LOCUS

A Network Transparent, High Reliability Distributed System[1]

G. Popek, B. Walker, J. Chow, D. Edwards,

C. Kline, G. Rudisin, G. Thiel

University of California at Los Angeles

LOCUS is a distributed operating system that provides a very high degree of network transparency while at the same time supporting high performance and automatic replication of storage. By network transparency we mean that at the system call interface there is no need to mention anything network related. Knowledge of the network and code to interact with foreign sites is below this interface and is thus hidden from both users and programs under normal conditions. LOCUS is application code compatible with Unix[2], and performance compares favorably with standard, single system Unix. LOCUS runs on a high bandwidth, low delay local network. It is designed to permit both a significant degree of local autonomy for each site in the network while still providing a network-wide, location independent name structure. Atomic file operations and extensive synchronization are supported.

Small, slow sites without local mass store can coexist in the same network with much larger and more powerful machines without larger machines being slowed down through forced interaction with slower ones. Graceful operation during network topology changes is supported.

1. Introduction

LOCUS is an integrated distributed system whose goals include a) making the development of distributed applications no more difficult than single machine programming, and b) realizing the potential that distributed systems with redundancy have for highly reliable, available operation. Much of the research reported in this paper is the subject of a forthcoming Ph.D. dissertation [Walker82]. The system is application code compatible with the Unix operating system, [Ritchie74] and a prototype running on DEC PDP-11s, models 44, 45, and 70, connected by local networks ranging in speed from 1 to 10 megabits/second is operational at UCLA. LOCUS has also been ported to the DEC VAX.

[1]This research was supported by the Advanced Research Projects Agency under research contract DSS MDA 903-77-0211.

[2]Unix is a registered trademark of Western Electric.

1.1 System Design Assumptions and Goals

Most existing distributed systems were constructed by making relatively minor modifications to adapt the single machine systems to permit interaction with other copies of themselves, or even with other systems. The basic structure and operational philosophy of those systems was invariably preserved. Our goal was to understand, given the freedom to start largely anew, what the structure of a distributed system ought to be. One fundamental assumption we made concerned the interconnection network; it was of high bandwidth, low delay, with a low error rate - so called "thick wire" networks, represented by such examples as the Ethernet. Broadcast capability was not assumed. We explicitly ruled out Arpanet or Telenet style networks because of their limited bandwidth and significant delay. Satellite based networks were also not explicitly addressed because of their significant delay.

General applications were to be supported, with focus on "computer utility" functions and data management. Essential goals were to provide high reliability and availability in a distributed environment. Performance was also critical, as was providing a much simpler user and programming interface than that of current distributed systems. We were willing to insist that all sites in the network run a given system, but the sites should be able to vary widely in power and storage capacity.

1.2 LOCUS Overview

LOCUS is a distributed operating system whose architecture strongly addresses our goals of network transparency, high reliability and availability, and good performance. The machines in a LOCUS net cooperate to give all users the illusion of operating on a single machine; the network is essentially invisible. Invisible here means that there is no need to refer to a specific site or to the network at all. There is considerable belief and evidence that a uniform interface to all resources is very attractive in the distributed environment. Support for this illusion is entirely within the operating system code. Nevertheless, each machine is a complete system and can operate gracefully alone. LOCUS is designed to automatically replicate resources to the degree indicated by associated reliability profiles. Graceful operation in the face of network partitions, as well as nodal failures, is supported. Finally, all this increased functionality is provided within the constraint of good performance. It is expected that these characteristics are suitable for the support of a wide variety of applications, including general distributed computing, office automation, and database management.

Note that LOCUS was designed to provide network transparency access to and replication of both data resources and processing resources or agents. This paper primarily addresses issues in the data resource component, in part because functionality in that area is essential. More importantly, however, concepts needed for solution in the data area provide a solid framework for extension, and further, the mechanisms that must be built are much of what is needed for distributed processing support.

Section 2 of this paper is dedicated exclusively to motivating the concept of low-level network transparency. In section 3 we discuss the components of the software system architecture necessary to implement network transparency and to achieve increased reliability and availability. Although performance is mentioned in section 3, a more detailed discussion, including both the design principles followed and the results attained, is presented in section 4.

2. Network Transparency

As real distributed systems come into existence, an unpleasant truth is being learned: the development of software for distributed applications is often far harder to design, implement, debug and maintain than the analogous application written for a centralized system. There are several reasons. First, typically the means by which a remote resource is accessed is different from, and more complicated than, the access method for a corresponding local resource. For example, one *opens* a local file, but may have to execute a multistep *file transfer protocol* to access a remote file. Second, the error modes of a collection of machines connected by a network appear much more substantive than in a centralized environment. For example, on a single machine one rarely worries about *partial failure.* In a distributed environment, it is quite reasonable to expect that one or several of the sites supporting the application may fail at awkward moments while others continue unaware. One typically assumes by contrast on a central machine that a system failure causes the application to stop.

A further problem with distributed systems is that local storage may be limited, necessitating that the user explicitly move copies of files around the network, archiving and garbage collecting his own storage. Redundant copies for the sake of reliability are the user's concern. The user must keep track of different versions of what is intended to be the same file, especially when the copies have resulted from network partitions (leading to parallel changes). As a result, the application program and user must explicitly deal with each of these facts, at considerable cost in additional software.[1] On a centralized machine, with a single integrated file system, many of these problems do not exist, or are more gracefully handled.

An appealing solution to this increasingly serious problem is to develop a network operating system that supports a high degree of *network transparency;* all resources are accessed in the same manner independent of their location. If *open (file-name)* is used to access local files, it also is used to access remote files. That is, the network becomes "invisible", in a similar manner to the way that virtual memory hides secondary store. Of course, one still needs some way to control resource location for optimization purposes, but that control should be separated from the syntax and semantics of the system calls used to *access* the resources. Ideally then, one would like the graceful behavior of an integrated storage system for the entire network while still retaining the many advantages of the distributed system architecture. That is, the existence of the network should not concern the user or application programs in the way that resources are accessed. If such a goal could be achieved, its advantages would include the following.

1. *Easier software development.*
   Since there is only one way to access resources, and the details of moving data across the network are built in, individual software packages do not require special software for this purpose. As a result, many traditionally distributed applications could be implemented in the same manner as local operations.

2. *Incremental Change Supported.*
   Changes made below the level of the network wide storage system are not visible to application software. Therefore, changes in both hardware and software resources can be made easily.

3. *Potential for Increased Reliability.*
   Local networks, with a fair level of redundancy of resources (both hardware and stored data), possess considerable potential for reliable, available operation. However, if this potential is to be realized, it must be possible to substitute various resources for one another easily (including processors, copies of files and programs, etc.). A uniform interface which hides the binding of those resources to programs would seem to be necessary if the high reliability goal is to be realized.

4. *Simpler User Model.*
   By taking care of the details of managing the network, the user sees a conceptually simpler storage facility, composed merely of files, without machine boundaries, replicated copies, etc. The same is true for other user visible resources.

There are a number of aspects to network transparency. Is the location of a resource apparent in the (by necessity global) resource name? Preferably not, since one then has the freedom to move resources for optimization or reliability reasons without changing application software. How are the much richer set of errors which occur in a distributed system reflected to the caller? Given that most implementations to support transparency will involve considerable levels of mapping, how can that be done without imposing any significant performance penalty? Each of these issues is addressed by the LOCUS design.

There are also some system aspects that militate against full transparency. If the hardware bases of each site aren't the same, then it may be necessary to have many different executable load modules corresponding to a given name, so that when a user (or another program) issues a standard name, the appropriate file is invoked as a function of the machine on which the operation is to be performed. There are other examples as well; they all have the characteristic that a standard name is desired for a function or object that is replicated at some or all sites, and a reference to that name needs to be mapped to the local, or nearest instance in normal circumstances.

[1] Recently one of our staff built a network wide printer spooler for our local network which at the time ran Arpanet software. Over twenty processes and several thousand lines of code were required to deal with the environment; the actual printer spooler included only two processes.

Nevertheless, in other circumstances, a globally unique name for each instance is also necessary; to install software, do system maintenance functions, etc. A solution that preserves network transparency and provides globally unique names within the normal name space while still supporting site dependent mapping for these special cases is needed. Our solution to this problem involves appending tags to file names which by convention indicate site dependent information. Various system and user call options default the tag value appropriately.

3. System Architecture

In this section, we discuss several key components of the software system which address the goals of network transparency, reliability and availability. First there is a more detailed discussion of global naming, which was mentioned in section 2. Second is a description of both the policy and mechanism involved in network-wide data access synchronization. Third is a treatment of several elements designed to increase reliability and availability including data replication and an atomic action/commit mechanism. Also reviewed are aspects of local autonomy and prevention of error propagation from machine to machine. The final component concerns the handling of sites entering and leaving the network. Work in this area involves recognition and initialization issues, cleanup issues and the major issue of checking and resolving the consistency of multiple copies of files.

3.1. Global Names

In developing a naming mechanism for a distributed system, one faces a number of issues. What is the nature of user sensible, global names? What are the system internal global names? How, and when, is mapping between the two done? How, and when, are names of either sort allocated and invalidated? Should there be multiple levels of naming? The design choices made in LOCUS, and the basic reasons, are outlined below.

There are two significant levels of abstraction in the LOCUS file system, which serves as the heart of the naming mechanism. The user and application program view of object names in LOCUS is analogous to a single, centralized Unix environment; virtually all objects appear with globally unique names in a single, uniform, hierarchical name space. Each object is known by its path name in a tree[1], with each element of the path being a character string. Any user and/or program can set a "working directory" appropriately to avoid having to employ the longer path names that this large hierarchy may imply. No notion of object location appears in the path name, so that an object can easily be moved without changing its visible name.

Low level names in LOCUS are also globally unique. The name space is organized around the concept of *file groups.*[2] As in Unix, each file group (of which there may be many on a single standard Unix system) is composed of a section of a mass store, and divided into two parts; a small set of file descriptors which serve as a simple low level "directory", and a large number of standard sized data blocks. A file descriptor (inode) contains pointers to the data blocks which compose that file. Data blocks can be used as indirect pointer blocks for large files. Under these conditions, a low level file name is a <file group number, file descriptor number>.[1]

[1]There are a few exceptions to the tree structure, provided by Unix links.

[2]The term file group in this paper corresponds directly to the Unix term *file system.*

LOCUS uses the same method, although since a file group can be replicated, one speaks of a *logical* file group number. There is an additional map from logical to physical file group number; this is a one to many map.[2] Files may be replicated only at sites which store the containing file group. However, it is not necessary to replicate a given file at all such sites. Thus the amount of storage allocated to the physical versions of a file group can differ, since each can be incomplete. When a file is not stored at a physical file group, the file descriptor there is not kept current.

The application visible naming hierarchy is implemented by having certain low level files serve as high level directories. An entry in such a directory contains an element of a path name and the index of the file descriptor corresponding to the next directory or data file in the path.

The collection of file groups therefore represent a set of naming trees; these are patched together to form the single, network wide naming tree by system code and a network wide *mount* table.

Creating a file requires allocating a file descriptor, so race conditions could lead to the same file descriptor being allocated to different files. Therefore, some control is needed. One solution would be not to use the descriptor number as part of the name; rather have some unique ID as the name, and store it in the descriptor as well as in the higher level directory entry that pointed at the descriptor. The descriptor number would be treated only as a guess and the IDs would be compared at each access attempt. No particular method to manage file deletion and subsequent reuse of file descriptors is then needed, assuming that IDs are never reused.

However, in LOCUS, this approach would have meant changing the contents of higher level directories, which are application code visible in Unix. Therefore, we chose to retain the file descriptor number as part of the name. To avoid allocation races, the file descriptor space for each file group is logically partitioned; each storage site for a file group may allocate descriptors only from its own part. Deletion of a file requires interaction among all storage sites for a given file before the descriptor may actually be reused.

Given the high and low level name space, the next issue concerns the map between the two. In LOCUS, conversion to low level name is done as soon as possible; then only low level names are passed within the system. In this way, one avoids remapping names. This philosophy could have been carried even further; one could have found the disk address of a page, and used that as a component of the 'name' to be passed. We did not do this for two reasons. First, since a given file may not be stored at all storage sites for that file group, address information might not be available, and so another mechanism would be needed anyway. Second, we wanted each storage site to be able to provide some significant degree of internal system data structure consistency by itself. For example, while a site updating a remotely stored file can cause the contents of that file to be inappropriately altered at the storage site, it cannot damage other files stored at the storage site (an action which would be possible if physical page addresses were being used.)

[1]Devices also appear in the file system as leaves in the name structure and they have global names.

[2]It is implemented as an extension to *mounting* a file group.

3.2. Synchronization

Since storage may be replicated, and there are multiple users, the problem of synchronization of access to logical files and their physical counterparts must be addressed if a consistent file system is to be presented to users. Standard Unix is quite bereft of such controls, so these represent an addition to the user and program interface.[1]

3.2.1. Synchronization Policy in LOCUS

The policy in LOCUS is based on a global "multiple readers, single writer" policy. Such a basic policy provides concurrent read access to replicated copies of data while preventing concurrent update. However, the actual policy which has been implemented is more sophisticated for three principal reasons. First, in a modern operating system environment, more functionality is appropriate. For example, when one process forks another, it is best for the newly created process to inherit and retain the access rights held by the parent, since in most cases the family of processes are cooperating on the same task. They can coordinate among themselves if intrafamily synchronization is needed.

Another example results from the hierarchical file system. Whenever a file is opened through the use of a full path name, it is necessary to open for read all the directory files in the path. One expects high traffic of this sort in an environment with many users. Shared read locks would suffice, if it were not for the fact that any file creation or rename implies a write lock to the containing directory. While these locks would be of short duration, the potential performance impact of temporarily blocking *all* file opens in the network that contain that directory in their path name is significant.

The LOCUS solution for reading directories is a new access type, *nolock read.* When a file is opened for nolock read, there is essentially no synchronization with other activity on the file. The only guarantee is that a single operating system read call will return with a consistent set of bits, i.e. from one version of a (possibly dynamically changing) file.[2] Directory entries, and the system software that accesses them, have suitable characteristics to be accessed in this way (although convincing oneself of this fact is not easy). A higher overhead alternative would have been to implement record locks in the file system.

The third reason for a sophisticated synchronization policy is Unix compatibility. Since standard Unix has few restrictions, normal program behavior sometimes would be blocked by a straightforward policy.

3.2.2. Synchronization Mechanism in LOCUS

Many techniques have been proposed for the synchronization mechanisms in distributed systems [Bernstein80]. After careful analysis, we chose a centralized synchronization protocol with distributed recovery. For a given file group, one site is designated as the *Current Synchronization Site (CSS)* This site is responsible for coordinating access to the files contained in the associated file group. That is, all *open* calls involve a message to the CSS (except for nolock reads). In this way it is possible for the CSS to assure that a requester gets the latest version of a file. However, it is not necessary for the CSS to be the site from which data access is obtained. Any site which has a copy of the file can support this open request; that is, a particular *Storage Site* (SS) is designated to support an open request at the time of the open. A fairly general synchronization policy can be supported, since the synchronizing point is centralized for any given decision.

An outline of how actual file open and read operations are handled may serve to clarify the interaction of synchronization and file system structure. Suppose an *open* request is made for file A. After any pathname directory searching (described below), the requesting site has obtained A's low level name (the pair <file group number, file descriptor number>). This name is sent in an open message from the requesting, or using site (US) to the CSS. The CSS selects a storage site and sends it a message. After the storage site decides to provide service, it replies to the CSS, which maintains the synchronization information and notifies the using site. The using site installs the appropriate file descriptor in its internal table (just as if the file were local) and the open is complete. Note that a 3 site open (US, CSS and SS functions all on different sites) is worst case. Message traffic and processing is decreased when any two or all three of the functions are on the same site.

Directory searching is done by the operating system using much of the same mechanism described above. Each pathname component is internally opened and read to find the low-level name of the next pathname component. The major different between these file opens and regular file opens is the use of the nolock read open mode described in the previous section. Via that mode all message traffic is eliminated if a copy of the desired directory file resides locally.

After a non-local file is open, processing at the using site continues as if the file had been local, with the US requesting pages of the file from the SS very much the same as it requests pages from a local disk. Effectively, LOCUS "page faults" across the network for file pages.

The mechanism used in LOCUS is quite different than primary copy strategies [Alsberg78], in that all physical copies of a logical file may be used, even for update. The primary copy strategy requires that all activity involve a single specified copy, with all other copies eventually used for back-up. In this way, optimization that takes into account the location of resources can be done.

3.3. Reliability

Reliability and availability represent a whole other aspect of distributed systems which has had considerable impact on LOCUS. Four major classes of steps have been taken to achieve the potential for very high reliability and availability present in distributed systems with redundancy.

First, an important aspect of reliable operation is the ability to substitute alternate versions of resources when the original is found to be flawed. In order to make the act of substitution as straightforward as possible, it is desirable for the *interfaces* to the resource versions to look identical to the user of those resources. While this observation may seem obvious, especially at the hardware level, it also applies at various levels in software, and is another powerful justification for network transparency. In LOCUS, copies of a file can be substituted for one another with no visibility to application code. Consistency issues are handled by the version vector mechanism discussed in the next section.

[1]The introduction of synchronization control is a minor source of incompatibility for a few existing programs.

[2]This is true as long as the read did not cross a block boundary.

From another point of view, once network transparency is present, one has the *opportunity* to enhance system reliability by substituting software as well as hardware resources when errors are detected. In LOCUS, considerable advantage is taken of this approach. Since the file system supports automatic replication of files transparently to application code, it is possible for graceful operation in the face of network partition to take place. If the resources for an operation are available in a given partition, then the operation may proceed, even if some of those are data resources replicated in other partitions. A partition merge procedure detects any inconsistencies which may result from this philosophy, and for those objects whose semantics the system understands (like file directories, mailboxes, and the like), automatic reconciliation is done. See section 3.4.

Recovery from partitioned operation is done hierarchically, in that first lower level software attempts to merge a file; if it cannot, then the problem is reported to the next level up, eventually to the user (owner) of the file.

A second step in increasing the reliability of operation is the support in LOCUS of file *committing* [Lampson78] [Gray78]. For a given file, one can be assured that either all the updates are done, or none of them are done, even in the face of system crashes and/or network failures. Commit normally occurs automatically at file close time if the file had been open for write, but application software may request commits at any time during the period when a file is open. To do a commit, first the current storage site permanently records the changed file. Then the CSS and user code are notified of successful commit, and user execution may go on. From this point on, the new version of the file is propagated in parallel to other storage sites.

Third, even though a very high level of network transparency is present in the syntax and semantics of the system interface, each site is still largely autonomous. When, for example, a site is disconnected from the network, it can still go forward with work local to it. This goal has had significant architectural impact, which is discussed in section 4.2.

Fourth, the interaction among machines is strongly stylized to promote "arms length" cooperation. The nature of the low level interfaces and protocols among the machines permits each machine to perform a fair amount of defensive consistency checking of system information. As much as feasible, maintenance of internal consistency at any given site does not depend on the correct behavior of other sites. (There are, of course, limits to how well this goal can be achieved.) Each site is master of its own resources, so it can prevent flooding from the network.

3.4. Topology Change and Replicated File Consistency

A procedure within the operating system is executed whenever a network topology change is observed (one or more sites entering or leaving the network). Currently this topology change procedure runs a simple algorithm to choose a coordinating site which polls the others to reconstruct network-wide data structures. It chooses new CSS's for file groups when necessary. The new CSS is then responsible for synchronizing information relating to that file group, and invoking the file group recovery system to establish replicated file consistency.

Given that a data resource can be replicated, a policy issue arises. When the network is partitioned and a copy of that resource is found in more than one partition, can the resource be modified in the various partitions? It was felt important for the sake of availability that such updates be permitted. Of course, this freedom can easily lead to consistency conflicts at partition merge time. However, our view is that such conflicts are likely to be rare, since actual sharing in computer utilities is known to be relatively low.

Further, we developed a simple, elegant algorithm to detect conflicts if they have occurred. See [Parker80]. The core of the method is to keep a *version vector* with each copy of the data object. There are as many elements in the vector as there are sites storing the object.[1] Whenever an update is made to a copy of the object at a given site, that site's element of the version vector associated with the updated copy is incremented. The conflict detection criterion is then very simple. When merging two copies of an object, compare their version vectors. If one dominates the other (i.e. each element is pairwise greater than or equal to its corresponding element), then there is no conflict; the copy associated with the dominating vector should propagate. Otherwise a conflict exists.

Most significant, for those data items whose update and use semantics are simple and well understood, it may be quite possible to reconcile the conflicting versions automatically, in a manner that does not have a "domino effect"; i.e. such a reconciliation does not require any actions to data items that were updated during partitioned operation as a function of the item(s) being reconciled.

Good examples of data types whose operations permit automatic reconciliation are file directories and user mailboxes. The operations which apply to these data types are basically simple: *add* and *remove*. The reconciled version is the union of the several versions, less those items which have been removed.[2] There are of course situations where the system does not understand the semantics of the object in conflict. The LOCUS philosophy in these cases is to report the conflict to the next higher level of software, in case resolution can be done there. Data Management software for example might be capable of such resolution. If there is no higher level software to resolve the conflict, it is reported to the user. It is surprising how many applications have straightforward enough semantics that nearly full operation can be permitted during partitioned operation while still allowing automated reconciliation at merge time.

The special issues of partitioned operation and replicated file recovery after partition have been topics of considerable work, including two dissertations ([Faissol81] considers partitioned operation in applications such as banking and airline reservations while [Rudisin82] will deal with the file recovery issue).

[1]The assumption made in LOCUS is that a file is either replicated a relatively few number of times (the normal case for user data files), or is stored everywhere (the case for system modules needed for a site to run LOCUS). When an *all* bit is set in the file descriptor, it is understood that the file is stored everywhere that the file group exists, and the version vector applies only to a limited number of sites, the only sites which can serve as a storage site for an update.

[2]The situation for directories is somewhat more complex, since it is possible to have a *name* conflict; when two different files of the same name are created in two different partitions. LOCUS detects such conflicts and reports them to the users via electronic mail, and marks the conflicted files so that special action is needed before normal access can occur.

4. Performance and its Impact on Software Architecture

"In software, virtually anything is possible; however, few things are feasible." [Cheatham71] While the goals outlined in the preceding sections may be attainable in principle, the more difficult goal is to meet all of the above while still maintaining good performance within the framework of a well structured system without a great deal of code. A considerable amount of the LOCUS design was tempered by the desire to maintain high performance. Perhaps the most significant design decisions in this respect are:

a) specialized "problem oriented protocols",

b) integrated rather than partitioned function,

c) a lightweight process mechanism for serving network requests inside the operating system kernel,

d) special handling for local operation.

Below, we discuss each of these in turn.

4.1. Problem Oriented Protocols

It is often argued that network software should be structured into a number of *layers*, each one implementing a protocol or function using the characteristics of the lower layer. In this way, it is argued, the difficulties of building complex network software are eased; each layer hides more and more of the network complexities and provides additional function. Thus layers of "abstract networks" are constructed. More recently, however, it has been observed that layers of protocol generally lead to layers of performance cost. (See for example, [Kleinrock75]). In the case of local networks, it is common to observe a cost of up to 5,000 instructions being executed to move a small collection of data from one user program out to the network.[Lampson79]

In a local network, we argue that the approach of layered protocols is frequently wrong, at least as it has been applied in long haul nets. Functionally, the various layers were typically dealing with issues such as error handling, congestion, flow control, name management, etc. In our case, these functions are not very useful, especially given that they have significant cost. The observed error rate on local networks is very low, although certainly not zero. Congestion is generally not a problem; the token ring in use at UCLA [Farber74][Clark78] uses a circulating token, supported by hardware, to control access to the transmission medium. The Ethernet [Xerox80] has an analogous mechanism. Much flow control in LOCUS is a natural consequence of the nature of higher level activity, since many incoming messages to a site result from explicit requests made by that site. Some additional flow control is needed, since a storage site can in principle be swamped by requests from using sites which have open files supported by the given storage site. Name management largely must be done at a high level within LOCUS, so that any nontrivial lower level mechanism within the network layer would largely be ignored.

In fact, despite its limitations, Saltzer's "end to end" argument applies very strongly in LOCUS.[Saltzer80] Summarized very quickly, Saltzer points out that much of the error handling that is done at low levels in computer networks is *redone* at higher levels. The reason is that low level handling cannot mask or provide recovery from higher level faults.

In the design of LOCUS, we found it necessary to deal with a significant collection of error events within the operating system; managing the impact of remote sites failing, synchronization difficulties, etc. Many types of low level errors therefore are also detected by the mechanisms that we needed to build anyway; hence the (expensive) low level supports were dropped. Notice that this principle cannot be followed blithefully. There are certain low level events that cannot be detected at a higher level. These must be addressed in the basic network protocols. So, for example, we support automatic retransmission, as well as certain forms of redundancy in data transmission.

All of these observations lead one to develop specialized *problem oriented protocols* for the problem at hand. That is what occurred in LOCUS. For example, when a user wishes to read a page of a file, the *only* message that is sent from the using site to the storage site is a *read* message request. A *read* is one of the primitive, lowest level message types. There is no connection management, no acknowledgement overhead, etc. The only software ack in this case is the delivery of the requested page. (The hardware does provide a low level acknowledgement.)

Our experience with these lean, problem oriented protocols has been excellent. The effect on system performance has been dramatic. See section 4.5.

4.2. Functional Partitioning

It has become common in some local network developments to rely heavily on the idea of "servers", where a particular machine is given a single role, such as file storage, name lookup, authentication or computation. [Swinehart79] [Rashid80] [Needham 80] We call this approach the *server model* of distributed systems. Thus one speaks of "file servers", "authentication servers", etc. However, to follow this approach purely is inadvisable, for several reasons. First, it means that the reliability/availability of an operation which depends on multiple servers is the product of the reliability of all the machines and network links involved. The server design insures that, for many operations of interest, there will be a number of machines whose involvement is essential.

Second, because certain operations involve multiple servers, it is necessary for multiple machine boundaries to be crossed in the midst of performing the operation. Even though the cost of network use has been minimized in LOCUS as discussed above, it is still far from free; the cost of a remote procedure call or message is still far greater than a local procedure call. One wants a design where there is freedom to configure functions on a single machine when the situation warrants. Otherwise serious performance costs may result, even though the original designers believed their allocation of functions across machine boundaries did not require those boundaries to be crossed by the "tightloops" of certain applications.

Third, since it is unreasonable to follow the server machine philosophy strictly, one is led to multiple implementations of similar functions; to avoid serious performance costs, a local *cache* of information otherwise supplied by a server is usually provided for at least some server functions. The common example is file storage. Even though there may be several file servers on the network, each machine typically has its own local file system. It would be desirable to avoid these additional implementations if possible.

An alternative to the server model is to design each machine's software as a complete facility, with a general file system, name interpretation mechanism, etc. Each machine in the local network would run the same software, so that there would be only one implementation. Of course, the system would be highly configurable, so that adaptation to the nature of the supporting hardware as well as the characteristics of use could be made. We call this view the *integrated model* of distributed systems. LOCUS takes this approach.

It should be noted that an integrated architecture can be easily made largely to look like a server oriented system if suitable configuration flexibility has been provided. For example, if there were significant cost or performance advantages to be gained by having most files stored at a few sites, no software changes would be necessary to LOCUS. In fact, one of the planned configurations for LOCUS includes workstation-like machines with no local file storage at all.

4.3. Lightweight Processes

To maximize LOCUS performance, we wished at any given site to be able to serve a network request without the usual overhead of scheduling and invoking an application level process. Hence, implementing much of LOCUS as user processes was not acceptable. Further, the services that a given machine must provide to remote requesters is quite stylized in LOCUS; they correspond to the specific message types in the low level protocols. Hence implementation in the nucleus of LOCUS was indicated. We took this view even though it could have been quite convenient to implement network service in user processes.

The solution chosen was to build a fast but limited process facility called *server processes.* These are processes which have no non-privileged address space at all. All their code and stack are resident in the operating system nucleus. Any of their globals are part of the operating system nucleus' permanent storage. Server processes also can call internal system subroutines directly. As a result of this organization, the body of server processes is quite small; the total amount of code to process all message types is several hundred lines of C code (some of which is subroutine calls to already existing system functions). As network requests arrive, they are placed in a system queue, and when a server process finishes an operation, it looks in the queue for more work to do. Each server process serially serves a request. The system is statically configured with some number of these processes at system initialization time.

These lightweight processes permit efficient serving of network requests (therefore keeping protocol support cost low) while at the same time avoiding implementation of another structuring facility besides processes. In retrospect, this was an excellent early design decision, both because of the structuring simplicity which resulted, and because of the contribution to performance.

4.4. Local Operation

The site at which the file access is made, the *Using Site* (US), may or may not be the same as the CSS or SS. In fact, any combination of these roles are possible, or all may be played by different sites.

When multiple roles are being played by a single site, it is important to avoid much of the mechanism needed to support full, distributed operation. These optimizations are supported in LOCUS. For example if, for a given file *open,* CSS=SS=US, then this fact is detected immediately and virtually all the network support overhead is avoided. The cost of this approach is some additional complexity in protocols and system nucleus code.

The system design is intended to support machines of heterogeneous power interacting in an efficient way; large mainframes and small personal computers sharing a replicated file system, for example. Therefore, when a file is updated, it is not desirable for the requesting site to wait until copies of the update have propagated to all sites storing copies of the file, even if a commit operation is desired. The design choice made in LOCUS is for updated pages to be posted, as they are produced, at the storage site providing the service. When the file is closed, the disk image at the storage site is updated and the using site program now continues. Copies of the file are propagated to all other storage sites for that file in a demand paging manner.[1] In this way, it is straightforward for a system with no local file storage to participate in a LOCUS network.

4.5. Performance Results

To evaluate the performance of a network transparent system such as LOCUS satisfactorily, one would like answers to at least the following questions. First, when all the resources involved in an operation are local, what is the cost of that operation, especially compared to the corresponding system that does not provide for network operation, either in its design or implementation? This is often a difficult comparison to make. It is not sufficient just to "turn off" the network support code in LOCUS, as the existence of that code altered the very structure of the system in ways that wouldn't happen otherwise. Fortunately, in our case, we could compare local operation under LOCUS with the corresponding situation under standard Unix.

The performance of remote behavior is also of paramount importance, since after all, it is distributed operation that largely motivated the LOCUS architecture.

Three initial experiments are reported here:
- i) reading one file page,
- ii) writing one file page, and
- iii) sequentially reading a 600 block file.

Each experiment was run on:
- a) standard Version 7 Unix,
- b) local LOCUS (i.e. the program and data both resided at the same machine),
- c) distributed LOCUS (i.e. the data resided on the machine used in cases a and b, but the program ran on another machine.)

For the first two experiments, the quantity reported is system time - the amount of cpu time consumed in servicing the request. For distributed LOCUS, the quantity given is the *sum* of the system time at the using site and the storage site. In the third experiment, total *elapsed* time is reported.

A great deal of care was taken to assure comparability of results; the same data blocks were read, numbers of buffers were comparable, the same disk was employed, etc. The tests were repeated multiple times. The values reported are

[1]While this method avoids useless waiting at the using site, it does raise the possibility that a new copy of the file is successfully created and then immediately made unavailable by a crash or partition. In fact, this is just one of the many race conditions which can occur. They are all handled by the version management algorithm described in [Parker80].

means, but the standard deviations were very small. The machines on which these measurements were run are DEC PDP-11/45s, with an average time of greater than 2 microseconds per instruction. The disk used is capable of delivering one data page per 15 milliseconds if there is no head movement. The network speed at the time of the measurements was 1 megabit/second.

System Time Measurements
(one data-page access - results in milliseconds)

System	Read	Write
Unix	6.1	4.3
Local LOCUS	6.2	4.3
Distributed LOCUS[1]	6.3+8.0=14.3	4.3+3.2=7.5

The sequential activity results are as follows.

Elapsed Time Measurements
(600 block sequential read)

System	Time (seconds)	Milliseconds/page
Unix	9.40	15.6
Local LOCUS	9.15	15.25
Distributed LOCUS	10.31	17.18

4.5.1. Interpretation of Results

In our view, these results strongly support the argument for network transparency. Local operation for the cases measured is clearly comparable to a system without such a facility. There is more system overhead for remote operation, but this is not surprising. Two network interface devices are involved in every remote access in addition to the one storage device. For a read, there are two messages involved, the request and the data-reply; hence a total of four additional I/Os must occur beyond the single disk I/O that was always present. That additional cost is about 4000 instructions in LOCUS (8 milliseconds, 2 microseconds/instructions). In the absence of special hardware or microcode to support network I/O, we consider this result quite reasonable.

It is especially encouraging that for operations which can take advantage of the involvement of multiple processors, the speed of remote transparent access is indeed comparable to local access. Sequential processing is one such example, since prefetching of data, both from the disk and across the network, is supported in LOCUS. Here, access in all cases is running at the maximum limit of the storage medium. No significant delay is imposed by the network transparent system.

Most important however, one should compare the performance of a distributed system such as LOCUS, where the network support is integrated deep into the software system architecture, with alternate ways of gaining access to remote resources. The traditional means of layering software on top of centralized systems leads to dramatically greater overhead. Before the development of LOCUS, Arpanet protocols were run on on the same network hardware connecting the PDP-11s. Throughput was not even within an order of magnitude of the results reported here.

[1]The total read and write time given are the sum of the system time used at the using site and the storage site.

Several caveats are in order however as these results are examined. First, these measurements were made before replicated storage was implemented. Hence no conclusions in that respect can be made. Second, because of the lack of available network interconnection hardware, only a small network configuration is available for study, and so only light loading occurs. On the other hand, since most computer utility experience shows limited concurrent sharing, we expect little synchronization interference when the network is significantly larger and more users are supported.

LOCUS also requires more memory to operate than standard Unix for comparable performance. This fact occurs for two principal reasons. First, there is more code in the LOCUS nucleus than in the Unix kernel, in part because of low level network support and synchronization mechanisms, but also because of more sophisticated buffer management, and because of the server process code. Second, LOCUS data structures are generally larger than the corresponding structures in Unix (to keep track to network-related information). For the configuration at UCLA, Unix requires 41K bytes code and 44K bytes data. The equivalent LOCUS system (with the same number of buffers, etc.) would require 65K bytes code and 51K bytes data.

LOCUS also contains considerable additional code to handle recovery, but this code is run as a conventional user process, and therefore is not locked down. It is also rarely involved.

5. Current State and Future Plans

A LOCUS prototype that supports network transparency for the file system is operational. Remote process creation and cooperation of processes on different machines was being implemented when this paper was written. Transparent access to most remote devices is operational. Replicated storage of files is now supported, although the currently operational recovery mechanism is simplistic; the methods discussed earlier, while coded, are not yet integrated into the rest of the LOCUS system. Finishing these various portions of the system, followed by additional testing and performance tuning, are obvious next steps.

In addition to the PDP-11 version of LOCUS, a LOCUS prototype for the VAX 780/750 is also operational. One of the goals of this porting effort was to investigate the effect of having different hardware on the network.

In designing remote process support, it was concluded that the Unix model of process interaction is not quite suited for distributed systems. In Unix, two processes in the same family can operate in a very intimate fashion; they may share the same current pointer into an open file for example, so that when one process reads one character, the next read call by the other process gets the following character. To support so intimate and unstructured a shared state can be expensive without shared memory. It also seems unnecessary. Nevertheless, our first version of network process support is fully compatible with the strict Unix model.

Beyond these issues, there are two major sets of plans. First, to best demonstrate the value of network transparency, one would wish to build a distributed application on top of LOCUS, and show that doing so was considerably easier than using a collection of machines running conventional operating systems, and that the result operated satisfactorily. Data management is one planned application. To support it properly, a few changes to LOCUS will be necessary, including more powerful user available synchronization primitives, a way to modify portions of a file without the entire file being transmitted when replicated copies need to be updated, and support for multimachine, multifile commit.

Second, we wish to be able to make stronger statements about reliable and available operation. While network transparency provides considerable opportunity for smooth and graceful resource substitution, more advantage of this fact needs to be taken. For example, modest extension of LOCUS would support the ability to snapshot a process and move that snapshot to another machine in preparation for continued running if the original computation site should fail. Synchronization of that snapshot with open files and other resources can be done with the multiobject commit mechanism. Additional work in detecting software conditions which would suggest reconfiguration or otherwise altered operation would be profitable. Lastly, we cannot yet make a quantitative statement about the level of reliability and availability actually achieved.[1]

6. Conclusions

The two most significant specific conclusions we draw from our LOCUS experience are:

1. High performance network transparency in a local network is feasible.
2. Network transparency in a local network possesses so many advantages that a choice not to adopt it ought to be very carefully justified.

We have not gained enough experience with the effects of the integrated vs. server model to make a strong statement. Nothing we have seen so far shakes our confidence in the integrated model, however.

In general, our experience suggests that the system architectures suitable for local networks differ markedly from those for significantly lower bandwidth, higher delay environments. Remaining with the more traditional structures misses significant opportunities.

7. Bibliography

Alsberg, P. A., Day, J. D., *A Principle for Resilient Sharing of Distributed Resources,* Second International Conference on Software Engineering, San Francisco, California, October 13-15, 1976, pp. 562-570.

Bernstein, P., *Algorithms for Concurrency Control in Distributed Database Systems,* Technical Report CCA-80-05, Computer Corporation of America, February 1980.

Cheatham, T., Private communication 1971.

Clark, D., K. Pogran, and D. Reed, *An Introduction to Local Area Networks,* Proceedings of the IEEE, Vol. 66, No. 11, November, 1978, pp. 1497-1517.

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

[1]Empirical observations can be made. However, there is not yet enough experience with heavy loading of LOCUS in a non-toy network configuration to make meaningful statements.

Gray, J., *Notes on Database Operating Systems,* in *Operating Systems: An Advanced Course,* Vol 60 of *Lecture Notes in Computer Science,* Springer-Verlag, (1978), pp. 393-481.

Kleinrock, L., Opderbeck, H., *Throughput in the Arpanet - Protocols and Measurement* Fourth Data Communications Symposium, Quebec City, Canada, October 7-9 , 1975, pp. 6-1 to 6-11.

Lampson, B., and H. Sturgis, *Crash Recovery in a Distributed Data Storage System* working paper, Xerox PARC, Nov 1976.

Lampson, B., Private Communication, 1979.

Mockapetris, P. V., Lyle, M.R., Farber, D.J., *On the Design of Local Network Interfaces,* Proceedings of IFIP Congress '77, Toronto, August 8-12, 1977, pp. 427-430.

Parker, S., G. Popek, et. al., *Detection of Mutual Inconsistency of Distributed Systems,* accepted for publication in IEEE Transactions on Software Engineering.

Rashid, R., and P. Hibbard, *Research into loosely-coupled Distributed Systems at CMU,* Notes from IEEE Workshop on Fundamental Issues in Distributed Systems, Pala Mesa, Ca., Dec 15-17, 1980.

Ritchie, D., "The Unix Time-Sharing System," *Communications of the ACM,* Vol. 17, No. 7, July 1974, pp. 365-375.

Rudisin, G. *Reliability and Recovery Methods for Partitioned, Distributed File Systems,* Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, 1982 (forthcoming).

Saltzer, J., D. Reed, and D. Clark, *End-to-End Arguments in System Design,* Notes from IEEE Workshop on Fundamental Issues in Distributed Systems, Pala Mesa, Ca., Dec 15-17, 1980.

Swinehart, D., G. McDaniel, D. Boggs, *WFS: A Simple Shared File System for a Distributed Environment* Proceedings of the Seventh Symposium on Operating Systems Principles, Dec 10-12, 1979, Asilomar, Ca. pp 9-17.

Walker, B. J., *Issues of Network Transparency and File Replication in Distributed Systems: LOCUS,* Ph.D. Dissertation, Computer Science Department, University of Califor-nina, Los Angeles, 1982 (forthcoming).

Wilkes, M., and R. Needham, *The Cambridge Model Distributed System,* Notes from IEEE Workshop on Fundamental Issues in Distributed Systems, Pala Mesa, Ca., Dec 15-17, 1980.

Xerox, *The Ethernet: A Local Area Network - Data Link Layer and Physical Layer Specifications, Version 1.0,* Sept 30, 1980. Available from Digital Equipment Corporation, Maynard, Massachusetts; Intel Corporation, Santa Clara, California; Xerox Corporation, Stamford, Connecticut.