Network Tasking in the Locus Distributed Unix System

David A. Butterfield and Gerald J. Popek

Locus Computing Corporation
3330 Ocean Park Blvd., Santa Monica, CA 90405

ABSTRACT

Locus is a distributed Unix operating system which provides a high degree of network transparency while maintaining excellent performance characteristics. The system executes on the Digital Equipment Corporation VAX, the Motorola 68000, and other processors. Users and programs have the illusion that they are running on a single computer system, even though their file resources may be distributed around the network and cooperating processes may be on different machines, or may even move during execution. In this paper, network tasking in general and the tasking aspects of Locus in particular are discussed.

1 The Locus Operating System

The Locus operating system is a distributed version of Unix, with extensive additional features to aid in distributed operation and support for high reliability and availability behavior. Many of these facilities are of substantial value even when Locus is used on single, standalone computers. Locus provides a surprising amount of functionality in a reasonably small package. The system, while somewhat larger than standard Unix, is significantly smaller than other extended Unix systems.

The system makes a collection of computers, whether they are workstations or mainframes, as easy to use as a single computer, by providing so significant a set of supports for the underlying network that it is virtually entirely invisible to users and applications programs. This *network transparency* dramatically reduces the cost of developing and maintaining software, and considerably improves the user model of the system. It also permits a great deal of system configuration flexibility, including diskless workstations, full duplex I/O to large mainframes, transparently shared peripherals, and incremental growth over a large range of configurations (from one workstation to a large network including mainframes) with *no* effect on applications software required to take advantage of the altered configuration. This transparency is supported even in a heterogeneous network of various cpu types. For example, if a user or program running on one machine type in a Locus network attempts to execute a program for which the only available version runs on a different machine type, Locus will automatically cause that execution to occur on the appropriate machine, completely transparently to the caller.

Locus has been in operational use for over two years on DEC VAX-11/750 computers. Systems containing as many as 17 nodes and three cpu types have been operated. Nodes in a Locus network may communicate over a variety of media; standard ten megabit Ethernet is common.

This paper first briefly reviews the major characteristics of Locus, and then concentrates on the remote tasking aspects of distributed operation. Remote process handling is described both philosophically and concretely as implemented in Locus. The Locus user model is presented, and some of the difficulties encountered in implementing it are mentioned. We conclude with comments on the feasibility of implementing the network process model and experiences with actual applications using the Locus implementation.

1.1 Transparent Distributed Operation

The power of network transparency and compatibility with an existing operating system environment can hardly be overestimated. It is commonplace to observe that distributed computing is very important, both in the near and longer term future. It is less well understood that unless a wide collection of distributed system application software becomes rapidly available, the growth of distributed computing will be severely retarded.

The Locus philosophy of very extensive transparency, together with Unix compatibility, even in distributed mode, means that all programs which run on a single machine Unix system will operate, without change, with their resources distributed. That is, one can take a standard application program, set a few configuration parameters, and cause it to execute with parts of the program running on one machine, parts on other machines, and both the data and devices which it accesses distributed throughout the Locus network, all without change to the application. This ability means that there is immediately available a very large collection of distributed application software, the necessary prerequisite to commercially effective distributed computing. Locus also includes facilities to control where data is located and which users and machines can access data, peripherals, cpus, and logical resources.

1.2 High Reliability

Reliability and availability are key to applications envisioned for Locus for two general reasons. First, many of the applications themselves demand a high level of reliability and availability. For example, the effect of a machine failure for an office system user is the equivalent of randomly locking that user out of his office at a random point for an unknown duration, an unacceptable situation. Database requirements for reliable operation are well known. Second, the distributed environment presents serious new sources of failures, and the mechanisms needed to cope with them are more difficult to construct than in single machine environments. To require each application to solve these problems is a considerable burden. Locus provides advanced facilities for both the local and distributed environment which operate transparently. These facilities include a file system which guarantees that, even in the face of all manner of failures, each file will be re-

tained in a consistent state, either completely updated or unchanged, even if alterations were in progress at the time of failure. This *commit* mechanism involves no additional I/O and so performs very well.

One of the important reliability and availability features of Locus is its support for automatic replication of stored data, with the degree of replication dynamically under user control. Loss of a copy of a replicated file does not affect continued operation, and Locus assures that the old version will be automatically brought up to date when it becomes available. A general transaction mechanism is provided that includes such advanced features as support for nested transactions, even when the transaction itself is distributed. Measurements demonstrate that this facility performs very well. Redundancy checks are included to avoid propagation of errors.

1.3 Unix Compatibility

Locus has been derived from Unix by successive modification of its internals and considerable extension. For virtually all applications code, the Locus system can provide complete compatibility, at the object code level, with both Berkeley Unix and System V, and significant facilities are present to provide high degrees of compatibility with other versions of Unix as well. As a result, one routinely takes unrecompiled load modules from other Unix systems and runs them unchanged on Locus. The process of converting an existing Unix system to Locus is designed to be a largely automated step of removing the Unix kernel and associated software, inserting the Locus kernel with its supporting system software, and reformatting secondary storage to add the reliability facilities and functional enhancements.

1.4 Heterogeneous Machines and Systems

Locus contains a set of facilities that permits different cpu types to be connected in a single Locus system. A high degree of transparency is supported across the various machines. For example, attempts to execute programs prepared for different cpus will automatically cause execution on the appropriate target machine, all transparently to the calling program or user. Such sophisticated transparency is accomplished in a manner that does not add complexity to the user view of the system.

2 Introduction to Network Tasking

There were two very strong considerations in choosing a user model of network tasking for Locus. One of these considerations was that Locus is a Unix system, and as such it should be compatible with other Unix systems. This means that portable C programs from other Unix systems should run unmodified in the Locus network environment. It was considered desirable for these programs to be able to take some advantage of the network, even if they were naive about its existence. It was also considered desirable that it be easy to add minimal code to an existing program which would allow it to make extensive use of the network.

The other major consideration was the overall Locus philosophy of *network transparency*, which requires that processes operate and interact with files and other processes across the network in the same way and with the same effect as locally. Adhering to this philosophy allows the user model to be considerably simpler than it might otherwise be, makes development of distributed algorithms easier, and allows graceful reconfiguration or degradation of distributed groups of cooperating processes when some nodes of the network are unavailable. (References to more complete discussions of various aspects of network transparency may be found in the Bibliography.)

Thus, the major considerations when choosing a network model were that it fit well into the Unix model, providing compatibility, and that it allow local and remote processes to be manipulated in the same ways, providing network transparency. Other important desirable characteristics were ease of user control, high performance, and retention of local autonomy of individual nodes in the network.

2.1 Network Tasking in Locus

The major Unix process control functions *fork* and *exec* have been extended in an upward compatible way. Recall that *fork* creates a new process, running the same program image as the caller, and that *exec* replaces the code and data of a running process with a new program and data image. In Locus, both of these calls have been extended for the network. For compatibility, the system calls look the same as in Unix, but under certain circumstances they operate over the network. *Fork* can cause a new process to be created locally or remotely. *Exec* has been extended to allow a process to migrate to another site as it replaces its image.

The decision about where the new process or new image will execute is made by examining a list of preferred execution sites specified by information associated with the calling process. This information is inherited by child processes when a process forks, and can be set dynamically by a *setxsites* system call. By issuing *setxsites*, a running program can specify where its subprocesses and future images will execute. An interactive user can set the site of execution of subsequent programs by giving a command to his shell, which will issue the *setxsites* call.

Use of an additional system call to set execution site information was chosen rather than adding arguments to the process calls for two reasons. First, by not changing the existing system call interfaces, programs written without knowledge of the network continue to work, and in fact may have their execution sites selected by their parents. This aspect is especially valuable when source code is not available. Second, it was considered desirable to separate those functions which affect the semantics of a program from those functions which are performance optimizations. With few exceptions, the site where a process runs does not have any effect on the results that the process computes.

A new system call, *migrate*, has been added to permit a process to change its site of execution while in the midst of execution. A long-running process might choose to do this, for example, if it received a signal telling it that the current site was about to be taken down.

Another application for process migration is load leveling. Process migration can be induced externally by sending a new signal whose default action is to migrate the process. If the user code of the process does not catch or ignore the signal, then the process will move to a new site. Using this mechanism, a load leveling daemon can monitor loads on various machines on the network and cause long-running cpu-intensive processes to move to less loaded sites.

2.2 Local Autonomy of Sites

In some Locus networks, it may be desirable to restrict the use of various machines in the network for use by only certain users. For instance, some machine may be reserved for only a particular group. The Locus network tasking protocols are designed so that each site in the network makes the final decision about whether to accept a process which is attempting to migrate to that site, and whether to create a process when a fork to that site has been requested. The site can implement whatever permission checking it deems necessary. In this way, the local autonomy of the site is preserved with respect to cpu resources expended upon user processes.

In addition to the Unix limitation on the number of processes a user may have on a site, Locus associates a site permission mask with each process (inherited by children) which specifies to which sites the process may perform network process creation or migration operations. This permission mask is set at signon time by *login*, on an individual user basis.

2.3 Integrated Fork/Exec Call

In typical Unix programs which create subprocesses, very often the *fork* call is quickly followed by an *exec* call. In Locus another new system call, *run*, has been added. *Run* is intended to achieve the effect of a combination of *fork* and *exec*. *Run* creates a new process and invokes a new program image in it before returning to the user code of the child process. In this way, the costly overhead of the *fork* system call can be avoided, without sacrificing machine independence.

To be useful in most situations, *run* takes as an argument a structure specifying certain operations to be performed by the kernel in the child process before the new program image begins execution. The permitted operations are those found between the *fork* and the *exec* in typical Unix programs. These include operations which close or duplicate file descriptors and change signal actions.

Run can create the child process locally or remotely, just as *fork*, and since it invokes a new image, may operate across different cpu types.

2.4 Local File Names

There are a few cases where it is useful, by convention, to associate certain files with a given site in the network. In Locus, it is common for there to be a directory for each machine in the global tree structure, e.g. */mach_i*, */mach_j*, and so on. One may even wish to make these directories local to their associated machine by making them mounted file systems. Such directories can, if the system administrator chooses, be used in Locus to hold the following types of items:

a. Per-site utility files: these include boot command sequences, local accounting data, and so on;

b. Temporary scratch storage: the files typically put in /tmp;

Thus, for example, the globally transparent name for the temporary disk storage file *save* on site *payroll* would be */payroll/tmp/save.*

However, in Unix, by convention, the names used for the few purposes mentioned above do not include a machine_name directory as part of the path name. Therefore, Locus includes an *alias* mechanism in the file system to map these old names to the new places in the name tree. Each process has associated with it a context variable which contains *machine_name.* Under normal conditions, if a process issues one of the old names, say /etc/utmp, then the name which would actually be referenced might be /machine_name/etc/utmp.

There are very few file names handled in this way; typically a dozen or so, primarily only system utility related items, of little or no consequence to most users.

The context variable is initialized to be the name of the machine where the user logs in. Its value is inherited when a parent creates a child process, and is unaltered by migrating a process. Thus, members of a process family, even if distributed among a number of machines, experience the same environment, and process migration need not alter the effect of a program's execution.

Through this alias mechanism, Locus accommodates the few cases in Unix where conventions embodied in a small number of old, existing programs, together with performance considerations, mandate that a few names not be interpreted in a global, transparent way. Since all resources can still be naturally accessed by their basic, globally transparent names, it is hoped that most of these few vestiges of single computer operation will eventually atrophy.

2.5 Heterogeneous Locus Networks

In Locus networks with more than one cpu type, it is clear that *fork* and *migrate* can be done only between matching cpu types, since the process retains substantial internal state information, such as registers, stack frames and so forth, not to mention the instruction stream. The source and destination site cpu types of an *exec*, by contrast, may differ, because most of the machine dependent information is reset.

When a load module is selected for execution, it is examined to determine on what cpu type it will run. The execution site list for the process is searched until a site of the appropriate type is found, or an indicator is reached that the system should simply choose an appropriate site.

Networks with heterogeneous cpus present a special problem for storing of load modules. Since the naming hierarchy is globally known throughout the network, there can exist only one object called, for example, */bin/cat*. In a homogeneous network of VAXen, this file would contain a VAX load module for the Unix *cat* program. In a homogeneous network of 68000s, it would contain a 68000 load module for the Unix *cat* program. In a heterogeneous network containing both VAX and 68000 processors, the obvious problem arises.

One might say that there is no problem here at all, that whichever program is placed into the file, *cat* will still work from any site in the network. Suppose that */bin/cat* contains a VAX load module. In this case, if the file is executed from a 68000, the system will notice that the file cannot run locally, and will find a VAX in the network to run the program.

Unfortunately, this would require that there always be a VAX up and available in the network, in order for *cat* to run. Further, this requires that every execution of *cat* from a 68000 be performed remotely, rather than locally.

The Locus solution is a *hidden directory*. A *hidden directory* is a special directory used for load modules; one load module for each machine type in the network is placed into the hidden directory, and when an attempt is made to execute the directory, the system automatically selects one of the load module files in the directory to run. In the example above, */bin/cat* would be a hidden directory containing two files: */bin/cat/vax*, and */bin/cat/68000*. *Exec*ing */bin/cat* would select one of the load modules from within the directory and execute it.

The directories are termed *hidden* because opening the directory to read it (e.g. by executing *size /bin/cat*) also selects the appropriate component from the directory and opens that file. Thus, the *hidden directories* appear to be simple files to most application programs. This is compatible with Unix systems.

A process has associated with it a list of names which are used to choose the components from hidden directories. This list is inherited by children from their parents, and may be set explicitly. It is initially set to select components for the type of site on which the user logged in.

Note again that hidden directories are needed only in heterogeneous Locus networks, to collect together the load modules for the various machine types in the network. In homogeneous networks, there is only one load module, so it may simply be installed as a file.

2.6 Network Tasking Applications

One application of remote tasking is load leveling. As was noted in a previous section, load leveling can be performed automatically by a daemon process which looks for cpu bound processes and unloaded sites, and asks the processes to migrate there.

A simpler form of load leveling may be performed by the shell. If the user has requested this option, the shell can keep track of the load on various sites and automatically execute the user's commands on the least loaded site. The mechanisms required to implement load leveling were almost trivial to construct under Locus.

Another application of remote tasking is the distribution of programs which have inherent parallelism. The *make* program is an example. *Make* has been modified so that multiple machines may simultaneously participate in making a final target which requires several intermediate targets. With two machines participating, the compile time of a major program is cut nearly in half. The changes required to *make* were quite simple.

2.7 Implementation Challenges

Implementing transparent remote tasking presented several difficulties. A major difficulty was *process tracking.* It is sometimes necessary to find a process in the network to deliver a signal or to inform it that its child or parent has died. To do this without having to search the entire network requires that the system keep track of processes when they migrate about. The process tracking code must be extremely careful when processes are moving around while delivery of a signal is being attempted, so that the signal is delivered reliably.

Another major effort was the code which implements file sharing among related processes on different sites. Unix file sharing semantics require inherited file descriptors to share read/write pointers among the processes involved. Locus implements these semantics across the network; the most common cases execute as fast with the processes distributed as when they are collected on a single site.

2.8 Network Tasking Experience

Our experience with transparent tasking has been an unqualified success, from the user's point of view. As illustrated above, the job of building software to execute in a distributed environment has been substantially eased, and the simplification of the user interface has been found to be quite valuable. As a further example, some time ago, a demonstration of Locus on a 68000 workstation connected to a larger VAX Locus network was being given. The demonstrator edited a program source file for a given language, and then compiled and ran it, piping the results to another program running on his workstation. Only afterward did he realize that there was no compiler for the language on the workstation; the compilation and subsequent program execution took place, transparently, elsewhere, on a VAX.

However, from the Locus implementors' point of view, the design and development of transparent tasking required considerably more effort than had been originally contemplated. Of course, the existence of a very high degree of transparency in the basic Locus system was an essential prerequisite. Without it, transparent tasking is obviously not feasible. Nevertheless, transparency still had to be extended to process relevant issues, such as shared file descriptors, signals, and so on. Also, given how much had already been accomplished, the opportunity to further extend the Locus transparency mechanism to heterogeneous cpus was especially attractive. Although heterogeneity presents significant additional problems, and complete solutions are not feasible, nevertheless a great deal can be accomplished, and the result so obviously frees the user from so much unnecessary complexity in his environment that the cost of development has clearly been justified.

3 Conclusions

The Locus network tasking implementation permits one to execute programs at any site in the network, subject to permission control, in a manner just as easy as executing the programs locally. One can dynamically, even just before process invocation, select the execution site. No rebinding or any other action is required. The mechanism is entirely transparent, so that existing software can be executed either locally or remotely, with no change to that software. This facility makes execution of a program remotely, as well as the construction and execution of distributed programs, quite straightforward.

Locus tasking greatly simplifies the development of software for a distributed environment. Load leveling is simple to accomplish. Automatic selection of the site of execution as a function of needed resources, including cpu characteristics, is also done automatically. Execution of programs with inherent parallelism, such as *make*, can be done on multiple machines simultaneously with little change to the original, single machine program.

Now that a highly transparent distributed computing environment is available for everyday use, those of us who have become accustomed to it take it for granted. Those who still remember working in a conventional networking environment wonder how they ever got along without transparency, and generally can't conceive of going back, any more than they would willingly give up interactive computing, screen editors, or virtual memory.

4 Bibliography

The Locus Distributed System Architecture, Locus Computing Corporation, Santa Monica, California, 1983.

Popek, Gerald J. and Walker, Bruce J. *Network Transparency and its Limits in a Distributed Operating System*, UCLA Computer Science Department Technical Report CSD840228, Los Angeles, California, 1984.

Walker, Bruce J. *Issues of Network Transparency and file replication in the Distributed Filesystem Component of Locus*, UCLA Computer Science Department Technical Report CSD830905, Los Angeles, California, 1983.

Walker, Bruce J. *The Locus Distributed Computing Environment*, 1984 IEEE Aerospace Applications Conference Digest, New York, New York, 1984.