The LOCUS Distributed Operating System[1]

Bruce Walker, Gerald Popek, Robert English, Charles Kline and Greg Thiel[2]

University of California at Los Angeles

Abstract

LOCUS is a distributed operating system which supports transparent access to data through a network wide filesystem, permits automatic replication of storage, supports transparent distributed process execution, supplies a number of high reliability functions such as nested transactions, and is upward compatible with Unix. Partitioned operation of subnets and their dynamic merge is also supported.

The system has been operational for about two years at UCLA and extensive experience in its use has been obtained. The complete system architecture is outlined in this paper, and that experience is summarized.

1 Introduction

LOCUS is a Unix compatible, distributed operating system in operational use at UCLA on a set of 17 Vax/750's connected by a standard Ethernet[3]. The system supports a very high degree of *network transparency,* i.e. it makes the network of machines appear to users and programs as a single computer; machine boundaries are completely hidden during normal operation. Both files and programs can be moved dynamically with no effect on naming or correct operation. Remote resources are accessed in the same manner as local ones. Processes can be created locally and remotely in the same manner, and process interaction is the same, independent of location. Many of these functions operate transparently even across heterogeneous cpus.

LOCUS also provides a number of high reliability facilities, including flexible and automatic replication of storage at a file level, a full implementation of nested transactions[MEUL 83], and a substantially more robust data storage facility than conventional Unix systems. All of the functions reported here have been implemented, and most are in routine use.

This paper provides an overview of the basic LOCUS system architecture. The file system, especially its distributed naming catalog, plays a central role in the system structure, both because file system activity typically predominates in most operating systems and so high performance is critical, and because the generalized name service provided is used by so many other parts of the system. Therefore, the file system is described first. Remote processes are discussed next, including discussions of process creation, inter-process functions and error handling.

An important part of the LOCUS research concerns recovery from failures of parts of the system, including partition of a LOCUS system into separated but functioning subnetworks. The next sections of this paper discuss the several LOCUS facilities dedicated to recovery. First is the merging of the naming catalog; the hierarchical directory system. The handling of other object types in the file system is also briefly considered. These recovery algorithms

[1] This research was supported by the Advanced Research Projects Agency under research contract DSS-MDA-903-82-C-0189.

[2] Current address of Bruce Walker, Charles Kline and Greg Thiel: LOCUS Computing Corporation, 3330 Ocean Park Blvd., Santa Monica, Ca. 90405.

[3] Initial work was done on DEC PDP-11/45's and VAX 750's using both 1 and 10 megabit ring networks.

are designed to permit normal operation while resources are arriving and departing. Last, the protocols which LOCUS sites execute in order to maintain and define the accessible members of a network, i.e. the network topology, are discussed. These protocols are designed to assure that all sites converge on the same answer in a rapid manner.

The paper concludes with a set of observations regarding the use of LOCUS in production settings, especially the value of its network transparent interface.

2 Distributed Filesystem

2.1 Filesystem overview

The LOCUS filesystem presents a single tree structured naming hierarchy to users and applications. It is functionally a superset of the Unix tree structured naming system. There are three major areas of extension. First, the single tree structure in LOCUS covers all objects in the filesystem on all machines. LOCUS names are fully transparent; it is not possible from the name of a resource to discern its location in the network. Such location transparency is critical for allowing data and programs in general to move or even be executed from different sites. The second direction of extension concerns replication. Files in LOCUS can be replicated to varying degrees, and it is the LOCUS system's responsibility to keep all copies up to date, assure that access requests are served by the most recent available version, and support partitioned operation.

To a first approximation, the pathname tree is made up of a collection of filegroups, as in a conventional Unix environment[1]. Each group is a wholly self contained subtree of the naming hierarchy, including storage for all files and directories contained in the subtree. Gluing together a collection of filegroups to construct the uniform naming tree is done via the *mount* mechanism. Logically mounting a filegroup attaches one tree (the filegroup being mounted) as a subtree within an already mounted tree. The glue which allows smooth path traversals up and down the expanded naming tree is kept as operating system state information. Currently this state information is replicated at all sites. To scale a LOCUS network to hundreds or thousands of sites, this "mount" information would be cached.

[1] The term filegroup in this paper corresponds directly to the Unix term filesystem.

A substantial amount of the LOCUS filesystem design, as well as implementation, has been devoted to appropriate forms of error and failure management. These issues will be discussed throughout this paper. Further, high performance has always been a critical goal. In our view, solutions to all the other problems being addressed are really not solutions at all unless their performance is suitable. In LOCUS, when resources are local, access is no more expensive than on a conventional Unix system. When resources are remote, access cost is higher, but dramatically better than traditional layered file transfer and remote terminal protocols permit. Measured performance results are presented in [GOLD 83].

2.2 File Replication

2.2.1 Motivation for Replication

Replication of storage in a distributed filesystem serves multiple purposes. First, from the users' point of view, multiple copies of data resources provide the opportunity for substantially increased availability. This improvement is clearly the case for read access, although the situation is more complex when update is desired, since if some of the copies are not accessible at a given instant, potential inconsistency problems may preclude update, thereby decreasing availability as the level of replication is increased.

The second advantage, from the user viewpoint, concerns performance. If users of the file exist on different machines, and copies are available near those machines, then read access can be substantially faster compared to the necessity to have one of the users always make remote accesses. This difference can be substantial; in a slow network, it is overwhelming, but in a high speed local network it is still significant[1].

In a general purpose distributed computing environment, such as LOCUS, some degree of replication is essential in order for the user to be able to work at all. Certain files used to set up the user's environment must be available even when various machines have failed or are inaccessible. The start-up files in Multics, or the various Unix shells, are ob-

[1] In the LOCUS system, which is highly optimized for remote access, the cpu overhead of accessing a remote page is twice local access, and the cost of a remote open is significantly more than the case when the entire open can be done locally.

vious examples. Mail aliases and routing information are others. Of course, these cases can generally be handled by read-only replication, which in general imposes fewer problems[1].

From the system point of view, some form of replication is more than convenient; it is absolutely essential for system data structures, both for availability and performance. Consider a file directory. A hierarchical name space in a distributed environment implies that some directories will have entries which refer to files on different machines. There is strong motivation for storing a copy of all the directory entries in the backward path from a file at the site where the file is stored, or at least "nearby". The principal reason is availability. If a directory entry in the naming path to a file is not accessible because of network partition or site failure, then that file *cannot be accessed,* even though it may be stored locally. LOCUS supports replication at the granularity of the entire directory (as opposed to the entry granularity) to address this issue.

Second, directories in general experience a high level of read access compared to update. As noted earlier, this characteristic is precisely the one for which a high degree of replicated storage will improve system performance. In the case of the file directory hierarchy, this improvement is critical. In fact, the access characteristics in a hierarchical directory system are, fortunately, even better behaved than just indicated. Typically, the top of the hierarchy exhibits a very high level of lookup, and a correspondingly low rate of update. This pattern occurs because the root of the tree is heavily used by most programs and users as the starting point for name resolution. Changes disrupt programs with embedded names, and so are discouraged. The pattern permits (and requires) the root directories to be highly replicated, thus improving availability and performance simultaneously. By contrast, as one moves down the tree toward the leaves, the degree of shared use of any given directory tends to diminish, since directories are used to organize the name space into more autonomous subspaces. The desired level of replication for availability purposes tends to decrease as well. Further, the update traffic to directories near the leaves of the naming tree tends to be greater, so one would have less directory replication to improve performance.

The performance tradeoffs between update/read rates and degree of replication are well known, and we have already discussed them. However, there are other costs as well. For example, concurrency control becomes more expensive. Without replication the storage site can provide concurrency control for the object since it will know about all activity. With replication some more complex algorithm must be supported. In a similar way, with replication, a choice must be made as to which copy of an object will supply service when there is activity on the object. This degree of freedom is not available without replication. If objects move, then, in the no replication case, the mapping mechanism must be more general. With replication a move of an object is equivalent to an add followed by a delete of an object copy.

2.2.2 Mechanism Supporting Replication

File replication is made possible in LOCUS by having multiple physical containers for a logical filegroup. A given file belonging to logical filegroup X may be stored at any subset of the sites where there exist physical containers corresponding to X. Thus the entire logical filegroup is not replicated by each physical container as in a "hot shadow" type environment. Instead, to permit substantially increased flexibility, any physical container is incomplete; it stores only a subset of the files in the subtree to which it corresponds.

To simplify access and provide a basis for low level communication about files, the various copies of a file are assigned the same file descriptor or inode number within the logical filegroup. Thus a file's globally unique low-level name is:
<logical filegroup number, file descriptor (inode) number>
and it is this name which most of the operating system uses.

In the case where not all sites are communicating and even for a short time while they are communicating right after a file update, not all the copies of the file are necessarily up to date. To record this and to ensure that the latest copies will be used for any accesses, each copy has a version vector associated with it that maintains necessary history information. See [PARK83].

[1] The problems which remain are present because few files are strictly read-only; it is just that their update rate is low. When an update is done, some way to make sure that all copies are consistent is needed. If the rate is low enough, manual methods may suffice.

2.3 Accessing the Filesystem

There were several goals directing the design of the network-wide file access mechanism. The first was that the system call interface should be uniform, independent of file location. In other words, the same system call with the same parameters should be able to access a file whether the file is stored locally or not. Achieving this goal of transparency would allow programs to move from machine to machine and allow data to be relocated.

The primary system calls dealing with the filesystem are *open, create, read, write, commit, close* and *unlink*. After introducing the three logical sites involved in file access and the file access synchronization aspect of LOCUS, these system calls are considered in the context of the logical tasks of file reading, modifying, creating and deleting.

2.3.1 LOCUS Logical Sites for Filesystem Activities

LOCUS is designed so that every site can be a full function node. As we saw above, however, filesystem operations can involve more than one host. In fact there are three logical functions in a file access and thus three logical sites. These are:

a. *using site*, (US), which issues the request to open a file and to which pages of the file are to be supplied,
b. *storage site*, (SS), which is the site at which a copy of the requested file is stored, and which has been selected to supply pages of that file to the using site,
c. *current synchronization site*, (CSS), which enforces a global access synchronization policy for the file's filegroup and selects SSs for each open request. A given physical site can be the CSS for any number of filegroups but there is only one CSS for any given filegroup in any set of communicating sites (i.e. a partition). The CSS need not store any particular file in the filegroup but in order for it to make appropriate access decisions it must have knowledge of which sites store the file and what the most current version of the file is.

Since there are three possible independent roles a given site can play (US, CSS, SS), it can therefore operate in one of eight modes. LOCUS handles each combination, optimizing some for performance.

Since all open requests for a file go through the CSS function, it is possible to implement a large variety of synchronization policies. In LOCUS, so long as there is a copy of the desired resource available, it can be used. If there are multiple copies present, the most efficient one to access is selected. Other copies are updated in background, but the system remains responsible for supplying a mutually consistent view to the user. Within a set of communicating sites, synchronization facilities and update propagation mechanisms assure consistency of copies, as well as guaranteeing that the latest version of a file is the only one that is visible.

Since it is important to allow modification of a file even when all copies are not currently accessible, LOCUS contains a file reconciliation mechanism as part of the recovery system (described in section 4).

2.3.2 Strategy for Distributed Operation

LOCUS is a procedure based operating system - processes request system service by executing system calls, which trap to the kernel. The kernel runs as an extension to the process and can sleep on behalf of the process. In general, application programs and users cannot determine if any given system call will require foreign service. In fact, most of the high level parts of the system service routines are unaware of the network. At the point within the execution of the system call that foreign service is needed, the operating system packages up a message and sends it to the relevant foreign site. Typically the kernel then sleeps, waiting for a response, much as it would after requesting a disk i/o to be performed on behalf of a specific process.

This flow of control structure is a special case of remote procedure calls. Operating system procedures are executed at a remote site as part of the service of a local system call. Figure 1 traces, over time, the processing done at the requesting and serving site when one executes a system call requiring foreign service.

2.3.3 Reading Files

To read a file, an application or system supplied program issues the *open* system call with a filename parameter and flags indicating that the open is for read. As in standard Unix, pathname searching (or directory interrogation) is done within the operating

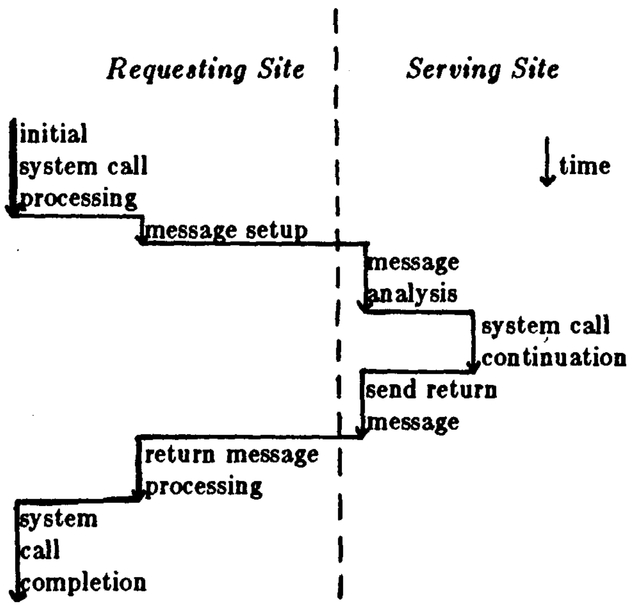

Figure 1: Processing a System Call Requiring Foreign Service

system open call.[1] After the last directory has been interrogated, the operating system on the requesting site has a <logical filegroup number, inode number> pair for the target file that is about to be opened. If the inode information is not already in an incore inode structure, a structure is allocated. If the file is stored locally, the local disk inode information is filled in. Otherwise very little information is initially entered.

Next, the CSS is interrogated. If the local site is the CSS, only a procedure call is needed. If not, the CSS is determined by examining the logical mount table, a message is sent to the CSS, the CSS sets up an incore inode for itself, calls the same procedure that would have been called if the US is the CSS, packages the response, and sends it back to the US. The CSS is involved for several reasons. One is to enforce synchronization controls. Enough state information is kept incore at the CSS to support those synchronization decisions. For example, if the policy allows only a single open for modification, the site where that modification is ongoing would be kept incore at the CSS. Another reason for contacting the CSS is to determine a storage site. The CSS stores a copy of the disk inode information whether or not it actually stores the file. Consequently it has a list of packs which store the file. Using that information and mount table information the CSS can select potential storage sites. The potential sites are polled to see if they will act as storage sites.

Besides knowing the packs where the file is stored, the CSS is also responsible for knowing the latest version vector. This information is passed to potential storage sites so they can check it against the version they store. If they do not yet store the latest version, they refuse to act as a storage site.

Two obvious optimizations are done. First, in it's message to the CSS, the US includes the version vector of the copy of the file it stores, if it stores the file. If that is the latest version, the CSS selects the US as the SS and just responds appropriately to the US. Another simplying case is when the CSS stores the latest version and the US doesn't. In this case the CSS picks itself as SS (without any message overhead) and returns this information to the US.

The response from the CSS is used to complete the incore inode information at the US. For example, if the US is not the SS then all the disk inode information (eg. file size, ownership, permissions) is obtained from the CSS response. The CSS in turn had obtained that information from the SS. The most general open protocol (all logical functions on different physical sites) is:

US --> CSS	OPEN request
CSS --> SS	request for storage site
SS --> CSS	response to previous message
CSS --> US	response to first message.

Figure 2 displays this generally message sequence.

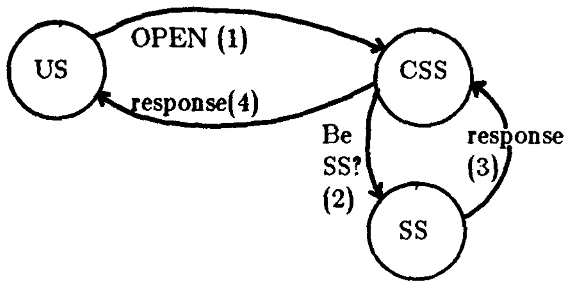

Figure 2: Open Protocol

[1] Pathname searching is described in the next section.

After the file is open, the user level process issues read calls. All such requests are serviced via kernel buffers, both in standard Unix and in LOCUS. In the local case, data is paged from external storage devices into operating system buffers and then copied from there into the address space of the process. Access to locally stored files is the same in LOCUS as in Unix, including the one page readahead done for files being read sequentially.

Requests for data from remote sites operates similarly. Instead of allocating a buffer and queueing a request for a page from a local disk, however, the operating system at the US allocates a buffer and queues a request to be sent over the network to the SS. The request is simple. It contains the <logical filegroup, inode number> pair, the logical page number within the file and a guess as to where the incore inode information is stored at the SS. The CSS is out of the i/o communication.

At the SS, the request is treated, within the operating system, as follows:

a. The incore inode is found using the guess provided;
b. The logical page number is translated into a physical disk block number;
c. A standard low level operating system routine is called to allocate a buffer and get the appropriate page from disk (if it is not already in a buffer);
d. The buffer is queued on the network i/o queue for transmission back to the US as a response to a read request.

The protocol for a network read is thus:[1]

US --> SS	request for page x of file y
SS --> US	response to the above request

As in the case of local disk reads, readahead is useful in the case of sequential behavior, both at the SS, as well as across the network.

One of several actions can take place when the *close* system call is invoked on a remotely stored file, depending on how many times the file is concurrently open at this US.

[1] There are *no* other messages involved; no acknowledgements, flow control or any other underlying mechanism. This specialized protocol is an important contributor to LOCUS performance, but it implies the need for careful higher level error handling.

If this is not the last close of the file at this US, only local state information need be updated in most cases. However, if this is the last close of the file, the SS and CSS must be informed so they can deallocate incore inode structures and so the CSS can alter state data which might affect it's next synchronization policy decision. The protocol is[1]:

US --> SS	US close
SS --> CSS	SS close
CSS --> SS	response to above
SS --> US	response to first message

Closes of course can happen as a result of error conditions like hosts crashing or network partitions. To properly effect closes at various logical sites, certain state information must be kept in the incore inode. The US of course must know where the SS is (but then it needed that knowledge just to read pages). The CSS must know all the sites currently serving as storage sites so if a certain site crashes, the CSS can determine if a given incore inode slot is thus no longer in use. Analogously, the SS must keep track, for each file, of all the USs that it is currently serving.

The protocols discussed here are the lowest level protocols in the system, except for some retransmission support. Because multilayered support and error handling, such as suggested by the ISO standard, is not present, much higher performance has been achieved.

2.3.4 Pathname Searching

In the previous section we outlined the protocol for opening a file given the <logical filegroup number, inode number> pair. In this section we describe how that pair is found, given a character string name.

All pathnames presented to the operating system start from one of two places, either the root (/) or the current working directory of the process presenting the pathname. In both cases an inode is incore at the US for the directory. To commence the pathname searching, the <logical filegroup, inode number> of the starting directory is extracted from

[1] The original protocol for close was simply:

US --> SS	US close of file y
SS --> US	SS close of file y

However, we encountered a race condition under this scheme. The US could attempt to reopen the file before the CSS knew that the file was closed. Thus the responses were added.

the appropriate inode and an internal open is done on it. This is the same internal open that was described at the start of the previous section, but with one difference. A directory opened for pathname searching is not open for normal READ but instead for an internal unsynchronized read. The distinction is that no global locking is done. If the directory is stored locally and there are no propagations pending to come in, the local directory is searched without informing the CSS. If the directory is not local, the protocol involving the CSS must be used but the locking is such that updates to the directory can occur while interrogation is ongoing. Since no system call does more than just enter, delete, or change an entry within a directory and since each of these actions are atomic, directory interrogation never sees an inconsistent picture.

Having opened the initial directory, protection checks are made and the directory is searched for the first pathname component. Searching of course will require reading pages of the directory, and if the directory is not stored locally these pages are read across the net in the same manner as other file data pages. If a match is found, the inode number of that component is read from the the directory to continue the pathname search. The initial directory is closed (again internally), and the next component is opened. This strategy is continued up to the last component, which is opened in the manner requested by the original system call. Another strategy for pathname searching is to ship partial pathnames to foreign sites so they can do the expansion locally, avoiding remote directory opens and network transmission of directory pages. Such a solution is being investigated but is more complex in the general case because the SS for each intermediate directory could be different.

Some special care is necessary for crossing filegroup boundaries, as discussed earlier, and for creating and deleting files, as discussed later.

2.3.5 File Modification

Opening an existing file for modification is much the same as opening for read. The synchronization check at the CSS is different and the state information kept at all three logical sites is slightly different.

The act of modifying data takes on two forms. If the modification does not include the entire page, the old page is read from the SS using the read protocol. If the change involves an entire page, a buffer is set up at the US without any reads. In either case, after changes are made, the page is sent to the SS via the write protocol, which is simply[1]:

US -> SS   Write logical page x in file y

The action to be taken at the SS is described in the next section in the context of the commit mechanism.

The close protocol for modification is similar to the read case. However, at the US all modified pages must be flushed to the SS before the close message is sent. Also, the mechanism at the SS interacts with the commit mechanism, so we turn to it now.

2.3.6 File Commit

The important concept of atomically committing changes has been imported from the database world and integrated into LOCUS. All changes to a given file are handled atomically. Such a commit mechanism is useful both for database work and, in general, and can be integrated without performance degradation. No changes to a file are permanent until a commit operation is performed. *Commit* and *abort* (undo any changes back to the previous commit point) system calls are provided, and closing a file commits it.

To allow file modifications to act like a transaction, it is necessary to keep both the original and changed data available. There are two well known mechanisms to do so: a) logging and b) shadow pages or intentions lists [LAMP 81a]. LOCUS uses a shadow page mechanism, partly because Unix file modifications tend to overwrite entire files, and partly because high performance shadowing is easier to implement.

The US function never deals with actual disk blocks but rather with logical pages. Thus the entire shadow page mechanism is implemented at the SS and is transparent to the US. At the SS, then, a new physical page is allocated if a change is made to an existing page of a file. This is done without any extra i/o in one of two ways: if an entire page is being changed, the new page is filled in with the new data and written to the storage medium; if the change is not of the entire page, the old page is read, the name of the buffer is changed to the new page, the changed data is entered and this new page is written to the storage medium. Both these cases leave the

[1] There are low level acknowledgements on this message to ensure that it is received. No higher level response is necessary.

old information intact. Of course it is necessary to keep track of where the old and new pages are. The disk inode contains the old page numbers. The incore copy of the disk inode starts with the old pages but is updated with new page numbers as shadow pages are allocated. If a given logical page is modified multiple times it is not necessary to allocate different pages. After the first time the page is modified, it is marked as being a shadow page and reused in place for subsequent changes.

The atomic commit operation consists merely of moving the incore inode information to the disk inode. After that point, the file permanently contains the new information. To abort a set of changes rather than commit them, one merely discards the incore information since the old inode and pages are still on disk, and free up page frames on disk containing modified pages. Additional mechanism is also present to support large files that are structured through indirect pages that contain page pointers.

As is evident by the mechanism above, we have chosen to deal with file modification by first committing the change to one copy of a file. Via the centralized synchronization mechanism, changes to two different copies at the same time is blocked, and reading an old copy while another copy is being modified is prevented.[1] As part of the commit operation, the SS sends messages to all the other SS's of that file as well as the CSS. At a minimum, these messages identify the file and contain the new version vector. Additionally, for performance reasons, the message can indicate: a) whether it was just inode information that changed and no data (eg. ownership or permissions) or b) which explicit logical pages were modified. At this point it is the responsibility of these additional SS's to bring their version of the file up to date by propagating in the entire file or just the changes. A queue of propagation requests is kept by the kernel at each site and a kernel process services the queue.

Propagation is done by "pulling" the data rather than "pushing" it. The propagation process which wants to page over changes to a file first internally opens the file at a site which has the latest version. It then issues standard read messages either for all the pages or just the modified ones. When each page arrives, the buffer that contains it is renamed and sent out to secondary storage, thus avoiding copying data into and out of an application data space, as would be necessary if this propagation mechanism were to run as an application level process. Note also that this propagation-in procedure uses the standard commit mechanism, so if contact is lost with the site containing the newer version, the local site is still left with a coherent, complete copy of the file, albeit still out of date.

Given this commit mechanism, one is always left with either the original file or a completely changed file but never with a partially made change, even in the face of local or foreign site failures. Such was not the case in the standard Unix environment.

2.3.7 File Creation and Deletion

The system and user interface for file creation and deletion is just the standard Unix interface, to retain upward compatibility and to maintain transparency. However, due to the potential for replicated storage of a new file, the create call needs two additional pieces of information - how many copies to store and where to store them. Adding such information to the create call would change the system interface so instead defaults and per process state information is used, with system calls to modify them.

For each process, an inherited variable has been added to LOCUS to store the default number of copies of files created by that process. A new system call has been added to modify and interrogate this number. Currently the initial replication factor of a file is the minimum of the user settable number-of-copies variable and the replication factor of the parent directory.

Initial storage sites for a file are currently determined by the following algorithm:

a. All such storage sites must be a storage site for the parent directory;
b. The local site is used first if possible;
c. Then follow the site selection for the parent directory, except that sites which are currently inaccessible are chosen last.

This algorithm is localized in the code and may change as experience with replicated files grows.

As with all file modification, the create is done at one storage site and propagated to the other storage sites. If the storage site of the created file is not local, the protocol for the create is very similar

[1] Simultaneous read and modification requests, even when initiated at different sites is allowed. Page-valid tokens are managed by the kernels for this purpose. Only one storage site can be involved, unlike the case when there are only multiple readers.

to the remote open protocol, the difference being that a placeholder is sent instead of an inode number. The storage site allocates an inode number from a pool which is local to that physical container of the filegroup. That is, to facilitate inode allocation and allow operation when not all sites are accessible, the entire inode space of a filegroup is partitioned so that each physical container for the filegroup has a collection of inode numbers that it can allocate.

File delete uses much of the same mechanism as normal file update. After the file is open for modification, the US marks the inode and does a commit, which ships the inode back to the SS and increments the version vector. As part of the commit mechanism, pages are released and other sites are informed that a new version of the file exists. As those sites discover that the new version is a delete, they also release their pages. When all the storage sites have seen the delete, the inode can be reallocated by the site which has control of that inode (i.e. the storage site of the original create).

2.4 Other Issues

The LOCUS name service implemented by the directory system is also used to support interprocess communication and remote device access, as well as to aid in handling heterogeneous machine types in a transparent manner. We turn to these issues now.

2.4.1 Site and Machine Dependent Files

There are several aspects to the hardware heterogeneity problem, such as number representation, byte ordering, instruction set differences and special hardware constraints (eg. floating point availability). Strong file typing and conversions during transmission can help some of these problems[1]. Here we address only the file naming problem.

While globally unique user visible file naming is very important most of the time, there can be situations where an uttered filename wants to be interpreted specially, based on the context under which it was issued. The machine-type context is a good example. In a LOCUS net containing both DEC PDP-11/45s and DEC VAX 750s, a user would want to type the same command name on either type of machine and get a similar service. However, the load modules of the programs providing that service could not be identical and would thus have to have different globally unique names. To get the proper load modules executed when the user types a command, then, requires using the context of which machine the user is executing on. A discussion of transparency and the context issue is given in [POPE 83a]. Here we outline a mechanism implemented in LOCUS which allows context sensitive files to be named and accessed transparently.

[1] Solutions to the number representation and byte ordering problems have not yet been implemented.

Basically the scheme consists of four parts:

a. Make the globally unique name of the object in question refer to a special kind of directory (hereafter referred to as a *hidden directory)* instead of the object itself.
b. Inside this directory put the different versions of the file, naming them based on the context with which they are associated. For example, have the command */bin/who* be a hidden directory with the file entries *45* and *vax* that are the respective load modules.
c. Keep a per-process inherited context for these hidden directories. If a hidden directory is found during pathname searching (see section 4.4 for pathname searching), it is examined for a match with the process's context rather than the next component of the pathnames passed to the system.
d. Give users and programs an escape mechanism to make hidden directories visible so they can be examined and specific entries manipulated.

As we shall see in section 3, not only does this naming scheme allow us to store and name load modules for different sites, but allows us to transparently run a requested command on the site for which a load module exists.

2.4.2 Other Filesystem Objects

In LOCUS, as in Unix, the name catalog also includes objects other than files; devices and interprocess communication (ipc) channels are the best known.

LOCUS provides for transparent use of remote devices in most cases[1]. This functionality is exceedingly valuable, but involves considerable care. The

[1] The only exception is remote access to raw, non-character devices and these can be accessed by executing processes remotely.

implementation architecture is beyond the scope of this paper.

Interprocess communication (ipc) is often a controversial subject in a single machine operating system, with many differing opinions. In a distributed environment, the requirements of error handling impose a number of additional requirements that help make design decisions, potentially easing disagreements.

In LOCUS, the initial ipc effort was further simplified by the desire to provide a network-wide ipc facility which is fully compatible with the single machine functions that were already present in Unix. Therefore, in the current LOCUS system release, Unix *named pipes* and *signals* are supported across the network. Their semantics in LOCUS are identical to those seen on a single machine Unix system, even when processes are resident on different machines in LOCUS. Just providing these seemingly simple ipc facilities was non-trivial, however. Details of the implementation are given in [WALK83].

3 Remote Processes

Transparent support for remote processes requires a facility to create a process on a remote machine, initialize it appropriately, support cross machine, inter-process functions with the same semantics as were available on a single machine, and reflect error conditions across machine boundaries. Each of these is discussed below.

3.1 Remote Process Creation

LOCUS permits one to execute programs at any site in the network, subject to permission control, in a manner just as easy as executing the program locally. In fact, one can dynamically, even just before process invocation, select the execution site. No rebinding or any other action is needed. The mechanism is entirely transparent, so that existing software can be executed either locally or remotely, with no change to that software.

The decision about where the new process is to execute is specified by information associated with the calling process. That information, currently a structured advice list, can be set dynamically. Shell commands to control execution site are also available.

Processes are typically created by the standard Unix *fork* call. Both fork and *exec*, the Unix call which installs a load module into a process and starts execution, are controlled by site information in the process environment. If exec is to occur remotely, then the process is effectively moved at that time. By doing so it is feasible to support remote execution of programs intended for dissimilar cpu types.

In both cases, a process body is allocated at the destination site following a message exchange between the calling site and the new process site. More significant, it is necessary to initialize the new process' environment correctly. This requires, for Unix compatibility, that the parent and child process share open file descriptors (which contain current file position pointers[1]), a copy of other process state information.

In the case of a fork, the process address space, both code and data, must be made a copy of the parents'. If the code is reentrant, and a copy already exists on the destination machine, it should be used. In any case, the relevant set of process pages are sent to the new process site.

For optimization purposes, a *run* call has been added that is similar to the effect of a fork followed by a exec. If the run is to execute remotely, the effect is a local fork and a remote exec. However, run is transparent as to where is executes. Run avoids the copy of the parent process image which occurs with fork, and includes parameterization that permits the caller to set up the environment of the new process, be local or remote.

3.2 Inter-process Functions

The semantics of the available functions by which processes interact determines, to a large extent, the difficulty involved in supporting a transparent process facility. In Unix, there are explicit functions such as signals and pipes (named or not), but there are also implicit mechanisms; shared open files are the most significant. The most difficult part of these functions' semantics is their expectation of shared memory. For example, if one process sharing an open file reads or writes a character, and then another does so, the second process receives or alters

[1] To implement this functionality across the network we keep a file descriptor at each site, with only one valid at any time, using a token scheme to determine which file descriptor is currently valid.

the character following the one touched by the first process. Pipes have similar characteristics under certain circumstances.

All of these mechanisms are supported in LOCUS, in part through a token mechanism which marks which copy of a resource is valid; access to a resource requires the token. This concept is used at various levels within the system. While in the worst case, performance is limited by the speed at which the tokens and their associated resources can be flipped back and forth among processes on different machines, such extreme behavior is exceedingly rare. Virtually all processes read and write substantial amounts of data per system call. As a result, most collections of Unix processes designed to execute on a single machine run very well when distributed on LOCUS.

3.3 Error Handling

In LOCUS, process errors are folded into the existing Unix interface to the degree possible. The new error types primarily concern cases where either the calling or called machine fails while the parent and child are still alive. When the child's machine fails, the parent receives an error signal. Additional information about the nature of the error is deposited in the parent's process structure, which can be interrogated via a new system call. When the parent's machine fails, the child is notified in a similar manner. Otherwise, the process interface in LOCUS is the same as in Unix.

4 LOCUS Recovery Philosophy

The basic approach in LOCUS is to maintain, within a single partition, strict synchronization among copies of a file so that all uses of that file see the most recent version, even if concurrent activity is taking place on different machines. Each partition operates independently, however. Upon merge, conflicts are reliably detected. For those data types which the system understands, automatic reconciliation is done. Otherwise, the problem is reported to a higher level; a database manager for example, who may itself be able to reconcile the inconsistencies. Eventually, if necessary, the user is notified and tools are provided by which he can interactively merge the copies.

An important example where replicated operation is needed, in a distributed system, is the *name service,* the mechanism by which the user sensible names are translated into internal system names and locations for the associated resource. Those mapping tables must themselves be replicated, as already pointed out. A significant part of the basic replication mechanism in LOCUS is used by its name service, or directory system, and so we will concentrate on that part of recovery in the remainder of our discussion.

4.1 Partitions

Partitions clearly are the primary source of difficulty in a replicated environment. Some authors have proposed that the problem can be avoided by having high enough connectivity that failures will not result in partitions. In practice, however, there are numerous ways that effective partitioning occurs. In single local area networks, a single loose cable terminator can place all machines in individual partitions of a single node. Gateways between local nets fail. Long haul connections suffer many error modes. Even when the hardware level is functioning, there are miriad ways that software levels cause messages not to be communicated; buffer lockups, synchronization errors, etc. Any distributed system architectural strategy which depends for its correct and convenient operation on the collection of these failure modes being exceedingly infrequent is a fragile model, in our judgment. In addition, there are maintenance and hardware failure scenarios that can result in file modification conflict even when two sites have never executed independently at the same time. For example, while site B is down, work is done on site A. Site A goes down before B comes up. When site A comes back up, an effective partition merge must be done.

Given partitioning will occur, and assuming replication of data is desired for availability, reliability, and performance, an immediate question is whether a data object, appearing in more than one partition, can be updated during partition. In our judgment, the answer must be yes. There are numerous reasons. First, if it is not possible, then availability goes down, rather than up, as the degree of replication increases. Secondly, the system itself must maintain replicated data, and permit update during partitioned mode. Directories are the obvious example. Solutions to that problem may well be made available to users at large. Third, in many environments, the probability of conflicting updates is low. Actual intimate sharing is often not the rule. Thus, unless the user involved needed to get at an alternate copy because of system failures, a policy of allowing update in all partitions will not lead to conflicting updates. To forbid update in all partitions, or all except one, can be a severe constraint, and in most cases will have been unnecessary.

Given the ability to update a replicated object during partition, one must face the problem of mutual consistency of the copies of each data object. Further, the merge procedure must assure that no updates are lost when different copies are merged. Solutions proposed elsewhere, such as primary copy [ALSB76], majority consensus [THOM78], and weighted voting [MENA77] are excluded. They all impose the requirement that update can be done in at most one partition. Even methods such as that used in Grapevine [BIRR82] are not suitable. While Grapevine assures that copies will eventually reach a consistent state, updates can be lost.

It is useful to decompose the replication/merge problem into two cases. In the first, one can assume that multiple copies of a given object may be reconciled independently of any other object. That is, the updates done to the object during partition are viewed as being unrelated and independent of updates (or references) to other objects.

The second case is the one that gives rise to transactions. Here it is recognized that changes to sets of objects are related. Reconciliation of differing versions of an object must be coordinated with other objects and the operations on those objects which occurred during partition.

LOCUS takes both points of view. The basic distributed operating system assumes, as the default, that non-directory file updates and references are unrelated to other non-directory files. The steps which are taken to manage replication under those assumptions are discussed in the next section. In addition, LOCUS provides a full nested transaction facility for those cases where the user wishes to bind a set of events together. Case specific merge strategies have been developed. The recovery and merge implications of these transactions are discussed later.

4.2 Detection of Conflicting Updates to Files

Suppose file f was replicated at sites S_1 and S_2. Initially assume each copy was identical but after some period sites S_1 and S_2 partitioned. If f is modified at S_1 producing f_1 then when S_1 and S_2 merge the two copies of f will be inconsistent. Are they then in conflict? **No.** The copy at S_1 (f_1) should propagate to S_2 and that will produce a consistent state. The copies of the object would be in conflict if during the partition not only was S_1's copy modified to produce f_1 but S_2's copy was modified to produce f_2. At merge a conflict should be detected. As already pointed out the system may be able to resolve the conflict. This is just a simple example. There could be several copies of the object and the history of the modifications and partitions can be complex. Detecting consistency under the general circumstances is non-trivial, but a elegant solution is presented in [PARK83], and is implemented in LOCUS.

For some types of system supported single file structures the system can mechanically resolve those conflicts which are detected. Directories and mailboxes have relatively simple semantics (add and delete are the major operations) and can be done in this manner. These cases are critical to LOCUS, and will be discussed below.

4.3 File System Merge

A distributed file system is an important and basic case of replicated storage. The LOCUS file system is a network wide, tree structured directory system, with leaves being data files whose internal structure is unknown to the LOCUS system nucleus. All files, including directories, have a *type* associated with them. The type information is used by recovery software to take appropriate action. Current types are:

directories
mailboxes (several kinds)
database files
untyped files

The LOCUS recovery and merge philosophy is hierarchically organized. The basic system is responsible for detecting all conflicts. For those data types that it manages, including internal system data as well as file system directories, automatic merge is done by the system. If the system is not responsible for a given file type, it reflects the problem up to a higher level; to a recovery/merge manager if one exists for the given file type. If there is none, the system notifies the owner(s) of the file that a conflict exists, and permits interactive reconciliation of the differences. Transaction recovery and merge can also be supported across partitions in LOCUS. See [FAIS81, THIE83 and MEULL83].

4.4 Reconciliation of a Distributed, Hierarchical Directory Structure

In this section, we consider how to merge two copies of a directory that has been independently updated in different partitions. Logically, the directory structure is a tree[1] but any directory can be replicated. A directory can be viewed as a set of records, each one containing the character string comprising one element in the path name of a file. Associated with that string is an index that points at a descriptor (inode) for a file or directory. In that inode is a collection of information about the file. LOCUS generally treats inode as part of the file from the recovery point of view. The significance of this view will become apparent as the reconciliation procedure is outlined.

To develop a merge procedure for any data type, including directories, it is necessary to evaluate the operations which can be applied to that data type. For directories, there are two operations:

insert (character string path element) and
remove (character string path element).

Although these operations have rather simple semantics, the merge rules are not so simple, primarily because:

a) operations (remove, rename and link) may be done to a file in a partition which does not store the file;

b) a file which was deleted in one partition while it was modified in another, wants to be saved;

c) a directory may have to be resolved without either partition storing particular files.

With these situations in mind, we note that no recovery is needed if the version vector for both copies of the directory are identical. Otherwise the basic rules are:

1. Check for name conflicts. For each name in the union of the directories, check that the inode numbers are the same. If they aren't, both file names are slightly altered to be distinguished. The owners of the two files are notified by electronic mail that this action has been taken.
2. The remaining resolution is done on an inode by inode basis, with the rules in general being:
   a) if the entry appears in one directory and not in the other, propagate the entry;
   b) if a deleted entry exists in one directory and not in the other, propagate the delete, unless there has been a modification of the data since the delete;
   c) if both directories have an entry and neither is deleted, no action is necessary;
   d) if both directories have an entry and one is a delete and other is not, the inode is interrogated in each partition; if the data has been modified since the delete, either a conflict is reported or the delete is undone; otherwise the delete is propagated.

Further augmentation to the directory merge algorithm must be done because of *links*. The complete algorithm is given in [POPE83b].

Since recovery may have to be run while users are active, it is necessary that regular traffic be allowed. To accommodate this, we support demand recovery, which is to say that a particular directory can be reconciled out of order to allow access to it with only a small delay.

4.5 Reconciliation of Mailboxes

Automatic reconciliation of user mailboxes is important in the LOCUS replication system, since notification of name conflicts in files is done by sending the user electronic mail. It is desirable that, after merge, the user's mailbox is in suitable condition for general use.

Fortunately, mailboxes are even easier to merge than directories. The reason is that the operations which can be done during partitioned operation are the same: insert and delete, but it is easy to arrange for no name conflicts, and there are no link problems. Further, since mailboxes are not a system data structure, and generally are seen only by the small number of mail programs, support for deletion information can be easily installed.

[1] With the exception of links.

Thus, for each different mail storage format[1] there is a mail merge program that is invoked after the basic file system has been made consistent again. These programs deal with conflicted files detected by the version vector algorithm which the typing system indicates are mail files.

4.6 Conflicts Among Untyped Data Objects

When the system has no mechanisms to deal with conflicts, it reports the matter to the user. In LOCUS, mail is sent to the owner(s) of a given file that is in conflict, describing the problem. It may merely be that certain descriptive information has been changed. Alternately, the file content may be in conflict. In any case, files with unresolved conflicts are marked so normal attempts to access them fail, although that control may be overridden. A trivial tool is provided by which the user may rename each version of the conflicted file and make each one a normal file again. Then the standard set of application programs can be used to compare and merge the files.

5 Dynamic Reconfiguration

5.1 Introduction

Transparency in LOCUS applies not only to the static topology of the network, but to the configuration changes themselves. The system strives to insulate the users from reconfigurations, providing continuing operation with only negligible delay. Requiring user programs to deal with reconfiguration would shift the network costs from the operating system to the applications programs.

This section discusses the concept of transparency as it relates to a dynamic network environment, gives several principles that the operating system should follow to provide it, and presents the reconfiguration protocols used in LOCUS. The protocols make use of a high-level synchronization strategy to avoid the message overhead of two-phased commits or high-level ACKs, and are largely independent of the specific architecture of LOCUS.

[1] There are two storage formats in LOCUS; one in which multiple messages are stored in a single file, the default, and another where each message is a different file, and messages are grouped by parent directory. This second storage discipline is used by the mail program *mh*.

The high-level protocols of LOCUS assume that the underlying network is fully connected. By this we mean that if site A can communicate with site B, and site B with site C, then site A can communicate with site C. In practice, this may be done by routing messages from A to C through B, although the present implementation of LOCUS runs on a broadcast network where this is unnecessary. The assumption of transitivity of communication significantly simplifies the high-level protocols used in LOCUS.

The low-level protocols enforce that network transitivity. Network information is kept internally in both a high-level status table and a collection of virtual circuits.[1] The two structures are, to some extent, independent. Membership in the partition does not guarantee the existence of a virtual circuit, nor does an open virtual circuit guarantee membership in the partition. Failure of a virtual circuit, either on or after open, does, however, remove a node from a partition. Likewise removal from a partition closes all relevant virtual circuits. All changes in partitions invoke the protocols discussed later in this paper.

The system attempts to maintain file access across partition changes. If it is possible, without loss of information, to substitute a different copy of a file for one lost because of partition, the system will do so. If, in particular, a process loses contact with a file it was reading remotely, the system will attempt to reopen a different copy of the same version of the file.

The ability to mount filegroups independently gives great flexibility to the name space. Since radical changes to the name space can confuse users, however, this facility is rarely used for that purpose, and that use is not supported in LOCUS. The reconfiguration protocols require that the mount hierarchy be the same at all sites.

[1] The virtual circuits deliver messages from *site* A to *site* B (the virtual circuits connect sites, not processes) in the order they are sent. If a message is lost, the circuit is closed. The mechanism defends the local site from the slow operation of a foreign site.

5.2 Requirements for the Reconfiguration Protocols

The first constraint on the reconfiguration protocol is that it maintain consistency with respect to the internal system protocols. All solutions satisfying this constraint could be termed correct. Correctness, however, is not enough. In addition to maintaining system integrity, the solution must insulate the user from the underlying system changes. The solution should not affect program development, and it should be efficient enough that any delays it imposes are negligible.

As an example of a "correct" but poor solution to the problem, the system could handle only the boot case, where all machines in the network come up together. Any failures would be handled by a complete network reboot. Such a solution would easily satisfy the consistency constraint; however, one might expect murmurs of complaint from the user community.

Similarly, a solution that brings the system to a grinding halt for an unpredictable length of time at unforseeable intervals to reconstruct internal tables might meet the requirement of correctness, but would clearly be undesirable.

Optimally, the reconfiguration algorithms should not affect the user in any matter whatsoever. A user accessing resources on machine A from machine B should not be affected by any activity involving machine C. This intuitive idea can be expressed in several principles:

1. User activity should be allowed to continue without adverse affect, provided no resources are lost.
2. Any delay imposed by the system on user activity during reconfiguration should be negligible.
3. The user should be shielded from any transient effects of the network configuration.
4. Any activity initiated after the reconfiguration should reflect the state of the system after the reconfiguration.
5. Specialized users should be able to detect reconfigurations if necessary.
6. No user should be penalized for increased availability of resources.[1]

All these principles are fairly intuitive. They merely extend the concept of network transparency to a dynamic network and express a desire for efficiency. They do, however, give tight constraints on the eventual algorithms. For example, those operations with high delay potentials must be partitioned in such a way that the tasks relevant to a specific user request can be run quickly, efficiently, and immediately.

The principles have far-reaching implications in areas such as file access and synchronization. Suppose, for example, a process were reading from a file replicated twice in its partition. If it were to lose contact with the copy it was reading, the system should substitute the other copy (assuming, of course, that it is still available). If a more recent version became available, the process should continue accessing the old version, but this must not prevent other processes from accessing the newer version.

These considerations apply equally to all partitions, and no process should loose access to files simply because a merge occurred. While the LOCUS protocols insure synchronization within a partition, they cannot do so between partitions. Thus, it is easy to contrive a scenario where the system must support conflicting locks within a single partition, and invoke any routines necessary to deal with inconsistencies that result.

5.3 Protocol Structure

As noted before, the underlying LOCUS protocols assume a fully-connected network. To insure correct operation, the reconfiguration strategy must guarantee this property. If, for instance, a momentary break occurs between two sites, all other sites in the partition must be notified of the break. A simple scan of available nodes is insufficient.

The present strategy splits the reconfiguration into two stages: first, a *partition* protocol runs to find fully-connected sub-networks; then a *merge* protocol runs to merge several such sub-networks into a full partition. The partition protocol affects only those sites previously thought to be up. It divides a partition into sub-partitions, each of which is guaranteed to be fully-connected and disjoint from all other

[1] This last point may cause violations of synchronization policies, as discussed below.

sub-partitions. It detects all site and communications failures and cleans up all affected multi-site data structures, so that the merge protocol can ignore such matters. The merge protocol polls the set of available sites, and merges several disjoint sub-partitions into one.

After the new partition is established, the recovery procedure corrects any inconsistencies brought about either by the reconfiguration code itself, or by activity while the network was not connected. Recovery is concerned mainly with file consistency. It schedules update propagation, detects conflicts, and resolves conflicts on classes of files it recognizes.

All reconfiguration protocols are controlled by a high-priority kernel process. The partition and merge protocols are run directly by that process, while the recovery procedure runs as a privileged application program.

5.4 The Partition Protocol

Communication in a fully-connected network is an equivalence relation. Thus the partitions we speak about are partitions, in the strict mathematical sense of the set of nodes of the network. In normal operation, the site tables reflect the equivalence classes: all members of a partition agree on the status of the general network. When a communication break occurs, for whatever reason, these tables become unsynchronized. The partition code must re-establish the logical partitioning that the operating system assumes, and synchronize the site tables of its member sites to reflect the new environment.

In general, a communication failure between any two sites does not imply a failure of either site. Failures caused by transmission noise or unforeseen delays cannot be detected directly by foreign sites, and will often be detected in only one of the sites involved. In such situations, the partition algorithm should find maximum partitions: a single communications failure should not result in the network breaking into three or more parts.[1] LOCUS implements a solution based on iterative intersection.

[1] Breaking a virtual circuit between two sites aborts any ongoing activity between those two sites. Partition fragmentation must be minimized to minimize the loss of work.

A few terms are helpful for the following discussion. The *partition set*, P_α, is the set of sites believed up by site α. The *new partition set*, P_α', is the set of sites known by α to have joined the new partition.

Consider a partition P after some set of failures has occurred. To form a new partition, the sites must reach a consensus on the state of the network. The criterion for consensus may be stated in set notation as: for every $\alpha,\beta \in P$, $P_\alpha = P_\beta$. This state can be reached from any initial condition by taking successive intersections of the partition sets of a group of sites.

When a site α runs the partition algorithm, it polls the sites in P_α. Each site polled responds with its own partition set $P_{pollsite}$. When a site is polled successfully, it is added to the new partition set P_α', and P_α is changed to $P_\alpha \bigcap P_{pollsite}$. α continues to poll those sites in P_α but not in P_α' until the two sets are equal, at which point a consensus is assured, and α announces it to the other sites.

Translating this algorithm into a working protocol requires provisions for synchronization and failure recovery. These two requirements are antagonistic--while the algorithm requires that only one active site poll for a new partition, and that other sites join only one new partition, reliability considerations require that sites be able to change active sites when one fails--and make the protocol intrinsically complex. Space precludes including the details of the algorithm.

5.5 The Merge Protocol

The *merge* procedure joins several partitions into one. It establishes new site and mount tables, and re-establishes CSS's for all the file groups. To form the largest possible partition, the protocol must check all possible sites, including, of course, those thought to be down[1]. In a large network, sequential polling results in a large additive delay because of the timeouts and retransmissions necessary to determine the status of the various sites. To minimize this effect, the merge strategy polls the sites asynchronously.

[1] In a large network with gateways one can optimize by polling the gateways.

The algorithm itself is simple. The site initiating the protocol sends a request for information to all sites in the network. Those sites which are able respond with the information necessary for the initiating site to build the global tables. After a suitable time, the initiating site gives up on the other sites, declares a new partition, and broadcasts its composition to the world.

The algorithm is centralized and can only be run at one site, and a site can only participate in one protocol at a time, so the other sites must be able to halt execution of the protocol. To accomplish this, the polled site sends back an error message instead of a normal reply:

```
IF ready to merge THEN
        IF merging AND actsite === locsite THEN
                IF fsite < locsite THEN
                        actsite := fsite;
                        halt active merge;
                ELSE decline to merge
                FI
        ELSE actsite := fsite;
        FI
ELSE decline to merge
FI
```

If a site is not ready to merge, then either it or some other site will eventually run the merge protocol.

The major source of delay in the merge procedure is in the timeout routines that decide when the full partition has answered. A fixed length timeout long enough to handle a sizeable network would add unreasonable delay to a smaller network or a small partition of a large network. The strategy used must be flexible enough to handle the large partition case and the small partition case at the same time.

The merge protocol waits longer when there is a reasonable expectation that further replies will arrive. When a site answers the poll, it sends its partition information in the reply. Until all sites believed up by some site in the new partition have replied, the timeout is long. Once all such sites have replied, the timeout is short.

5.6 The Cleanup Procedure

Even before the partition has been reestablished, there is considerable work that each node can do to clean up its internal data structures. Essentially, each machine, once it has decided that a particular site is unavailable, must invoke failure handling for all resources which it's processes were using at that site, or for all local resources which processes at that site were using. The action to be taken depends on the nature of the resource and the actions that were under way when failure occurred. The cases are outlined in the table below.

Local Resource in Use Remotely

Resource	Failure Action
File (open for update)	Discard pages, close file and abort updates
File (open for read)	Close file

Remote Resource in Use Locally

Resource	Failure Action
File (open for update)	Discard pages, set error in local file descriptor
File (open for read)	Internal close, attempt to reopen at other site

Interacting Processes

Failure Type	Action
Remote Fork/Exec, remote site fails	return error to caller
Fork/Exec, calling site fails	notify process
Distributed Transaction	abort all related subtransactions in partition

Once the machines in a partition have mutually agreed upon the membership of the partition, the system must select, for each filegroup it supports, a new synchronization site. This is the site to which the LOCUS file system protocols direct all file open requests. Once the synchronization site has been selected, that site must reconstruct the lock table for all open files from the information remaining in the partition. If there are operations in progress which would not be permitted during normal behavior, some action must be taken. For example, file X is open for update in two partitions, the system policy permits only one such use at a time, and a merge occurs. The desired action is to permit these operations to continue to completion, and only then per-form file system conflict analysis on those resources.[1] Finally, the recovery procedure described in section 4 is run for each filegroup to which it is necessary.

After all these functions have been completed, the effect of topology change has been completely processed. For most of these steps, normal processing at all of the operating nodes continues unaffected. If a request is made for a resource which has not been merged yet, the normal order of processing is set aside to handle that request. Therefore, higher level reconfiguration steps, such as file and directory merge, do not significantly delay user requests.

5.7 Protocol Synchronization

The reconfiguration procedure breaks down into three distinct components, each of which has already been discussed. What remains is a discussion of how the individual parts are tied together into a robust whole. At various points in the procedure, the participating sites must be synchronized, and control of the protocol must be handed to a centralized site. Those sites not directly involved in the activity must be able to ascertain the status of the active sites to insure that no failures have stalled the entire network.

One approach to synchronization would be to add ACKs to the end of each section of the protocol, and get the participants in lock-step before proceeding to the next section. This approach increases both the message traffic and the delay, both critical performance quantities. It also requires careful analysis of the critical sections in the protocols to determine where a commit is required, and the implementation of a commit for each of those sections. If a site fails during a synchronization stage, the system must still detect and recover from that failure.

LOCUS reconfiguration uses an extension of a "failure detection" mechanism for synchronization control. Whenever a site takes on a passive role in a protocol, it checks periodically on the active site. If the active site fails, the passive site can restart the protocol.

As the various protocols execute, the states of both the active and the passive sites change. An active site at one instant may well become a passive site the next, and a passive site could easily end up waiting for another passive site. Without adequate control, this could lead to circular waits and deadlocks.

One solution would be to have passive sites respond to the checks by returning the site that they themselves are waiting for. The checking site would then follow that chain and make sure that it terminated. This approach could require several messages per check, however, and communications delays could make the information collected obsolete or misleading.

Another alternative, the one used in LOCUS, is to order all the stages of the protocol. When a site checks another site, that site returns its own status information. A site can wait only for those sites who are executing a portion of the protocol that precedes its own. If the two sites are in the same state, the ordering is by site number. This ordering of the sites is complete. The lowest ordered site has no site to legally wait for; if it is not active, its check will fail, and the protocol can be re-started at a reasonable point.

While no synchronization "failures" can cause the protocols to fail, they can slow execution. Without ACKs, the active site cannot effectively push its dependents ahead of itself through the stages of the protocol. Nor can it insure that two passive sites always agree on the present status of the reconfiguration. On the other hand, careful design of the message sequences can keep the windows where difficulties can occur small, and the normal case executes rapidly.

6 Experience

Locus has now been operating for about two years at UCLA, and has been installed at a few other sites. Most of the experience with a large network configuration has occurred at UCLA. A 17 Vax-11/750 Ethernet network is the host facility, with additional machines, including a Vax-11/780, to be added shortly. The network is typically operated as three separate Locus networks; one for production use, one as a beta test net, and a few machines for developing new system software. On the beta net, for example, it has recently been typical to have 5 machines operational with about 30-40 users. The production net configuration is correspondingly larger. These systems are used for virtually all the interactive computing in the UCLA Computer Science Department, except for those classes using the Vax-11/780 that has not yet been converted. We estimate that over 100,000 connect hours have been

[1] LOCUS currently does not support this behavior.

delivered by the Vax Locus networks by early 1983. Most of the user experience with Locus has been acquired on a version of the system that did not support replication of files, the nested transaction facility or general remote process forking. The following observations largely reflect the experience at UCLA.

First, it is clearly feasible to provide high performance, transparent distributed system behavior for file and device access, as well as remote, interprocess interaction. Measurements consistently indicate that Locus performance equals Unix in the local case, and that remote access in general, while somewhat slower than when resources are local, is close enough that no one typically thinks much about resource location because of performance reasons. This ability to provide a substantial degree of *performance transparency* surprised us. There is still no indication that Ethernet bandwidth is any significant limitation. It is, however, difficult to swap load images across the network with high performance because of the software cost of packet disassembly and reassembly for the current hardware and low level protocols. Much larger configurations clearly cannot share the same broadcast cable of that bandwidth, of course.

Most of the problems which were encountered by users resulted from those situations where transparency was not completely supported, either because the implementation was not finished, or because explicit decisions to make exceptions were made. Overwhelming however, experience with transparency has been very positive; giving it up, once having had it, would be nearly unthinkable.

LOCUS executes programs locally as the default. We found that the primary motivation for remote execution was load balancing. Remote process execution is also used to access those few peripheral devices which are not remotely transparent. We expect that the remote processing facility will also be heavily used when a LOCUS network is composed of heterogeneous cpus. The non-Vax machines at UCLA (PDP-11's) were decommissioned before the remote processing facilities were generally available, so this activity is not heavily represented in operational experience.

Experience with replicated storage was limited at the time this paper was written, so few firm conclusions can be drawn, except that we certainly cursed its absence.

As usual, the tasks involved in providing a reasonably production quality environment were far more extensive and much more painful than anyone had anticipated, even though we held university standards for 'production quality', and we all knew to expect this phenomenon.

We estimate that, from the project inception in 1979 till early 1983, about 50 man years were spent on Locus. This effort included almost a year in a conceptual phase (transparency was only a vague idea at that time, for example), an extensive design phase that lasted almost a year, initial construction on PDP-11s, multiple ports to the Vax and Ethernet, development of the general-use, development and testing configuration at UCLA, extensive debugging, some redesign and reimplementation to more fully maintain Unix compatibility, and a significant number of masters and PhD theses. Nevertheless, a great deal remains at the time this paper was written. Fortunately, most of that work is now being done in a commercial environment rather than by a university research project.

7 Conclusions

The most obvious conclusion to be drawn from the LOCUS work is that a high performance, network transparent, distributed file system which contains all of the various functions indicated throughout this paper, is feasible to design and implement, even in a small machine environment.

Replication of storage is valuable, both from the user and the system's point of view. However, much of the work is in recovery and in dealing with the various races and failures that can exist.

Nothing is free. In order to avoid performance degradation when resources are local, the cost has been converted into additional code and substantial care in implementation architecture. LOCUS is approximately a third bigger than Unix and certainly more complex.

The difficulties involved in dynamically reconfiguring an operating system are both intrinsic to the problem, and dependent on the particular system. Rebuilding lock tables and synchronizing processes running in separate environments are problems of inherent difficulty. Most of the system-dependent problems can be avoided, however, with careful design.

The fact that LOCUS uses specialized protocols for operating system to operating system communication made it possible to control message traffic quite selectively. The ability to alter specific protocols to simplify the reconfiguration solution was particularly appreciated.

The task of developing a protocol by which sites would agree about the membership of a partition proved to be surprisingly difficult. Balancing the needs of protocol synchronization and failure detection while maintaining good performance presented a considerable challenge. Since reconfiguration software is run precisely when the network is flaky, those problems are real, and not events that are unlikely.

Nevertheless, it has been possible to design and implement a solution that exhibits reasonably high performance. Further work is still needed to assure that scaling to a large network will successfully maintain that performance characteristic, but our experience with the present solution makes us quite optimistic.

In summary, however, use of LOCUS indicates the enormous value of a highly transparent, distributed operating system. Since file activity often is the dominant part of the operating system load, it seems clear that the LOCUS architecture, constructed on a distributed file system base, is rather attractive.

8 Acknowledgements

The authors would like to thank all those members of the research contract who have helped, over the years, in the design, implementation and support of LOCUS. In addition, special recognition should be given to Terry Gray, who is responsible for keeping the LOCUS facility running at UCLA.

9 Bibliography

ALSB 76 Alsberg, P. A., Day, J. D., *A Principle for Resilient Sharing of Distributed Resources,* Proceedings of Second International Conference on Software Engineering, October 1976.

BARL 81 Bartlett J.F., *A NonStop Kernel,* Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

BIRR 82 Birrell, A. D., Levin, R., Needham, R. M., Schroeder, M. D., *Grapevine: An Exercise in Distributed Computing,* CACM, Vol. 25, No. 4, April 1982, pp. 260-274.

DION 80 Dion, J., *The Cambridge File Server,* Op. Sys. Rev, 14(4), pp. 26-35, Oct. 1980.

FAIS 81 Faissol, S., *Availability and Reliability Issues in Distributed Databases,* Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, 1981.

GOLD 83 Goldberg, A., and G. Popek, *Measurement of the Distributed Operating System LOCUS,* UCLA Technical Report, 1983.

GRAY 78 Gray, J. N., *Notes on Data Base Operating Systems,* Operating Systems, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, 393-481.

HOLL 81a Höller E., *Multiple Copy Update,* Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 284-303.

HOLL 81b Holler E., *The National Software Works (NSW),* Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 421-442.

JONE 82 Jones, M.B., R. F. Rashid and M. Thompson, *Sesame: The Spice File System,* Draft from CMU, Sept. 82.

LAMP 81a Lampson B.W., *Atomic Transactions,* Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 246-264.

LAMP 81b Lampson B.W., *Ethernet, Pub and Violet,* Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 446-484.

LAMP 82 Lampson, B.W. and H.E. Sturgis, *Crash Recovery in a Distributed Data Storage System,* CACM (to appear)

LELA 81 LeLann G., *Synchronization,* Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 266-282.

LIND 79 Lindsay, B. G. et. al., *Notes on Distributed Databases,* IBM Research Report RJ2571(33471), IBM Research Laboratory, San Jose, CA, July 14, 1979, 44-50.

MENA 77 Menasce, D. A., Popek, G. J., Muntz, R. R., *A Locking Protocol for Resource Coordination in Distributed Systems,* Technical Report UCLA-ENG-7808, Dept. of Computer Science, UCLA, October 1977.

MEUL 83 Meuller E., J. Moore and G. Popek, *A Nested Transaction System for LOCUS,* SOSP '83.

MITC 82 Mitchell J.G. and J. Dion, *A Comparison of Two Network-Based File Servers,* CACM, Vol. 25, No. 4, April 1982.

NELS 81 Nelson, B.J., *Remote Procedure Call,* Ph.D. Dissertation, Report CMU-CS-81-119, Carnegie-Mellon University, Pittsburgh, 1981.

PARK 83 Parker, D. Stott, Popek, Gerald J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C., *Detection of Mutual Inconsistency in Distributed Systems,* IEEE Transactions of Software Engineering, May 1983.

POPE 81 Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., *LOCUS: A Network Transparent, High Reliability Distributed System,* Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

POPE 83a Popek, Gerald J., and Walker, Bruce J., *Network Transparency and its Limits in a Distributed Operating System,* Submitted for Publication.

POPE 83b Popek G.J., et.al., *LOCUS System Architecture,* LOCUS Computing Corporation Technical Report, 1983.

RASH 81 Rashid, R.F., and Robertson, G.G., *Accent: A Communication Oriented Network Operating System Kernel,* Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

REED 78 Reed, D. P., *Naming and Synchronization in a Decentralized Computer System,* Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, M.I.T., 1978.

REED 80 Reed, D.P, and Svobodova L, *SWALLOW: A Distributed Data Storage System for a Local Network,* Proc. of the International Workshop on Local Networks, Zurich, Switzerland, August 1980.

RITC 78 Ritchie, D. and Thompson, K., *The UNIX Timesharing System,* Bell System Technical Journal, vol. 57, no. 6, part 2 (July - August 1978), 1905-1930.

SALT 78 Saltzer J.H., *Naming and Binding of Objects,* Operating Systems, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, 99-208.

SPEC 81 Spector, A. Z., *Performing Remote Operations Efficiently on a Local Computer Network,* CACM, Vol. 25, No. 4, April 1982.

STON 76 Stonebraker, M., Wong, E., Kreps, P., *The Design and Implementation of Ingres,* ACM Transactions on Database Systems, Vol. 1, No. 3, Sept. 1976, pp. 189-222.

STUR 80 Sturgis, H.E. J.G, Mitchell and J. Israel, *Issues in the Design and Use of a Distributed File System,* Op. Sys. Rev, 14(3), pp. 55-69, July 1980.

SVOB 81 Svobodova, L., *A Reliable Object-Oriented Data Repository For a Distributed Computer,* Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

THIE 83 Thiel, G., *Partitioned Operation and Distributed Data Base Management System Catalogues,* Ph.d. Dissertation, Computer Science Department, University of California, Los Angeles, June 1983.

THOM 78 Thomas, R.F., *A Solution to the Concurrency Control Problem for Multiple Copy Data Bases,* Proc. Spring COMPCON, Feb 28-Mar 3, 1978.

WALK 83 Walker, B.J., *Issues of Network Transparency and File Replication in Distributed Systems: LOCUS,* Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, 1983.

WATS 81 Watson R.W., *Identifiers (Naming) in Distributed Systems,* Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 191-210.