data structures at US, CSS and SS are cleared.

21. SET - displays users environment variables

Usage:

i. Default mode set - All open files are also displayed

22. UNSET - resets environment variable

Usage:

i. Default mode: unset variable

23. Q - quit from the DFS

Usage:

i. Default form; q - exits from the DFS shell and puts the user back in the

Unix shell.

APPENDIX B

DFS DATA BASE INTERFACE COMMAND REFERENCE

1. O Open data base file

   Usage:

   o file_name - open all images of data base file

2. C Create data base file

   Usage;

   c file_name - create data base file at two locations

3. P Put a record in the data base

   Usage:

   p file_name -the system prompts the user for record fields
   the record is written to a temporary file

4. G Get a record from the data base

   Usage:

   g file_name -the system prompts the user for record index

5. N Get the next record from the data base

   Usage:

   n file_name - read the next record from the current file
   pointer position

6. D Delete a record from the data base

   Usage:

   d file_name - the system prompts user for index of record

7. R Rewind the file pointer of the data base

   Usage;

   r file_name - reposition file pointer to top of file

8. BT Begin transaction

   Usage:

   bt - begin a transaction to encapsulate read and write operations on the data base

9. ET Commit the writes to the data base

   Usage:

   commit - all writes are committed to the data base and locks are retained

10. ABORT Abort a transaction

    Usage:

    abort -discard all writes made to the data base

11. CLOSE Close the data base file

    Usage:

    close file_name - close the file and deallocate data structures

12. Q quit from the data base shell

    Usage:

    q - returns the user back to the Unix shell

APPENDIX C

SYSTEM CALL INTERFACE TO THE UNIX KERNEL:

1. dfscreate (path, perms)

path - logical file name

perms - permissions

returns (file descriptor).

The dfscreate call behaves differently depending on whether the 'path' variable specifies default form or user form. If in default form, the dfscreate call creates the file at three system selected sites, designates one of them as the primary site and returns its file descriptor. In user form, the file is created only at the location specified by the user. The new file is assigned permissions as given by the 'perms' argument. If the file already exists, its length is truncated to zero, all file images opened for writing and the primary copy file descriptor returned. The file descriptor is internally generated at the US, which maintains a mapping between this local file descrptor and the file descriptor of the (possibly remote) SS.

2. dfsopen (path, perms)

path - logical file name

perms - permissions

returns (file descriptor).

The 'path' variable in this system call may specify default form or user form and the file must already exist for the call to return a file descriptor. In default form, the version number of each file image is inspected and the most recently accessed file is selected as the primary copy to be opened.

3. dfsclose (fd)

fd - file descriptor

returns (status).

The dfsclose call closes all file images that were opened. If the file was opened for writing, then an attempt is made to make all file images consistent by propagating the primary copy to the remaining sites and updating the version numbers.

4. dfsread (tid, fd, buf, nbuf)

tid - transaction identifier

fd - file descriptor

buf - address of user data buffer

nbuf - number of bytes to be read

returns (number of bytes read).

The dfsread call reads the 'nbuf' bytes pointed to by 'buf' from the open file represented by 'fd' It returns the number of bytes read, 0 on end of file or -1 on error. The dfsread call may deny access to the file if an exclusive lock has been applied on it by some other user. If the call is within a transaction then a shared lock is set on the file. If the file has been recently read, then it may exist in the buffer cache at the US. If it does, and the file has not been modified since the last read, the contents of the buffer is copied into 'buf' and returned to the user.

5. dfswrite (tid,fd, buf, nbuf)

tid - transaction identifier

fd - file descriptor

buf - address of user data buffer

nbuf - number of bytes to be written

Dfswrite writes the 'nbuf' bytes pointed to by 'buf' to the open file represented by 'fd' It returns the number of bytes read or -1 on error. The call may deny the request if the file is locked in exclusive mode by another user. If the call is within a transaction then an attempt is made to

acquire an exclusive lock on the file. In the event that lock acquiring fails, a finite number of re-tries is made. If all attempts fail, then the transaction is aborted.

6. dfsunlink (path)

path - logical path name

The dfsunlink call removes a link from a directory, reducing the link count by one. If a site is specified in 'path', only one file is unlinked, otherwise all file images are unlinked. If the link count goes to zero while some other process still has the file open, the file system will delay discarding the file until it is closed.

7. dfsmknod (path)

path - logical path name

The dfsmknod call is used to create directory files. 'Path' can specify default form or user form. If default form is specified, then three images at system selected sites are created.

8. dfschdir (path)

path - logical path name

The dfschdir call changes the current directory to the directory specified by it arguments.

9. dfslock (fd, flags, nbytes)

fd - file descriptor

flags - lock mode, one of {shared, exclusive, unlock}

nbytes - number of bytes to lock

returns (status).

The flags argument can have the following values:

LK_UNLCK - unlock file

LK__NBLCK - set an exclusive lock. Return with an error code if this results in an incompatible lock mode.

LK__NBRLCK - set a shared lock. Return with an error code if this results in an incompatible lock mode.

The file specified in the call must be previously opened before locking can be done. All file images are affected by this call. Locking granularity is at the file level in the current version of the DFS and 'nbytes' can only take on a value of zero (to indicate that the entire file is to be locked).

10. dfsrepl (path, site)

path - logical path name

site - site name

returns (status).

If default form is specified, three system selected sites are selected to replicate the file. In user form, the specified file is replicated at the specified site.

11. begintrans ()

returns (transaction identifier).

This call returns a transaction identifier that is an index into the transaction data structures at the CSS, used to keep track of all operations until an 'endtrans' call is encountered.

12. endtrans()

This call attempts to commit to disk, the results of all writes made within the 'begintrans' 'endtrans' envelope. If any of the writes fail, for any reason, the transaction is aborted and all writes are discarded, leaving the files untouched.

13. dfsshow (path)

path - logical path name

This call returns file attributes such as location, number of images, version numbers and locking information.

14. dfsmove (file1, file2)

file1 - logical path name

file2 - logical path name.

This call is used to rename file objects. Both *file1* and *file2* can specify default or explicit forms. If both are in default form, *file1* is simply renamed *file2*. If both specify explicit form, then the file is copied from the source site (specified by *file1*) to the destination site (specified by *file2*) in addition to being renamed. Any other combinations result in *file1* being simply renamed to *file2* with no file movement between sites.

APPENDIX D

DFS Internal network messages:

This appendix contains a description of the internal messages which make up the DFS network protocols. Each of these command tokens are received at the logical site destination, processed and a response message sent back. Along with a brief description of each message is an indication of the information sent in the message and the characteristics of the message.

OPEN_CMD (US -> CSS): Open a logical file. The file may be opened for reading, writing or both. If the US has the file cached, it includes the last modification time in its message to the CSS. if the US needs a particular SS to be opened (explicit mode), it so indicates. If the open succeeded, the US is returned the site vector which is a list of open file sites.

CREATE_CMD (US -> CSS): Create a logical file. Files are created for reading and writing at two randomly located sites. Files are created with a '.tmp' extension indicating that they are temporary files. Writes to the files are committed on a USCLOSE_CMD command. If the create succeeded, the site vector it returned to the US.

USCLOSE_CMD (US -> SS): This message is sent to the primary SS. If the file was created, the writes are committed. The SS sends the SSCLOSE_CMD to the CSS and then proceeds to deallocate data structures associated with the open file.

USDBCLOSE_CMD (US ->SS): This message is sent to the primary SS and closes a data base file. Since all writes to the data base are committed only on the USCOMMIT_CMD, this call simply de-allocates data structures at the SS. The SS sends an SSDBCLOSE_CMD command to the CSS.

SSCLOSE_CMD (SS ->CSS): If the file was opened for writing, a copy of the file containing the writes is sent to the CSS. The writes are committed only after successfully sending the file to the CSS and receiving the CSSCLOSE_CMD from it.

SSDBCLOSE_CMD (SS -> CSS): This is sent if the file is a data base file. Data structures are deallocated at the primary SS.

CSSCLOSE_CMD (CSS -> SS): This message is sent to all the open images of the file including the primary image. A copy of the file containing the writes is sent to all secondary images. All images commit the writes to the file.

CSSDBCLOSE_CMD (CSS -> SS): This message is sent to all the open images of the data base file except the primary image. All secondary images deallocate data structures associated with the open file.

READ_CMD (US -> SS): Read the logical file image located at the SS. 512 bytes of data are read at a time. The file is cached at the US for subsequent reads.

WRITE_CMD (US -> SS): Write to the logical file. 512 bytes of data are written at a time. If the write is unsuccessful, an alternate site is chosen and the call is restarted.

UNLIN_CMD (US -> CSS): Unlink the named file. All images of the file are unlinked. If any of the image sites are inaccessible at the time of unlinking, a warning message is returned to the US.

LINK_CMD (US -> CSS): A new link (directory entry) is created for the existing file.

MKNOD_CMD (US -> CSS): Create a new directory file at two randomly selected locations unless the user is in explicit mode. Directory files are treated in the same way as regular files.

CHDIR_CMD (US -> CSS): Change directory to that specified by the user. The CSS reads the name server to check whether the named directory is a valid name in the logical name space.

RREAD_CMD (US -> SS): Read a data base file and return a record. The index of the record that the user wishes to read is included in the message and the SS searches the database for record.

NRREAD_CMD (US -> SS): Read the next record from the data base pointed to by the current location of the file pointer.

RWRITE_CMD (US -> SS): Write a record into the data base. Writes are not committed until the USCOMMIT_CMD is issued.

RDELETE_CMD (US -> SS): Delete a record from the data base. The index of the record the user wishes to delete is included in the message.

SHOW_CMD (US -> CSS): This message queries the name server for the named files attributes. Status information is returned to the US.

MKROOT_CMD (US -> CSS): This message is similar to the MKNOD_CMD. It is used to create the root directory. Since the root directory has no parent directory, processing of the call is slightly different from that of a regular directory.

SEEK_CMD (US -> SS); This is sent to an open files image and repositions the read /write pointer to the top of the file.

NSREAD_CMD (CSS -> SS): Read the name server data base and return the record indexed by the named file. Since any changes to this file also entail changes to the parent directory, this command

also returns the record corresponding to the files parent directory. The message is sent to the site that stores the most up to date image of the name server.

NSWRITE_CMD (CSS -> SS): Write a record to the name server data base. All writes are made only to the local image. No attempt is made to contact the other images if the local write fails.

NSUNLINK_CMD (CSS -> SS); Delete a record from the name server data base. Here again the delete is only done to the local image.

DIRW_CMD (CSS -> SS): A new link in the directory file is created. All images of the directory are updated.

DIRD_CMD (CSS -> SS): Delete a link from the directory file. All images of the directory are updated.

DIRCK_CMD (CSS -> SS): Check if the directory is empty. This message is sent when a directory is being unlinked.

BTRN_CMD (US -> CSS): Begin a transaction. At the CSS the transaction is set to RUNNING and the transaction identifier is returned to the US.

USCOMMIT_CMD (US -> CSS): Commit a running transaction. At the CSS an attempt is made to commit the records in its intentions list to the various data base images.

USABORT_CMD (US -> CSS): Abort the transaction and discard all writes made within the transaction. The intentions list is deleted at the CSS.

SSCOMMIT_CMD (CSS -> SS): This message is sent to the SS as a result of the USCOMMIT_CMD command. The CSS sends the record to be committed to the SS and the SS then proceeds to insert the record in the data base.

SSABORT_CMD (CSS -> SS): This message is sent to the SS as a result of the USABORT_CMD command. The SS discards the writes that were made during the transaction.

TSS_WRITE (SS -> CSS): On a transaction write, the SS sends the record to be written to the CSS which then inserts it into its intentions list.

USLOCK_CMD (US -> SS): Lock the named file in the appropriate mode. If the file is already locked in an incompatible mode the request is denied otherwise an SSLOCK_CMD is sent to the CSS.

SSLOCK_CMD (SS -> CSS): A lock request from the SS to lock the remaining images.

CSSLOCK_CMD (CSS -> SS): Request the SS to lock the file. If the file is already locked in an incompatible mode, the lock request is denied.

CSSMKCNS_CMD (CSS -> SS): This command is sent when a file is opened. It attempts to make a file or directory consistent. At the CSS, the version vector for the file is read and checked for inconsistencies. If any exist the image with the highest version number is sent this message and the responding SS sends the file back to the CSS. This message is also sent by the watch dog program when updating the name server.

SSMKCNS_CMD (CSS -> SS): This message is sent to the SS which store older versions of the file. The CSS sends the file to the SS which then overwrites the older version with the newer version.

SSMKNSCNS_CMD (CSS -> SS): This message is similar to the SSMKCNS_CMD except that it is sent when updating the name server and so the version vector associated with the name server itself is also updated.

FLUSH_CMD (US -> CSS): Flush all the data structures at the US, CSS and SS. All open files which have timed out are closed. This message results in the CSSFLUSH_CMD being sent by the CSS.

CSSFLUSH_CMD (CSS -> SS): Close the open file and deallocate data structures.

APPENDIX E

System configuration, installation and administration:

The DFS source code and system file organization is shown in figure 1. A description of each directories contents follows:

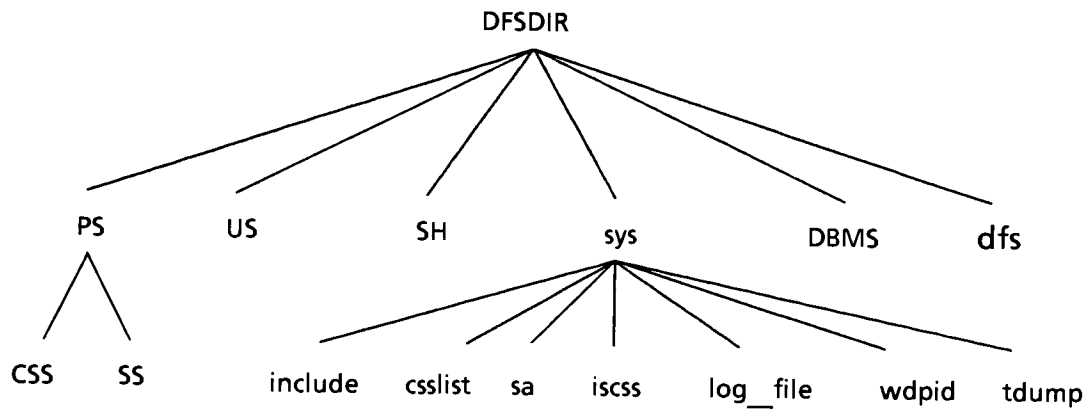


Fig. 1 DFS system file structure

teeldksglakdfglakdfg

DFSDIR / PS: Source code for the process server.

DFSDIR / CSS: Source code for the current synchronization program.

DFSDIR / SS: Source code for the serving site program.

DFSDIR / US: Source code for the using site program. kjh

DFSDIR / SH: Source code for the shell interface program

DFSDIR /DBMS: Source code for the database interface program.

DFSDIR / dfs: Locations for the physical files local to site.

DFSDIR / sys: System directory for the file system.

DFSDIR /sys/include: Contains the C header files.

DFSDIR /sys/csslist: An ascii file that contains a list of potential CSS locations. The system reads this file every time a message needs to be sent to the CSS. The first entry is always the current CSS. When the CSS crashes and the error recovery procedure is run, the first entry is deleted and

repositioned at the end of the file. The next entry now becomes the CSS location. A sample 'csslist' file is shown in figure 2.

locust
mimosa

Fig. 2 Sample csslist file

DFSDIR /sys/sa: This file contains the first and last IP addresses of the machines on the DFS system. It is assumed that all machine addresses between the first and last addresses are included in the DFS system. This file is referenced whenever the system is seeking random locations to create files at. A sample 'sa' file is shown in figure 3.

129.21.40.54
129.21.40.59

Fig. 3. Sample sa file

DFSDIR /sys/iscss: This file contains a code which tells the system whether a given site is the CSS or not. At system boot time, if the local site is a potential CSS location, this code is set to 1. Potential CSS sites thus contain this file while others do not. When a message arrives at a potential CSS location, this code is set to 0 indicating that it is now the CSS. If the CSS crashes and is rebooted, the code is set to -1 indicating to the system that the site has recovered from a crash and is no longer the CSS. All sites that send messages to this site receive an 'ENOTCSS' error message and can thus select a new CSS and run a CSS recovery procedure. Once a CSS has crashed, the only way it can be the CSS again is if the code is set to 1. This needs to be done manually. The system thus works automatically in the face of CSS crashes until all MAXCOPY sites have crashed, at which point human intervention is required.

DFSDIR /sys/log_file: This file contains system generated diagnostic error messages. The message is time stamped for easy analysis.

DFSDIR /sys/wdpid: This file contains the process id of the watch dog program. If the process server is brought down or crashes, the watch dog program is sent a signal and is brought down also.

DFSDIR /sys/tdump: This file holds a dump of a transaction write in the event of site failure so that the watch dog program can attempt to commit it later.

To make a site fully functional, the process server and US programs are run in the background. The shell and database interfaces are invoked by users as needed.

BIBLIOGRAPHY

[ANDR 87]

Andrews, Gregory R., Schtlichting, Richard D., Hayes, Roger, and Purdin, Titus D., *The Design of the Saguaro Distributed Operating System*, IEEE Transactions on Software Engineering, Vol. SE-13, No.1, January 1987.

[AT&T 87]

AT&T UNIX System V Network Programmers Guide, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[BACH 86]

Bach, Maurice J., *The Design of the Unix Operating System*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1986.

[BRER 82]

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

[BROW 82]

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

[FORD 86]

Ford, Edward., *Distributed file systems for Unix*, M.S. Dissertation, Rochester Institute of Technology, School of Computer Science and Technology, 1986.

[OPEN 83]

Open, D.C., and Dalal, Y.K., *The Clearinghouse: A decentralized agent for locating named objects in a distributed environment*, ACM Transactions on Office Information Systems, Vol. 1, July, 1983.

[LAMP 85]

Lampson, B.W., *Atomic Transactions*, Distributed Systems, Architecture and Implementation, Edited by Lampson, B.W et al,. Springer-Verlag, Berlin, 1985.

[LELA 85]

LeLann, Gerald., *Motivations, objectives and characterization of distributed systems*, Distributed Systems, Architecture and Implementation, Edited by Lampson, B.W et al,. Springer-Verlag, Berlin, 1985.

[LEVI 87]

Levine, Paul H., *The Apollo DOMAIN Distributed File System*, Distributed Operating Systems. Theory and Practice, Edited by Paker, Y., et al., Springer-Verlag, Berlin, 1985.

[LUD 81]

Luderer, G.W.R., Che, H., Haggerty, J.P., Kirslis, P.A., and Marshall, W.T., *A Distributed Unix System based on a Virtual Circuit Switch,* Proc. of the Eighth Symposium on Operating System Principles, Dec 14-16, 1981, Asilomar, Ca.

[NELS 84]

Nelson, D.L., and Leach, P.J., *The Architecture and Applications of the Apollo Domain*, IEEE Computer Graphics and Applications, April 1984.

[POPE 81]

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

[POPE 85]

Popek, G., and Walker, B., *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[POST 81a]

Postel, J., (ed) , *Transmission Control Protocol DARPA Internet Program Protocol Specification*, RFC 793, USC/Information Sciences Institute, September, 1981.

[POST 81b]

Postel, J., (ed) , *Internet Protocol DARPA Internet Program Protocol Specification*, RFC 791, USC/Information Sciences Institute, September, 1981.

[ROCH 85]

Rochkind, Mark J., *Advanced Unix Programming*, Advanced Programming Institute Ltd., Boulder, Colorado, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1985.

[SAND 85]

Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., Design and Implementation of the SUN Network File System, Usenix Proceedings, Summer 1985.

[SATY 85]

Satyanarayan, M., Howard, John, H., Nichols, David, A., Sidebotham, Robert N., Spector, Alfred Z., and West, Michael J., *The ITC Distributed File System: Principles and Design*,

[STAN 87]

Stanly, Allen., *Simulation of a distributed file system,* M.S. Dissertation, Rochester Institute of Technology, School of Computer Science and Technology, 1987.

[STUR 80]

Sturgis, H.E., Mitchell, J., and Israel, J., *Issues in the Design and Use of a Distributed File System,* Operating System Review, July 1980.

[TANE 81]

Tanenbaum, Andrew S., *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1981.

[WALK 83a]

Walker, B., Popek, G., English, R., Kline, C., and Thiel, G., *The LOCUS Distributed Operating System,* Proceedings of the Ninth Symposium on Operating System Principles, Bretton Wood, New Hampshire, October 1983.

[WALK 83b]

Walker, Bruce James, *Issues of Network Transparency and File Replication in the Distributed File System Component of LOCUS*, Ph.D Dissertation, UCLA Computer Science 1983.

[WALS 85]

Walsh, D., *An Overview of the SUN network file system,* Usenix Proceedings, Dallas, 1985.

[WATS 85]

Watson, R.W,. *Identifiers (naming) in Distributed Systems* Distributed Systems, Architecture and Implementation, Edited by Lampson, B.W et al,. Springer-Verlag, Berlin, 1985.

[WEIN 86]

Weinstein, M.J., Page, T.W.Jr., Livezey, B.K., and Popek, G.J., *Transactions and Synchronization in a Distributed Operating System*, Operating System Review, Vol. 20, No. 1, January 1985.