UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems

Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle

Technical Report CSRI-257
April 1992

(To appear in *Software — Practice and Experience*)

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems

Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle
Computer Systems Research Institute
University of Toronto

Keywords: load sharing, load balancing, load index, remote execution, parallel computing, distributed computer systems, heterogeneous computer systems.

Abstract

Load sharing in large, heterogeneous distributed systems allows users to access vast amount of computing resources scattered around the system and may provide substantial performance improvements to applications. We discuss the design and implementation issues in UTOPIA, a load sharing facility specifically built for large and heterogeneous systems. The system has no restriction on the types of tasks that can be remotely executed, involves few application changes and no operating system change, supports a high degree transparency for remote task execution, and incurs low overhead. The algorithms for managing resource load information and task placement take advantage of the clustering nature of large-scale distributed systems; centralized algorithms are used within host clusters, and directed graph algorithms are used among the clusters to make UTOPIA scalable to thousands of hosts. Task placements in UTOPIA exploit the heterogeneous hosts and consider varying resource demands of the tasks. A range of mechanisms for remote execution is available in UTOPIA that provides varying degrees of transparency and efficiency.

A number of applications have been developed for UTOPIA, ranging from load sharing command interpreter, to parallel and distributed applications, to distributed batch facility. For example, an enhanced UNIX command interpreter allows arbitrary commands and user jobs to be executed remotely, and a parallel make facility achieves speedups of 15 or more by processing a collection of tasks in parallel on a number of hosts. Such performance is substantially better than that achieved using kernel-based process migration in experimental operating systems such as Sprite.
[1] [2]

[1]For correspondence, contact Songnian Zhou at CSRI, University of Toronto, 6 King's College Road, Toronto, Ontario, CANADA M5S 1A1;Tel.: (416) 978-3610; Email: zhou@white.toronto.edu.

[2]Pierre Delisle is currently with Sun Microsystems, Inc., 555 Boul. Dr. Fredrik Philips, Bureau 100, Saint-Laurent, QC, Canada,H4M 2X4.

1 Introduction

Distributed computing has been gaining importance over the last decade as a preferred mode of computing compared to centralized computing. It has been widely observed that usages of computing resources in a distributed environment is usually bursty over time and uneven among the hosts. A user of a workstation may not use the machine all the time, but may need more than it can provide while actively working. Some hosts may be heavily loaded, while others remain idle. Along with the dramatic decrease in hardware costs, resource demands of applications have been increasing steadily, and new, resource–intensive applications are being introduced rapidly. It is, and will remain to be, too expensive to dedicate a sufficient amount of computing resource to each and every user.

Load sharing is the process of redistributing the system workload among the hosts to improve performance and accessibility to remote resources. Intuitively, avoiding the situation of load imbalances and exploiting powerful hosts may lead to better job response times and resource utilization. Numerous studies on load sharing in the 1980s have confirmed such an intuition (see, example studies, [23, 16, 7, 14, 5, 32, 2, 31, 22]). Most of the existing work, however, has been confined to the environment of a small cluster of homogeneous hosts, and has focused on the sharing of the processing power (CPU). With the proliferation of distributed systems supporting medium to large organizations, system scale has grown from a few time–sharing hosts, to 10s of workstations supported by a few server machines, and to 100s and 1000s of hosts. For effective load sharing, computing resources besides processing power, such as memory frames, disk storage, and I/O bandwidth, should also be considered.

Another important development in distributed systems is heterogeneity. Heterogeneity may take a number of forms. There may be *configurational heterogeneity*, whereby hosts may have different processing power, memory space, disk storage, and so on. There may be *architectural heterogeneity*, which makes it impossible to execute the same code on different hosts. Finally, there may be *operating system heterogeneity*, thus the system facilities on different hosts vary and may be incompatible. Although heterogeneity imposes limitations on resource sharing, it also presents substantial opportunities. Firstly, even if both a local workstation and a remote, more powerful host are idle, the performance of a job may still be better if executed on the remote host, rather than on the local workstation. Secondly, by providing transparent resource locating and remote execution mechanisms, any job can be initiated from any host without considering where the resources needed by the task are, thus a job that can only be executed on a SUN host can now be initiated from an HP workstation.

The desirability of sharing load in a system with 100s or 1000s of hosts is not obvious. A performance study by Zhou [31] indicates that, in a homogeneous system running sequential jobs, little additional benefit due to load sharing is to be expected beyond a few tens of hosts. The two main motivations for sharing load in a large system are 1) configurational heterogeneity, and 2) parallel applications. Load sharing in large scale heterogeneous systems makes powerful hosts scattered around the system available to all the hosts. A parallel application may be able to make use of a large number of hosts.

There has been little research concerning issues related to large scale and heterogeneity in load sharing, yet they represent two of the most important research problems in load sharing in current and future distributed systems. As system scale increases by orders of magnitude and heterogeneity develops, existing algorithms for load sharing become inadequate, and new research issues emerge.

In this paper, we study the problems of and algorithms for load sharing in large, heterogeneous distributed computer systems by discussing the design and performance issues in UTOPIA, a load sharing system developed at the University of Toronto over the past several years. UTOPIA employs scalable scheduling algorithms for load sharing in systems running various kinds of Unix operating systems on multiple hardware platforms. The system has no restriction to the types of applications that can be remotely executed, and most applications are supported with no change. Computing resources in a distributed system are shared at two levels: First, monolithic applications can be transferred to remote hosts for execution. Second, applications can be divided into components and executed on multiple hosts simutalenously (i.e., parallel and distributed applications). Besides demonstrating the feasibility of a general purpose load sharing system for large heterogeneous distributed systems by building a system that is usable in diverse system and application environments, the two main contributions of our research to the field of resource sharing in distributed systems are 1) the algorithms for distributing load information in systems with 1000s of hosts and for making task placements based on tasks' resource demands and hosts' load information, and 2) the collection of remote execution mechanisms that are highly flexible and efficient, thus enabling interactive tasks that require a high degree of transparency, as well as relatively fine-grained tasks of parallel applications, to be executed remotely efficiently.

The rest of the paper is organized as follows. In Section 2, we discuss the design and organization of UTOPIA. The algorithms for load information dissemination and task placement are described in Section 3. The techniques used to achieve transparent remote execution in an Unix environment is described in Section 4. We discuss load sharing applications and their performance in Section 5. Relevant work is briefly surveyed and contrasted to our work in Section 6. Finally, we make a few concluding remarks in Section 7.

2 System Design and Structure

2.1 Desirable Properties

It is important to identify the desirable properties of a load sharing system before getting into a specific design. Below are some of the desirable (and often conflicting) properties of a load sharing system, which we used to guide our design of UTOPIA:

- *general purpose*: A load sharing system should make few assumptions about and have few restrictions to the types of applications that can be executed remotely. Interactive jobs, distributed and parallel applications, as well as non-interactive batch jobs, should all be supportable.

- *requiring little or no application change*: Applications evolve and new applications are being constantly developed. It is undesirable to have to modify the application programs in order to execute them remotely.

- *transparent*: The behavior and result of a task's execution should not be affected by the host(s) on which it executes. In particular, there should be no difference between local and remote execution. No user effort should be required in deciding where to execute a task or in initiating remote execution; a user should not even be aware of remote processing, except maybe better performance.
- *dynamic*: The algorithms employed to decide where to process a task should respond to load changes, and exploit the full extent of the resources available.
- *responsive to jobs' needs*: Different types of jobs have different resource requirements. Some may be CPU intensive and do not need much else, while others require a large amount of memory to perform well. Consequently, the best hosts for different jobs may well be different. To make intelligent job placement decisions, information about the resource demands of the jobs is needed.
- *efficient*: Load sharing should incur reasonably low overhead so that the benefits of sharing may be maintained.
- *scalable*: A load sharing system is scalable if it can make a large number of remote (powerful) hosts available to users throughout a large-scale system without substantially increasing the overhead.
- *supporting autonomy and protection*: While sharing is desirable, there may exist necessary restrictions to it. For instance, the conditions under which a host may be available for load sharing should be specifiable by the system administrator and/or its owner. Similarly, existing protection in the system should not be weakened by the provision of remote execution.
- *configurable and robust*: The size and organization of distributed systems vary greatly. To effectively exploit the resources in the system, a load sharing system should be configurable to suit the needs of a particular installation. The operation of the system should be simple, requiring little effort of the users and system administrators once it is properly set up.
- *easy to use*: The system should be easily usable to the three types of people involved in a load sharing system. It should be easy for the system managers to install and operate. It should enable application developers to build sophisticated applications with relative ease and without much knowledge of the system internals. Finally, users of the system should need to learn little to enjoy the benefits of load sharing; this is consistent with the transparency property.

In designing UTOPIA, substantial efforts were made to realize the above properties.

2.2 Design Decisions

A number of design decisions were made early on in the project. First, we decided *to implement* UTOPIA *entirely at the user level, transparent to both the operating system and most of the applications.* A layer of software is interposed between applications and the kernel that facilitates load sharing. To support heterogeneity, we want UTOPIA to be runable on a variety of system platforms. Changing their kernels would involve substantial efforts initially and with every new version of the systems. Similarly, it is also undesirable to require changes, or even just recompilation/relinking of applications for them to take advantage of load sharing. We expect a small number of applications to be modified to use UTOPIA via a high–level library interface. These applications may in turn provide remote execution support to arbitrary applications so that they can benefit from load sharing without change.

Our second design decision is *to support remote execution only at task initiation time*; no checkpointing or task migration is supported. Such a restriction makes it possible to allow arbitrary tasks to be executed remotely, and to produce an efficient implementation without changing the system kernel. For improving performance, initial task transfer may be sufficient; a modeling study by Eager, Lazowska, and Zahorjan suggests that dynamic task migration does not yield much further performance benefit except in some extreme cases [6]. The Condor distributed batch facility developed at the University of Wisconsin [22] requires that a workstation be completely idle in order to receive remote jobs, and, that once the owner of the workstation resumes activity on it, all remote tasks on it be migrated away. While such an approach maximizes host autonomy and ownership, it also makes remote execution more costly, leaves residual dependency on the originating host, and either requires kernel changes or restricts the types of jobs that can be executed remotely[3]. We view load sharing being the rule rather than the exception; a host is by default sharable. Our experience to date shows that, as long as overhead is reasonably low, load sharing is generally beneficial, even to a user whose workstation is being used by remote tasks, because while her workstation is busy, her tasks would be automatically transferred to other hosts. On the other hand, UTOPIA provides mechanisms to allow users and system administrators to set a host to be "send only" or "private", and to define saturation thresholds for its local resources so that it stops receiving remote tasks once load on a resource reaches its threshold. Priority mechanism can also be used to ensure services to local tasks.

Our third design decision is *to base* UTOPIA *on a shared file name space that is uniform throughout the system.* Consequently, the same sharable files can be accessed from any host using the same names. The benefits and feasibility of transparent file access in large distributed systems is evidenced by a number of existing large-scale distributed systems such as Andrew [15], Athena [4], and HCS [10]. Transparent file access not only provides accessibility to files from anywhere in the system, and facilitates the sharing of files among users, but also eases system management and administration. For those distributed file systems, such as NFS [26], that support a more general file naming structure, non-uniform file name space has been found to hinder accessibility and sharing. As distributed systems evolve, we believe

[3] In Condor, for instance, a remote job cannot create child processes, perform terminal I/O, or handle software signals.

that uniform file name space will become prevalent, even in very large distributed systems.

Load sharing in an environment without uniform file access would be restrictive and expensive since remotely executing tasks may reference files (using file names possibly embedded in their executables) that would either have to be transferred upon reference, or at least have to have their names translated on the fly to those usable on the execution host. Condor takes the former approach by intercepting all the UNIX file system calls and sending them back to a shadow process on the original host [22]. The shadow process performs the file operations (accessing file server hosts if necessary), and, for read, forwards data to the remote task. While this ensures transparent file access, the costs of the file operations and the volume of resulting network traffic are substantially increased, not to mention the increased dependency on the original host and software complexity. We feel that it is undesirable to impose such penalties in the common case in which the hosts involved share a uniform file space (supported by one or more file servers on the network). On the other hand, there are desirable exceptions to the uniform file space assumption. For instance, in a heterogeneous environment, a remote task should execute a version of its code for the execution host, rather than the one for the originating host. This is easily supported by initial task transfer since execution is started remotely, and the correct version of the code is used automatically. In contrast, dynamic task migration usually has to be restricted to be among homogeneous hosts.

2.3 Basic Architecture

The basic architecture of UTOPIA is shown in Figure 1. UTOPIA assumes a modular structure that clearly separates 1) the policies governing the exchange of load information and task placement decision making, 2) the mechanisms for transparent remote execution, 3) the interface supporting load sharing applications, and 4) the applications. We explain the four components below.

Load Information Manager (LIM). LIM is the policy module of UTOPIA. On each participating host runs an instance of LIM that exchanges load information with its peers on other hosts and gives advice to applications as to on which host(s) their task(s) should be executed. Multiple resources on each host, as well as the resource demands of each application, are considered in LIM's placement decisions. In addition, LIM provides load information to those applications that choose to make their own placement decisions. The scalability of UTOPIA is largely determined by the algorithms for load information exchange and task placement used by the LIMs, which we discuss in Section 3.

Remote Execution Server (RES). Another server on each host, RES, provides the mechanisms for transparent remote execution of arbitrary tasks. Before remote task initiation (typically after placement advice has been obtained from the LIM), a stream connection is established between the local application and its remote task, through the RES on the target host. UTOPIA supports several models of remote execution to meet the diverse functional and performance requirements of applications. The support for transparent remote execution in UTOPIA is presented in Section 4.

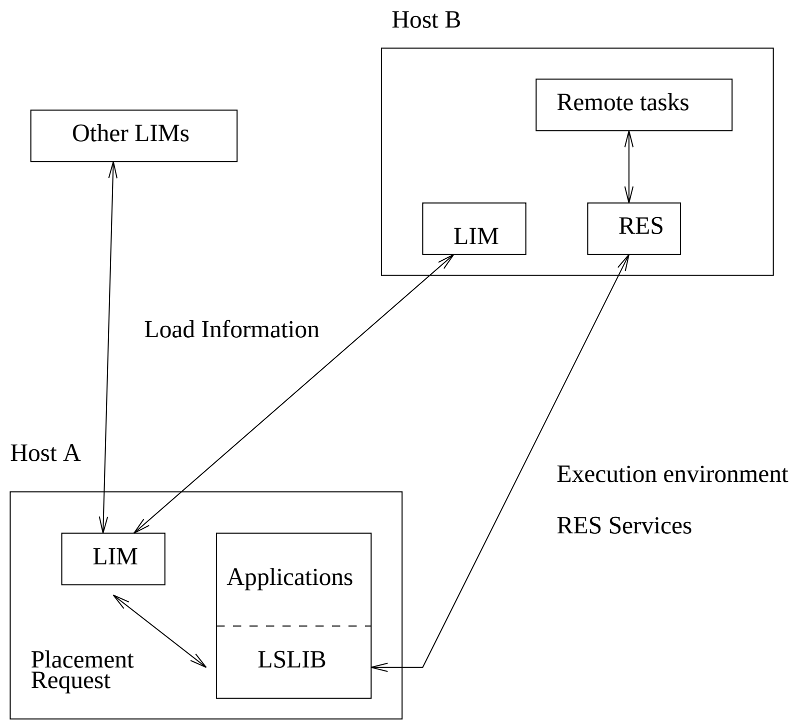


Figure 1: Basic structure of the UTOPIA system.

Load sharing library (LSLIB). A runtime library is provided for ease of developing load sharing applications. LSLIB implements a high–level procedural interface that allows applications to interact with LIM and RES.

Load sharing applications. Two types of applications are supported, those using the LSLIB interface directly, and those using such applications as their execution context. One distinguishing feature of UTOPIA is that, for the vast majority of applications that take advantage of load sharing, no modification to their source or binary is required. We discuss UTOPIA applications and interface support in Section 5.

The modular structure of UTOPIA offers at least two advantages. First, parts of the system can be used without others. For instance, a task can be executed at a remote host specified by the user. LSLIB would be used to contact the remote RES, but LIM would not be needed. Similarly, load information and placement advice from LIM may be obtained for purposes other than remote execution. It is also possible, though not usually desirable, for an application to interact with LIM and RES directly without using LSLIB. A second advantage of a modular structure for UTOPIA is that policies and mechanisms of load sharing may be changed independently of each other and applications.

3 Policies for Load Sharing

The task placement policy of UTOPIA supports two key functions needed by load sharing applications: locating resources needed by a task and choosing the best host(s) among all candidate hosts providing the required resources. The first function uses relatively static information about the configuration of the system. The second function requires dynamic information about the loading conditions of the resources in the participating hosts. We therefore discuss load indices and efficient and scalable methods for load information distribution before the task placement algorithms.

3.1 Load Indices

For each type of resource, a *load index* is defined that quantifies its loading condition. Depending on the nature of the resource, some possibilities are queue length, utilization, or the amount of free resource. Unix generally does not provide accurate measures of resource load, and our decision to stay at the user level precludes the possibility of generating ideal load indices by modifying the kernel, as is done by Ferrari and Zhou [11]. For each host, below are some of the resource load indices currently used by LIM:

- *CPU* : 15 second exponentially smoothed average CPU queue length based on periodically (typically at 5-second intervals) sampled CPU queue lengths,
- *memory* : the amount of free memory,
- *disk I/O* : the average disk transfer rate over all disks, and
- *login sessions* : the number of concurrent users.

In addition to the above, several other indices of host load are available in UTOPIA, such as 5- and 15-minute average CPU queue lengths, and the period of time a host has remained idle. They are typically used by certain applications to make their own task placement decisions.

For a heterogeneous environment, some of the indices may have to be adjusted to make them equitable. Our current UTOPIA implementation scales a host's CPU load index plus one (for the prospective remote task) according to its CPU speed, using the fastest host in the system as a base; hence, even if a slow host is idle, its CPU index would have a high value, thus discouraging remote task transfers to it. The above load indices are assembled into a record called a *load vector*.

3.2 Load Information Distribution

As we mentioned in the introduction, load sharing in large scale systems is desirable because of system heterogeneity and the needs of parallel applications. On the other hand, executing tasks, especially highly interactive ones such as CAD/CAM applications, on far away hosts may incur substantial communication overhead, or even aggravate congestion on backbone networks. The main difficulty of large scale load sharing is that the hosts desirable for sharing may be scattered, and their load information has to be made available to many other hosts; the overhead of information distribution may become intolerable.

The key technique in building scalable systems is *clustering*, by which hosts in the system are organized into clusters. This technique fits well with existing structure of large distributed systems, which are usually formed by clusters of hosts each supporting a group of (closely associated) users, or a workgroup. Hosts in the same cluster typically share resources extensively. Clusters may form a flat space, or be organized hierarchically. Our algorithms for load information distribution and task placement take advantage of this organization.

3.2.1 Load Information Distribution within a Cluster

There are a number of possibilities for making load information available for task placement in a *load sharing cluster*. A LIM, upon receiving a request for task placement, may query a few peers and select an acceptable host, as is proposed by Eager, Lazowska, and Zahorjan [5]. Alternatively, load information may be exchanged periodically among the LIMs in a load sharing cluster, as is proposed by Zhou [31], who shows that such an approach can result in little message overhead for a limited–scale environment and lower task delay, since placement may be made with readily available load information. Using Zhou's algorithms, placement decisions also tends to be optimal, as all of the available hosts are considered. We adopt two of Zhou's algorithms. For both algorithms, one of the LIMs is designated as the master, and periodically receives load vectors from all the other LIMs to be assembled into the *load matrix* [4]. In the GLOBAL algorithm, the load matrix is periodically broadcast to all LIMs where placements are made for locally originated tasks. In the CENTRAL algorithm, the load matrix is not broadcast, and all task placements in a cluster are performed by the master LIM. Compared to a fully distributed algorithm in which all hosts send their load vectors to all other hosts, the centralized approach in GLOBAL and CENTRAL results in much lower message overhead in all but the master LIM, whose overhead, if broadcast is used for load matrix distribution, is comparable to a host using a distributed algorithm [31].

3.2.2 Load Information Distribution among Clusters

GLOBAL or CENTRAL algorithm alone is clearly not suitable for a system with 100s or 1000s of hosts. The master LIM may easily become a bottleneck. It is highly unlikely that all the hosts in the system are needed for receiving tasks from far away (in terms of network distance and delay). To achieve optimal performance on a host, typically only other hosts in its local cluster and a select number of remote, powerful hosts, which we call *widely–sharable hosts* (be they ones with fast CPU, large memory, high I/O bandwidth, or special hardware/software), are needed. Using GLOBAL or CENTRAL in a large system would mean a great deal of "garbage" load information being widely distributed, which has to be sifted through at task placement time.

[4] An obvious optimization is to send the load vector to the master LIM only if there has been significant change in the values of the load indices. In practice, we found that this technique cut down message traffic by a factor of three or more (see Section 3.5).

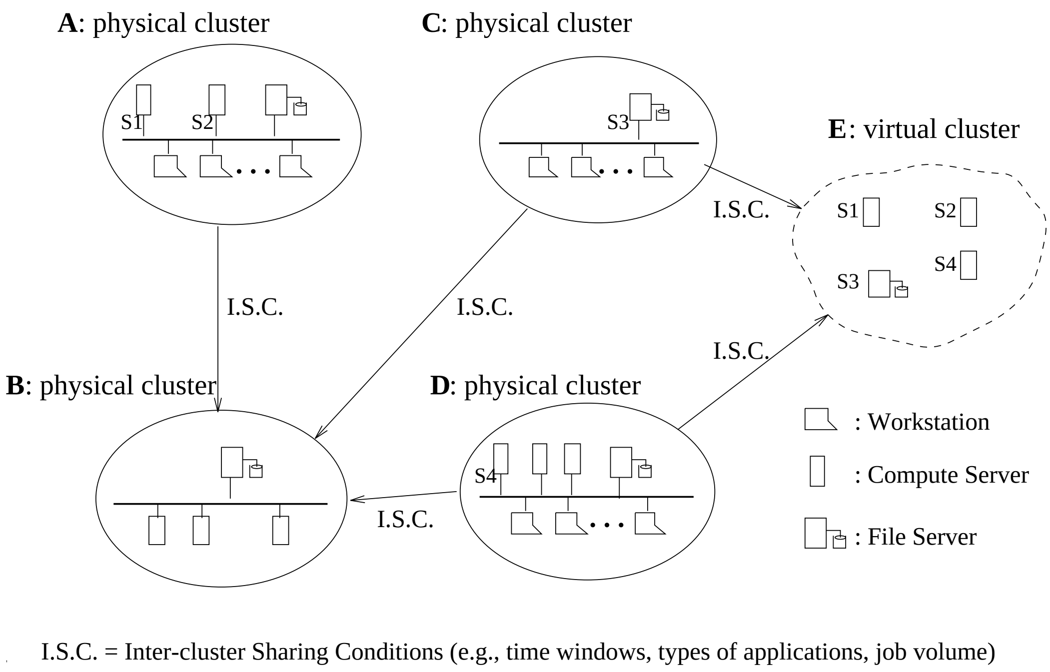


Figure 2: Directed graph among physical and virtual clusters for load information exchange.

Our solution to the scaling problem in load information distribution is based on two ideas: *directed graph among the clusters*, and *virtual clusters*. We treat load sharing clusters as nodes in a directed graph, with the edges indicating the directions of task flow for remote execution. The master LIMs of the clusters with incoming edges (called *target clusters*) send their load matrices "upstream" along the edges to the master LIMs of the clusters with tasks to send (called *source clusters*), which, if the GLOBAL algorithm is used, in turn distribute them to all hosts in their clusters along with their local load matrices. Task placement is still performed in the local cluster as before, and remote execution occurs directly between source and target hosts. For instance, in the example system shown in Figure 2, cluster B may receive tasks from Clusters A, C, and D. Such an arrangement is desirable because Cluster B is a compute server bank intended to be shared by other clusters.

In a less–than–ideal situation in which widely–sharable hosts are not concentrated in one or few clusters, but distributed over a large number of clusters, sharing them may result in a complete (or nearly–complete) directed graph such that all (or most) clusters send their load matrices to all (or most) other clusters, a clearly non–scalable situation. *Virtual clusters* are used to handle such situations. The scattered widely–sharable hosts are collected in one or more virtual clusters which behave similarly to a physical cluster: A virtual cluster has a master LIM which collects load vectors from all the other hosts in the cluster and distributes the resulting load matrix to the master LIMs of all the (physical) clusters wanting to send tasks to it (its source clusters). Cluster E in Figure 2 is one such example. Since a virtual cluster is formed solely for the purpose of collecting members' load information and making it available to interested physical clusters for placement, virtual clusters do not interact with each other, and the LIMs acting in a virtual cluster never receive placement requests. To participate in load sharing, each host in a system must belong to one physical cluster, and zero or more virtual clusters. The LIM on a host, if not a master for any of the clusters it belongs to, periodically sends its load vector to each of the master LIMs for which it is a slave. Virtual clusters make it possible to share load on widely dispersed hosts in a large scale system, without "information pollution" and undue overhead.

In summary, our load information distribution algorithm uses a combination of centralized techniques within clusters for their efficiency, and distributed, selective distribution techniques among the clusters for its scalability. The two techniques work especially well together, since the centralized technique within clusters makes it possible for the load matrices of the clusters to be distributed between their master LIMs, rather than having every host involved. Meaurement data of LIM overhead presented in Section 3.5 suggest that our load information distribution algorithms incur low overhead and may be scalable to 1000s of hosts.

3.3 Task Placement

Besides timely and comprehensive load information, specification of the resource requirements of the tasks is also necessary to make optimal task placement decisions. The resource requirements of tasks in a heterogeneous system can be classified into two types: general resource requirements and restrictive resource requirements.

General resource requirements are used to describe the usage demands on resources generally provided by every host, such as CPU, memory and I/O devices. While it is possible to quantify tasks' resource demands by trial executions and measurements, or by having the system collect statistics of their past executions [13], in the current UTOPIA implementation we take the simplified approach of qualitatively characterizing tasks by the resources they need the most. Thus, a simulation program may be purely CPU-bound, whereas a CAD tool may be both CPU and memory intensive. By matching the general resource requirements of a task with the load information about these resources, a better task placement decision can be made.

Restrictive resource requirements are used to describe tasks' need for specific kinds of resources that are not available on all hosts or are different on the hosts. This information is needed to locate the type of host for the correct execution of a task. Examples of restrictive resource requirements are architectures, operating systems and their versions, attached special devices such as disks and parallel processors, and the unique roles of some hosts, such as file servers and compute servers. For example, a compilation task may only be executed on hosts with a specific version of the desired operating system and architecture to produce the intended binary output.

In UTOPIA, resource requirement information is kept in a system–provided file containing a list of task names together with their resource requirements. (The load sharing clusters may share the same file, or each has its own for flexibility.) Usually only those tasks that consume a significant amount of resources or can only be executed on some particular types of hosts are placed on the list. Individual users may have their own lists as extensions to the system default list. The tasks on these lists are eligible for remote execution; hence, the lists are called *remote task lists.* The construction of the remote task lists is straight–forward with resource requirements being specified by expressions with meanings such as "CPU and I/O intensive, on VAX hosts with local disks" or "memory intensive, execution on any hosts except those running Unix System V". By putting locally unexecutable tasks in the remote task lists, these tasks can be invoked on a host as if they were all available on the local host.

Although UTOPIA has no restriction on remote execution, some tasks must be executed locally due to their semantics. The Unix command *ps*, which lists processes on the host, is one such example. For others, their resource needs may be so low that little performance gain is expected from remote execution, while the attendant overhead has to be incurred. Two modes of operation are provided in UTOPIA to an application for deciding whether a particular task is eligible for remote execution. In the *local mode*, only those tasks in the (system and user) remote lists are eligible. In the *remote mode*, another set of task lists, the *local task lists*, are consulted to determine whether a task has to be executed locally. If the task is not found in the local task lists, then it is considered eligible for remote execution, and the remote task lists are consulted for the resource requirements of the task to be used as input to the LIM for placement. If the task is not on any of the task lists, then some default resource requirement (currently CPU) is assumed.

In practice, both the local and the remote modes have been found to be useful. Task eligibility checking is performed by the applications library, LSLIB, prior to possibly contacting LIM for placement. Consequently, the mode of operation is selectable for each application, and may even be toggled dynamically.

One performance optimization is desirable for the task placement algorithm. Since remote execution incurs certain cost, and context has been set up on those hosts that an application has used before, such hosts should be given preferential consideration: Only if other hosts have significantly lower (scaled) load should they be used. Such a restriction is especially desirable in large systems in order to prevent a proliferation of connections from the applications to remote hosts. We call the set of hosts to which an application already has connections its *preferred host set.* This set may start empty, or with a few hosts in it if it is desirable to initiate connections with these hosts (asynchronously) at the beginning of the application, so that such overhead will not be incurred at remote task execution time. With every placement request, the preferred host set is sent to the LIM. If a non–member host is found to be significantly better, it is added to the set, and a connection to it is requested. In selecting a non–member host to use, the LIM gives preference to hosts in the local cluster, in a way similar to the use of the preferred host set.

Task placements are performed by local LIMs using the GLOBAL policy, and by the cluster master LIM using the CENTRAL policy. A potential problem with distributed task placement is *host overloading*, where a lightly loaded host is sent multiple jobs by possibly different hosts, thus becoming overloaded before other hosts notice its load increase [31]. Host overloading may have an adverse impact on performance, especially when dynamic task migration is not supported. A completely centralized algorithm such as CENTRAL is immune to such a problem, as the load indices of the resources on the target host are adjusted upward in anticipation of the additional load the task will generate. Similar technique for GLOBAL would only discourage multiple tasks from the same host from being sent to the same target host, as all the placements are made by the local LIM. In UTOPIA, after responding to an application's placement request, the local LIM sends a load adjustment message to the cluster master LIM to reduce the probability of several hosts sending tasks to the same host.

Parallel applications often need to place a group of related tasks onto remote hosts. In a placement request to a LIM, the number of hosts needed may be specified, together with task resource requirements. The LIM selects up to/exactly the requested number of hosts and returns them to the application. It is also possible for an application to make its own task placement by requesting from the LIM a set of load vectors for hosts that satisfy some restrictive resource requirements. For instance, the load sharing batch facility we have built, *lsbatch* (see section 5.3.4), requires longer term load information (such as 15-minute average CPU queue length and host idleness), rather than the load indices used by the LIM for placement. For such applications, the up-to-date load information from the LIMs is needed, but not task placement.

3.4 Master LIM Election

The master LIM in each cluster is a resource critical to load sharing. If the master LIM (or its host) goes down, the load information kept by the slave LIMs would rapidly become out of date in the case of the GLOBAL algorithm, and remote execution should be turned off. With the CENTRAL algorithm, task placement becomes unavailable as soon as the master LIM goes down. Although load sharing is primarily a performance improvement tool, so the absence of which should not affect the correct operation of the system, such situation should be corrected in a timely manner. We propose a simple master election algorithm for UTOPIA as illustrated by the state diagram in Figure 3.

Upon initialization, the LIM in each host reads the *cluster configuration file*, in which each host in the cluster is listed and assigned an order number, with the first host in the file being host 0. If host 0 is up, it is the master. At start up, any other host first enters the *no master* state where it waits until it receives a load matrix (for GLOBAL) or an "I am alive" message from the master (for CENTRAL), at which point it moves to the *slave* state. The event of the master LIM going down is detected by the slave LIMs after they have not heard from it for more than an allowed number of communication intervals. Each LIM will then go into the *no master* state, wait for a period of time proportional to its order number, and announce its intention to become the new master if no other host has done so. Using the host number in the computation of the waiting period reduces the probability of multiple hosts trying to become the master at about the same time. In case a conflict does arise, the potential master LIM with the smallest order number will insist its *master* state, and other LIMs will retire to the *slave* state.

Theoretically, it can be shown that election in distributed systems with possible network and host failures cannot be guaranteed to succeed in any finite amount of time [12]; however, our experience using UTOPIA over the past years shows that no maintenance of LIMs is

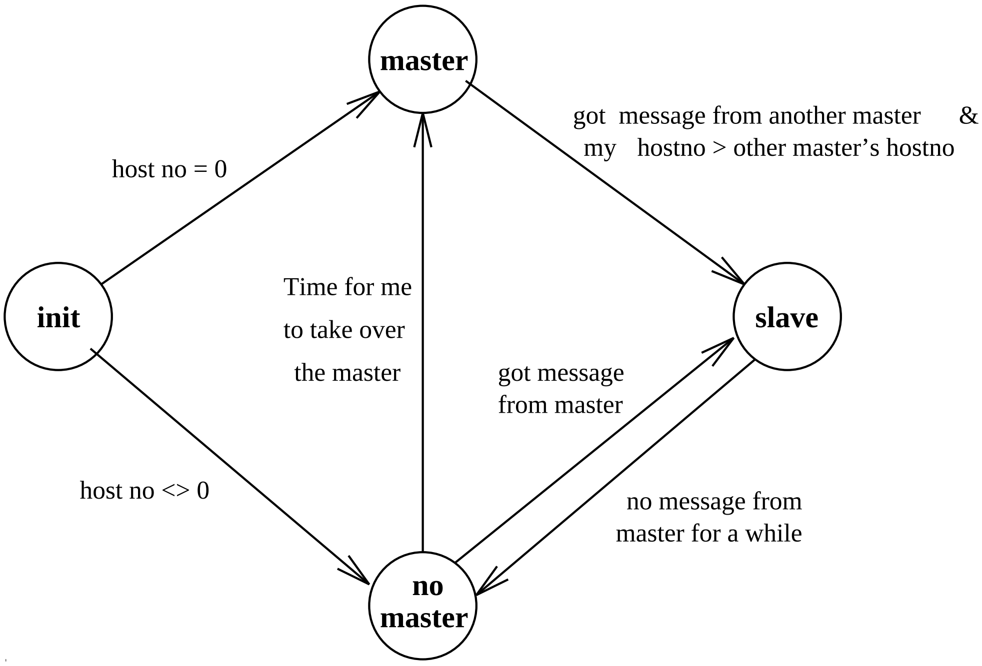


Figure 3: LIM state diagram (including master LIM election).

needed; the LIMs are always there as long as their hosts are up, and the periods with no master LIM are short.

3.5 LIM Overhead

LIM overhead may be measured in terms of response times of placement requests, and the amount of CPU time consumed and the network traffic caused by LIM. The response time of a placement decision is critical to application performance because the application has to wait synchronously for a decision to be made. This time depends on the total number of hosts to be considered and the number of hosts selected as a result of a placement decision. It also depends on LIM's policy (GLOBAL or CENTRAL). Table 1 lists the measured placement decision response times for different system sizes, number of hosts requested, and LIM policies. The measurements (and most of the subsequent ones in this paper) are performed on 60 SUN SPARC IPC workstations on two Ethernet networks supported by a single SUN 4/490 file server. We observe that the response times are relatively insensitive to the LIM policies. The delay grows slightly with the cluster size. It takes longer if the application requires more than one host to be selected by the LIM. In all cases, however, the response time is negligible compared to the typical amounts of CPU time consumed by eligible tasks (on the order of second). Small tasks requiring little CPU time to process are usually not eligible for remote execution, and therefore will not incur this delay.

Some LIM CPU and network overhead measurements are presented in Table 2. The CPU

Total number of candidate hosts	Number of hosts requested					
	CENTRAL policy			GLOBAL policy		
	1 host	5 hosts	10 hosts	1 host	5 hosts	10 hosts
15	3.8 ms	4.6 ms	5.0 ms	3.8 ms	4.5 ms	4.9 ms
33	4.5 ms	7.7 ms	8.4 ms	4.5 ms	7.5 ms	8.3 ms
60	5.8 ms	14.2 ms	18.5 ms	5.7 ms	13.5 ms	17.9 ms

Table 1: Task placement response times of LIM.

Load sharing configuration	Maximum CPU overhead of the master LIM		Maximum network traffic per second	
	CENTRAL	GLOBAL	CENTRAL	GLOBAL
One Sun cluster, 15 hosts	0.08% CPU	0.07% CPU	0.04 Kbytes	0.07 Kbytes
One Sun cluster, 33 hosts	0.15% CPU	0.13% CPU	0.09 Kbytes	0.17 Kbytes
One Sun cluster, 60 hosts	0.29% CPU	0.29% CPU	0.17 Kbytes	0.31 Kbytes
Two Sun clusters of 33 & 27 hosts, sharing each other	0.22% CPU	0.23% CPU	0.37 Kbytes	0.52 Kbytes
Four hosts from the above two Sun clusters form a virtual cluster, sharable by both physical clusters	0.17% CPU	0.19% CPU	0.11 Kbytes	0.20 Kbytes
As above, but the virtual cluster is also sharable by a DEC cluster	0.19% CPU	0.20% CPU	0.12 Kbytes	0.21 Kbytes
Emulation of 1000 hosts configured into 20 clusters of 50 hosts each. 100 (powerful) hosts are logically collected to form two virtual clusters of 50 hosts each, sharable by all clusters. *	1.1% CPU	1.5% CPU	1.20 Kbytes	2.06 Kbytes

* We emulate load exchange messages and task placement requests to the LIMs of a "typical" cluster so that the CPU and network resources are consumed as if there were 1000 hosts in the whole system. We made the following assumptions in the emulation: 1) A "typical" cluster receives load information from the two virtual clusters as well as one particular physical cluster, and sends its load information to another physical cluster. Thus, each host in the cluster can share all 50 hosts in the local cluster, as well as 150 hosts in other clusters. 2) There is one placement request from each host every two minutes, which is a higher frequency than the average rates we observed during the busiest hours in our local system. Compared with load information exchange, task placement decision making contributes only a small portion of the total CPU and network overhead.

Table 2: Master LIM CPU and network overheads.

overhead of a master LIM is much higher than that of a slave LIM, so only the maximum CPU overhead of the master LIM(s), as measured by the percentage of CPU time consumed by it, is given. We measured the overhead for a variety of configurations such as single clusters, multiple clusters, and multiple clusters with one or more virtual clusters. The load information exchange interval is 10 seconds, a value that ensures reasonably up-to-date load information.

It can be seen from Table 2 that, in all but the emulated case of 1000 hosts with large clusters and extensive inter-cluster load sharing, the maximum CPU overhead caused by the busiest master LIM is well below 1%. Even in the latter case, an overhead of 1% CPU for the master LIM in a 50-host cluster seems to be reasonably low. The CPU overhead grows slightly with the cluster size. Using the CENTRAL policy, the master LIM must handle all service requests (load update and placement requests) from all hosts in the local cluster. On the other hand, the master LIM now does not need to broadcast load information to all hosts in the local cluster. CENTRAL and GLOBAL algorithms consume comparable amounts of CPU time.

The network traffic generated by load information exchange is measured by adding the sizes of all messages sent by all LIMs in the largest cluster. With an exchange interval of 10 seconds, the probability that a host will send its load information (i.e., a significant change in one or more of its load indices has occurred during the last 10 seconds) was observed to be between 0.27 to 0.33 during busy hours. The CENTRAL policy out performs the GLOBAL policy in terms of network traffic generated, showing better scalability, since the load matrix is not broadcast by the master LIM. Except for the emulated large system, the two-cluster case generates the highest volume of traffic, but still consumes a negligible percentage of the Ethernet network bandwidth (0.04% with the GLOBAL policy). The virtual cluster configurations significantly reduce this overhead by selectively collecting load information from powerful hosts and distributing it to physical clusters, thus a host knows about the load information on all hosts in the local cluster and powerful hosts in one or more virtual clusters. Using the virtual cluster approach, the network traffic overhead is comparable to the single cluster case, yet still allows the sharing of powerful hosts among clusters. For the emulated case, which we expect to be a highly demanding configuration, the network traffic is still only 0.16% of the Ethernet network utilization using the GLOBAL policy. The above observations lead us to believe that using the virtual cluster approach and the CENTRAL policy, the LIM overhead would still be reasonably low for systems of several thousand hosts.

4 Transparent Remote Execution

4.1 Mechanisms

UTOPIA strives to achieve a high degree of execution location transparency by creating an execution environment for a remote task that is almost identical to the one that it would have if it were executed locally. Remote execution is initiated when a new process is created to run some executable code. A remote task is created by the parent process, and is started on a remote host with the help of the Remote Execution Server (RES) there. Like LIM, an RES is started at system boot time on each host, and waits at a well–known address for remote connection requests. During remote execution, neither the parent nor the child (i.e., remote task) knows that they are on different hosts — their behaviors and interactions are as if they were on the same host. The RES on the remote host acts as an intermediary to facilitate this transparency.

Before any remote execution on a host can be started, a load sharing application must first set up an initial stream connection to the RES on that host, and send a copy of its local execution environment. This execution environment will be stored by the RES on a per–application basis and will be used to start any task of that load sharing application. In an Unix system, the environment information may include user and group identifiers, default file creation mode, interval timers, signal masks, resource usage limits and statistics, current working directory, and environment variables such as terminal parameters and display name for windowing environment.

In addition to maintaining (close to) identical execution environment for remote tasks, RES also performs, for all remote sessions on its host, multiplexing and demultiplexing of input to and output from their remote tasks, and supports all Unix signals. For interactive applications, all terminal input and output associated with the remote task are transferred between the local application and the remote RES, which acts as the master side of a pseudo–terminal [21] to provide an emulating terminal for the remote task.

Similarly, whenever the user sends a signal to the remote task, for example, by typing control_C or control_Z for interrupt or stop signal, they will be propagated to the remote task through the connection between the application and the remote RES. The RES, upon receiving this signal, will send the signal to the remote task. Thus the remote task can receive the same signal as if it were running locally. In the other direction, in the event of the remote task being stopped or exiting, RES will inform the local application. If the remote task exits, the exit value is sent to the application. If the remote task is killed/stopped by some signal, the signal that killed the process is sent to the application. In this way, the parent process, i.e., the load sharing application, will know the status of the remote process.

A user–level implementation of remote task execution cannot achieve full transparency in an Unix environment. For UTOPIA, almost all applications will not notice any effect due to execution location. There are, however, a few minor breaches of transparency. For instance, there is currently no support for distributed process group, so a remote group member would not receive all the signals its local siblings would.

4.2 Two Models for Remote Execution

Several possibilities exist for the structuring of remote execution, and they can be grouped into two models. Before discussing the models, however, we examine the possible states of a load sharing application on a particular remote host, as illustrated in Figure 4. When an application first starts, it has no context on the particular remote host (in the *disconnected* state). The transition to the *connected* state may be initiated at the time of a remote task execution, or beforehand to reduce the delay for the first remote execution. Also, this transition

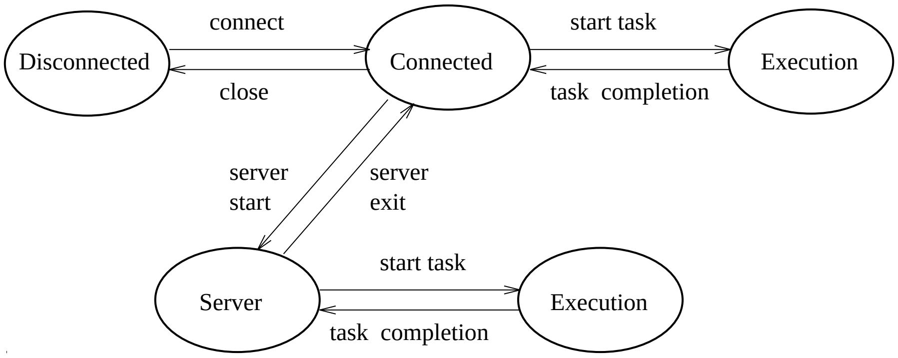
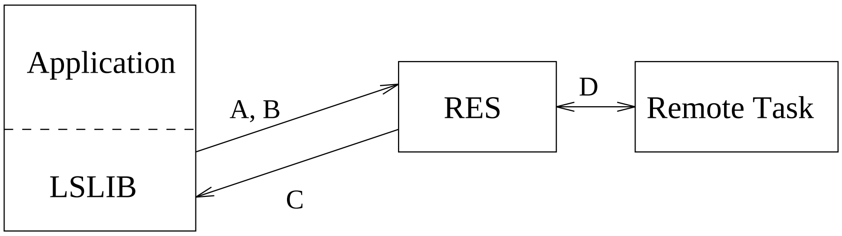


Figure 4: State transition diagram for a load sharing application with respect to a remote host.

may be made asynchronously: as soon as a connection is set up and the user environment information is transmitted to the RES, the local application returns to perform other work, while the RES establishes the execution context on the application's behalf.



1. Connect to remote RES (using ls_connect)

   A. Establish connection and send context to RES

2. Request remote execution of task (using ls_rexec)

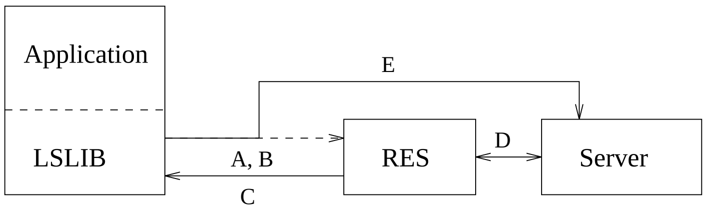
   B. Send command and arguments to RES

   C. RES calls back to establish I/O connection

   D. RES creates remote task

Figure 5: Remote operation model in UTOPIA.

Remote execution may happen in a variety of ways. An application may have an isolated task to be executed on a remote host, or may expect to have a sequence of remote tasks for it. A loose analogy would be datagram and virtual circuit models of communication over a network. In the first case, a message needs to be sent, and the simplest way is to treat it in isolation and route it to the destination host using its network address. In the second case, however, a stream of messages are expected to be sent to the same host, so it may be more



1. Connect to remote RES (using ls_connect)

A. Establish connection and send context to RES

2. Request remote server to be started (using ls_startserver)

B. Send server name and arguments to RES

C. RES calls back to establish I/O connection

D. RES creates the server

E. The server inherits the connection with the application;
RES closes the connection

Figure 6: Client–server model in UTOPIA.

desirable to form a "pipe" along the path to support this session of communication.

Similarly, two models of remote execution are supported in UTOPIA. An application may use the *remote operation model* to execute a task remotely, as shown in Figure 5. Upon a remote execution request by the application, LSLIB uses the initial connection between it and the remote RES to establish another stream connection for I/O and signal support. The initial connection is used for sending remote execution requests and responses between the application and the RES. It is possible for an application to have multiple outstanding remote tasks running on the same or different remote hosts, each with its own stream connection. For example, *lsmake*, a load sharing *make* facility that starts parallel subtasks on remote hosts uses the remote operation model. An application using the remote operation model typically creates a shared local *stub* process, which initiates the remote tasks as described above, and acts as an agent of the application handling terminal I/O and signals to/from its remote tasks. Once started, the stub process stays around until the application terminates. Thus, the cost of creating the stub is incurred once for each application run. In terms of the remote state diagram in Figure 4, each remote execution triggers a transition from the *connected* state to the *execution* state, and back when the remote task completes.

In contrast to the remote operation model, it may be desirable for a load sharing application to have its own server on the remote host that performs multiple tasks over time for the application. (The local application would also be called the *client* in this case.) We call such a model of remote execution the *client–server model*. For example, a command interpreter might want to have a server interpreter on a remote host so that it can provide many complicated features such as job control for commands running on a remote host [18]. RES, being a mechanism supporting remote execution, cannot and should not provide such sophisticated services particular to individual load sharing applications. In terms of the remote state diagram in Figure 4, a transition from the *connected* state to the *server* state may be triggered by the first remote task execution, or beforehand to reduce task execution delay. The reverse transition occurs either when the application completes, or after the server has remained idle for a sufficiently long period of time. A transition from the *connected* state back to the *disconnected* state may also be triggered to free up resources if the application has not used the remote host for a while.

Analogous to using a persistent local stub process, having a long running server helps to reduce the overhead of remote execution, for two reasons. First, the application–specific execution context may be kept on the remote site for the duration of the application so that, for each remote execution, such context does not have to be sent again. For this kind of load sharing applications, it is usually necessary to develop application–specific protocols for communication between the client and the server. Second, it is expensive to create a separate stream connection for each remote task, and it may not be necessary for load sharing applications such as a command interpreter; one connection may be enough. In such cases client–server model is desirable.

Figure 6 shows the mechanisms for supporting the client–server model. The mechanisms are similar to those for the remote execution model, except that, instead of sending a request for remote execution to the RES, the application sends a server creation request. The initial stream between the LSLIB and RES is then used directly between LSLIB and the server to support application-specific protocol, and another connection is established between LSLIB and RES for all terminal I/O and signals. No local agent process, such as *stub*, is involved — the application interacts with its server directly. Similar to the case with the remote operation model, an application may have several servers on the same or different remote hosts.

4.3 Remote Execution on Heterogeneous Hosts

Using initial placement of tasks for load sharing not only incurs lower overhead compared to checkpointing or dynamic migration, but also makes it feasible to execute remote tasks on hosts with different architecture or operating system than those of the local host. Since communication between the source and target hosts is carried out through a stream connection, and the interactions between the remote RES and the remote tasks occur on the same host, the same level of transparency is maintained by UTOPIA even if the two hosts are of different types. The ability to execute remote tasks on heterogeneous hosts implies that even if certain applications are only available on some type(s) of hosts, they are still transparently accessible throughout the system, a very attractive feature for many real-world environments. Restrictive resource requirements should be specified in the remote task lists for such applications so that an appropriate host would be selected by a LIM.

4.4 Authentication

The major security problem found in network services has to do with the authentication of a client requesting service from a server. In our case, the problem is whether RES can trust a load sharing application on the user identification it provides for remote execution.

UTOPIA supports two authentication options: a privileged port–based authentication scheme and an authentication server. The first scheme is also used by many other UNIX network applications, such as *rlogin* and *rsh* [19]. A connection request from a load sharing application to RES can only be honored if it comes from a privileged port number. The load sharing applications that use the LSLIB interface have to be root processes to be able to use privileged ports. Such applications should be installed with the *suid* bit set and be owned by root [19]. Typically, at lSLIB initialization, a number of sockets would be created and bound to privileged ports with effective user ID being root, the application then would revert to its real user ID, which is that of the user who invoked the application. It is undesirable to have a process running as root unless it is absolutely necessary. At the time of connecting to a RES on a remote host, a previously created privileged socket is used so that RES can trust the user ID.

In the second authentication scheme, security checking is done by an authentication server running as root on each host [27]. When RES receives a connection request from an application on a remote host, it will contact the authentication server on the client host by sending the port number used by the application to it. The authentication server then checks the port number allocation by reading the relevant kernel data structure and returns the ID of the user owning this port number to the RES. If the user ID returned to RES matches the one supplied by the application, the connection requested by the application is granted. This scheme is more convenient than the privileged port–based scheme in the sense that applications do not need to be *setuid* programs. It maintains the same level of security as the privileged port scheme. The extra delay for contacting the authentication server is observed to be only a few tens of milliseconds.

4.5 Remote Execution Efficiency

Remote execution of a task can be started from any state shown in Figure 4. Table 3 gives the measured times to execute a trivial task on a remote host when the application is in different remote execution states and when different remote execution models are used. In measuring the times for the client–server model, we used the *lstcsh*[5]. We observe that remote execution is much faster using UTOPIAthan the Unix utility, *rsh*, used by most of the current distributed applications. For the client–server model, once the server is started, further tasks sent to the server can be executed with a delay of around 0.1 seconds, which is hardly perceivable. For the remote operation model, once the stub process is started and a remote connection is set up by the first remote execution, further remote execution cost can be as low as 0.2 seconds.

The times in Table 3 assume that the operations are synchronous, i.e., the local application waits until the remote side finishes. UTOPIA system provides flexible ways for more

[5] We assume that the *lstcsh* is already in the memory, meaning that at least one session of the *lstcsh* is in active use by someone on that host so that it does not need to be loaded from the disk. We do this because, depending on the size of the executable file and the speed of the file system, the time to load the file varies widely. We want to focus on the efficiency of remote execution, not the performance of a specific server program or the file system.

Host pair	Client–server model			Remote operation model		
	from discon. state	from connected state	from server state	from discon. state w/o stub	from discon. state w/ stub	from connected state (w/ stub)
DEC→DEC	1.26 s	1.00 s	0.10 s	0.87 s	0.62 s	0.32 s
Sun IPC→Sun IPC	0.89 s	0.68 s	0.09 s	0.79 s	0.60 s	0.22 s
DEC→Sun IPC	1.08 s	0.88 s	0.13 s	0.92 s	0.71 s	0.28 s
Sun IPC→DEC	1.12 s	0.90 s	0.15 s	0.69 s	0.51 s	0.26 s

Table 3: Time to run a trivial task remotely (DEC = DEC 3100, Sun IPC = Sun Sparc IPC).

efficient remote execution. In fact, the LSLIB routines for establishing remote connections, starting remote tasks, and starting servers are all asynchronous: they all return as soon as all necessary information is passed to the RES on the remote host. Both establishing a remote connection with the RES and starting a server asynchronously take less than 0.1 seconds. Thus, for a parallel application, establishing connections with 10 remote hosts takes less than 1 second. Depending on the nature of a particular application, remote connections can be set up asynchronously at the beginning of the application, or servers can be started asynchronously at initialization time. This asynchronous nature of remote execution allows the local application to proceed in parallel with the remote processes, improving performance significantly, as will be shown in Section 5.4 for *lsmake*.

5 Applications and Their Support

5.1 Applications Support: LSLIB

Utopia is intended to be a layer of system software that aids users in exploiting computing resources scattered around a distributed system to improve applications performance and resource accessibility. As such, it is very important that the widest range of applications be efficiently supported, and that the system be easy to use. Ultimately, if load sharing is truly transparent, the users should not have to learn about it, or even be aware of its presence, except for improvement in performance. Utopia goes a long way to achieving this.

Application support in Utopia is realized through a runtime library, LSLIB, which provides a high level procedural interface to a set of applications [30]. These applications exploit load sharing directly, in the sense that they have to be modified to use the library; hence, we call them *direct load sharing applications*. Most parallel and distributed applications fall into this category, as subtasks are identified within their programs, and need to be sent to remote hosts. Direct applications (such as those to be discussed in Section 5.3) may in turn support applications that do not have to be changed or recompiled/relinked. These latter applications exploit load sharing indirectly, so we call them *indirect load sharing applications.*

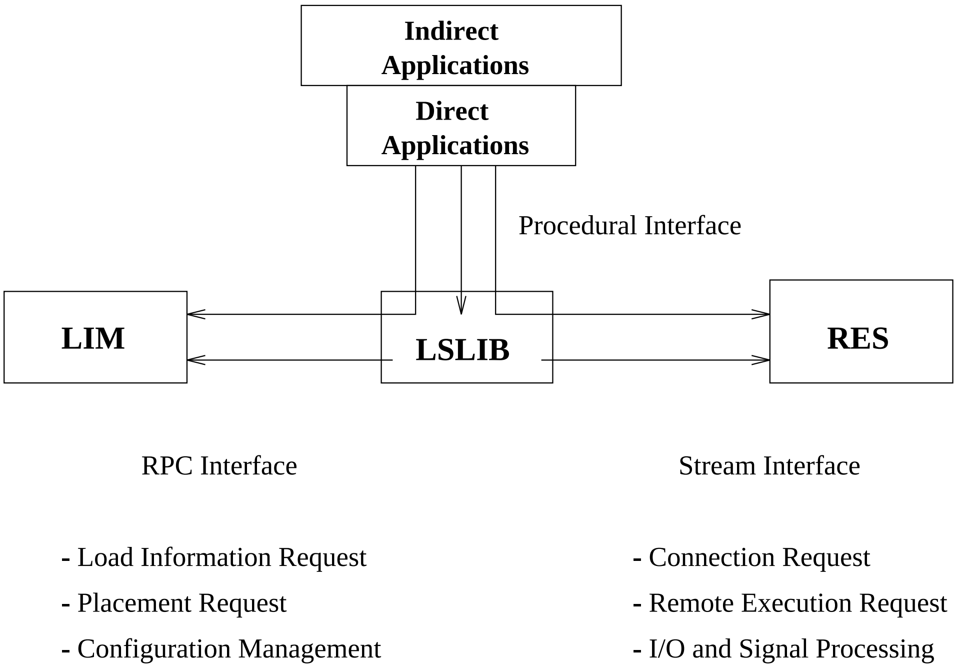


Figure 7: Interactions between applications, LIM, and remote RES through LSLIB.

LSLIB serves as a focal point through which all parts of the UTOPIA system interact with each other, as is shown in Figure 1 (in Section 2) and Figure 7. Three types of operations are supported by the LSLIB interface:

1. calls to LIM for load information and placement advice;
2. calls to remote RES for connection establishment and remote task execution and manipulation; and
3. calls internal to LSLIB for task eligibility checking and local state query and manipulation.

The first two types of calls are converted into requests to be sent to LIM and RES, respectively. Additional messages may be sent by the library on its own initiative.

5.2 Application Interface: A Simple Example

To illustrate the use of the UTOPIA library interface by applications, we present a "toy" application that implements a simple command interpreter [6]. The code presented in Figure 8

[6] For detailed specification of the UTOPIA application interface, the reader is referred to the LSLIB interface manual [30].

```
#include <lsf/lsf.h>

   ... ...

   ls_init();                          /* load sharing initializations */
   gethostname(myhostname, MAXHOSTNAMELEN);
   desthost = malloc(MAXHOSTNAMELEN);  /* space for best execution host name*/

   for (;;) {                          /* repeatedly read and execute commands */
      printf("Command>");              /* print prompt */
      read_command(commandName, arguments);           /*Read in a command line*/
      if (!ls_eligible(commandName, resourceReq)) {   /*trivial command?*/
         local = TRUE;                                /*local execution */
      } else {                         /*non-trivial command, ask LIM for best host*/
         ls_placereq(resourceReq, 1, &desthost, 0, 0); /*res. req. of the cmmd used*/
         if (strcmp(desthost, myhostname) == 0) {  /*local is best? */
             local = TRUE;
         } else {                      /*need to execute remotely on desthost*/
             local = FALSE;
             if (!ls_isconnected(desthost, socks)) /*connection not existent?*/
                 sock[0] = ls_connect(desthost);   /*set up initial connection*/
         }
      }

      pid = fork();                    /* create a child to execute the command*/
      if(pid == 0)  {                  /* I am the child process */
         if (local == TRUE)            /* local execution ?*/
            execvp(commandName, arguments);
         else                          /*send command to desthost to execute */
            ls_rexec(socks[0], commandName, arguments, 1);
         printf("Execution failed\n"); /*child process should not return */
         exit(-1);                     /*execution failed; child process exits */
      } else {                         /* I am the parent process */
         wait(&status);                /* wait for child to finish */
      }
  }
```

Figure 8: Simple command Interpreter using LSLIB interface.

is very close to an actual program except for omissions of some variable declarations and the routines to process the command strings. The application repeatedly reads in command lines from the user and runs them, either locally or remotely, in response to local and remote load changes and to the types of commands being submitted. For simplicity, the remote operation model is used. Users of this interpreter is completely unaware of the remote execution of some of her commands.

There is overhead for remote execution. Such overhead includes set up of the remote execution environment as well as transfer of input/output data from/to the local host to/from the remote host. So, often a load sharing application wants to execute a task remotely only if the resource demands of the task is non–trivial. A load sharing application can check whether a task is eligible for remote execution by calling *ls_eligible* (all the LSLIB interface calls have names starting with "*ls_*"), which searches through the remote and local task lists initialized by the *ls_init* call.

Placement decision by the LIM is obtained by calling *ls_placereq*, with the number of hosts requested as a parameter (in this case, 1). If the host name returned by *ls_placereq* is not that of the local host, the job is to be started on that host. If a connection with the RES on that host is not set up yet, *ls_connect* is first called to set up the connection, which is later used to transfer the command name and its arguments to the RES for remote execution. The program then creates a clone using the Unix system call *fork* and calls *ls_rexecv* on the child side to execute the command remotely. If for any reason the command is to be run on the local host, the program will simply *fork* and call *execvp* on the child side. In either case, the parent will simply wait for the completion of the child before initiating the next command.

5.3 UTOPIA Applications

The LSLIB interface is simple and high level, yet capable of supporting a diverse range of application types, many of which unforeseen at system design time. We discuss five types of applications below, each illustrated by an example.

5.3.1 Supporting Interactive Applications: Lstcsh

Lstcsh [29, 8] is a modified version of the popular Unix command interpreter, *tcsh* [1]. In essence, its operation is similar to the the toy interpreter discussed in Section 5.2, except that the server-client model of remote execution is used for its greater efficiency. An extensive set of functionalities are provided in *tsch*, such as job control, file name completion, command aliasing, history, and command path search. Our modifications to the program are well isolated, and have no effect to its functionalities other than the addition of remote command execution capabilities. As in the case of the toy interpreter, LIM is consulted just before command execution, and RES is used if remote execution is advised. Using *lstcsh*, essentially *all* Unix commands and user programs can be transparently executed on remote hosts without change.

For instance, a screen–based editor such as *vi* or a window–based application behaves identically on local and remote hosts. The individual commands in a shell script submitted

Utility program	Description
lseligible	Given the name of a program together with its arguments, check its eligibility for remote execution and obtain its resource requirements
lsplace	Given resource requirements, get task placement advice from LIM
lsrun	Execute a command on the specified remote host

Table 4: Three examples of simple utility programs supporting shell script applications.

by a user may be executed on different hosts, or the whole script may be shipped to a remote host from which its commands are further distributed.

5.3.2 Supporting Command Script Applications

A set of simple utility programs, three of which are shown in Table 4, have been developed to allow users to build customized load sharing applications at the command (shell) script level, often without changing existing programs. Such applications can be powerful while involving little effort. For example, at a large technology company, a sophisticated set of command scripts that perform load build (i.e., incremental updates of object and other derived software modules) for very large software systems have been modified using UTOPIA and the utility programs to explore large numbers of workstations for substantial performance gains. Although rather crude, command scripts can be effective in supporting parallel and distributed applications, as the user no longer has to select suitable hosts, log onto each of them, or use *rsh* [19] to start the job manually, a ritual all too familiar with people trying to use "bare" distributed systems for parallel execution. We have used this technique to support a distributed shared memory system, Mermaid [33], which runs a parallel application on heterogeneous hosts and maintains a virtually shared address space among them.

For simplicity, we choose to use a simple Bourne shell script to illustrate the use of the utility programs. The script in Figure 9 has exactly the same functionality as the example shown in Figure 8.

Alternatively, *lstcsh* built-in commands that provide the same services as the utility programs listed in Table 4 can be used to develop load sharing applications in the form of *lstcsh* scripts.

5.3.3 Supporting Parallel and Distributed Applications: Lsmake

UTOPIA supports parallel and distributed applications at two levels. The first level is command (shell) scripts developed using the utility programs or *lstcsh* as explained above. The second, more powerful level is to use the LSLIB interface directly. A load sharing make facility, *lsmake*, is one such example. *Lsmake* is a modified version of GNU *make*, a make facility that can execute multiple tasks on the same host to take advantage of parallelism between processing and file I/O. Modest speedups may be gained using GNU *make* as compared to a regular (sequential) *make* facility. We modified the program so that some of the tasks may

```
#! /bin/sh
... ...                                               /* initializations   */

PATH=......

myhost=`hostname`

while (true); do                                      /* loop for ever  */
echo Command\>
read command                                          /*Read command line */
if [ "$command" ]; then
    resourcereq=`eligible $command`
    if [ $resourcereq ]; then                         /*trivial command?*/
       if [ $resourcereq = "NON-ELIGIBLE" ]; then
          $command                                    /*local execution */
          continue
       fi
    fi
    exechost=`placereq $resourcereq`                  /*find the best execution host*/
    if [ $exechost = $myhost ]; then                  /*local is best? */
       $command                                       /*local execution */
       continue
    fi
    lsrun -h $exechost $command                         /*send for remote execution */
fi
done                                                  /*ready for next command*/
```

Figure 9: Simple command interpreter using Bourne shell script.

be executed on remote hosts. The changes are isolated to the module for starting subtasks, mostly new code for task placement and remote execution, which is similar in most direct load sharing applications. At startup time, a set of host names is obtained from the LIM for remote execution, and connections to the RESes on these hosts are initiated asynchronously for later use. The number of hosts (including the local one) to use is usually specified by the user, but the LIM may choose to provide a smaller number of hosts if there is a shortage of lightly loaded, compatible hosts. While the remote hosts perform authentication checking and set up the remote execution environment for this application in parallel, *lsmake* processes the make file and identifies parallel tasks to be dispatched to the hosts. As soon as a task on a host completes, another task, if available, is dispatched to the same host. For an application with a sufficient amount of parallelism, all the hosts may be kept busy, thus substantially speeding up the application execution, as will be shown in Section 5.4.

Lsmake may be regarded as a simple parallel application in which all communication occurs between the "master" task on the "home" host and the "slave" tasks on the "home" and remote hosts. UTOPIA is also used to support general purpose communication libraries (such as PVM) that provide message passing and synchronization among all the hosts participating in the execution of a parallel application. Together with UTOPIA, such a library will allow many of the parallel applications running on dedicated distributed memory parallel computers such as the Intel Hypercube computers to be executable on workstation clusters, making parallel computing available in distributed systems with little or no additional investment in equipment. The dynamic task placement facilities in UTOPIA are expected to be crucial to the feasibility of executing resource intensive parallel applications without affecting interactive users, whereas the efficient remote execution facility in UTOPIA is expected to be important to efficient execution of parallel applications with relatively fine granularity of parallelism.

5.3.4 Supporting Batch Applications: Lsbatch

We have used UTOPIA to support a load sharing distributed batch system, *lsbatch*, that supports parallel as well as sequential jobs. All batch jobs in a load sharing cluster are submitted to the *master batch daemon*, *mbatchd*, running on the same host as the master LIM of the cluster. The mbatchd puts the jobs into batch job queues, monitors the load of all hosts, dispatches jobs to hosts ready to start them, and reports job status upon user requests. Each participating host in a load sharing cluster runs a *slave batch daemon*, *sbatchd*, that receives batch jobs from the mbatchd, starts them, and controls their execution. Since batch jobs typically run for longer periods of time than interactive jobs, longer term load indices are appropriate. Therefore, instead of using the default placement advice from the LIM, the mbatchd obtains from the LIM the load vectors of hosts meeting the restrictive resource requirements of the batch jobs and makes its own placement decisions. We chose to use sbatchd as the remote execution mechanism for batch jobs, rather than the RES of the underlying UTOPIA, which is intended for interactive tasks. The flexibility of the modular structure of UTOPIA is again shown. Based on its needs, lsbatch can choose to use some parts of UTOPIA, such as its clustering structure, load information, and master election (see below), but not others, such as placement and remote execution.

The advantage of co-locating the mbatchd with the master LIM is that the distributed batch facility is easily made fault tolerant by taking advantage of the existing master LIM election. Information about pending and running batch jobs are logged to a file to enable a new mbatchd to take over after a master failure. Distributed job batching among multiple load sharing clusters can be accomplished by having a mbatchd in each cluster (just like the master LIMs), which coordinate among themselves to executed batch jobs in remote clusters when local cluster's load is high.

5.3.5 Supporting Session Applications: Lslogin

In many real world systems, users enter an application system (e.g., a CAD design environment or a database access facility) directly upon logging on, and perform most of their tasks within such applications, rather than using system-provided user interfaces such as a command shell. Similarly, a user may request to start an application session and have a window started on her workstation for it. In such cases, it is desirable to select a suitable host to support the user's session. We have implemented a load sharing version of the login program, *lslogin*, that selects a powerful and/or lightly loaded host for a user to log into. The program obtains a single placement advice from the LIM through the LSLIB. Similar applications can be developed with little effort.

5.3.6 Comments

An important contribution of the UTOPIA facility to load sharing research is the proof by example that a general-purpose, *extensible* load sharing system can be built with simple components and a uniform application interface. UTOPIA provides a unifying framework for many applications to take advantage of distributed resources; most of the existing load sharing systems for specific types of applications, such as distributed batch facilities, login time load balancer, and remote job server (e.g., text processor [17]) are *subsumed* by UTOPIA. Most of the above direct load sharing applications can in turn support indirect applications, such as jobs running on *lstcsh* or within *lsmake*. While each of the direct applications may be used independently, one direct application may also be used to support others. For instance, parallel applications can run with the distributed batch facility to allow users to submit many resource intensive parallel jobs all at once to be executed when the required resources become available. Job scripts can also be run with the batch system. Our experience in porting applications to UTOPIA shows that the system is capable of supporting a wide range of applications, and that application porting is usually easy. The original semantics of the applications is preserved and a high degree of transparency is achieved, while the distributed resources are exploited. As the system becomes more widely used and more experience is gained, we expect more load sharing applications to be developed, typically by converting single-host applications to exploit network resources.

Number of hosts used	lstcsh, 51 files, 51 compilations				X11R4 library, 245 files, 490 compilations			
	without linking		with linking		without linking		with linking	
	time (seconds)	speedup	time (seconds)	speedup	time (seconds)	speedup	time (seconds)	speedup
1	243	1	256	1	2437	1	2451	1
5	54.9	4.4	63.9	4.0	503	4.8	520	4.7
10	33.9	7.2	45.2	5.7	271	9.0	284	8.6
15	25.1	9.7	37.1	6.9	207	11.8	219	11.2
20	21.4	11.4	32.0	8.0	156	15.6	172	14.2
25	20.2	12.0	30.2	8.5	143	17.0	155	15.8

Table 5: Performance of *lsmake*

5.4 Application Performance

We measured the performance benefits from two of the load sharing applications developed on top of UTOPIA — *lstcsh* and *lsmake*. The benefits from *lstcsh* is obvious, particularly in a heterogeneous environment where some hosts are much faster than others. For instance, processing a 30-page document using *latex* takes 190 seconds on a VAXstation 3200, but only 21 seconds running remotely on a DECstation 5000. Because of the relatively low overhead for remote execution, even some trivial commands can benefit from running remotely on a faster host. For example, an "*ls -F* " command on a VAXstation 3200 takes 1.9 seconds for a directory containing 44 files, while remotely executing it on a DECstation 3100 workstation takes only 0.9 seconds.

Lsmake achieves substantial speedups by taking advantage of the asynchronous nature of remote execution primitives of the UTOPIA. Table 5 lists the performance of *lsmake* for building the *lstcsh* and *X11R4* library code. The times for the one host case are measured using (sequential) GNU make, with no overhead for task distribution. The times for both the purely parallel phase (without linking) and the entire make process (with linking) are given. Speedups are nearly linear upto ten hosts, and maximum value of close to 16 are achieved for X11 library (a relatively large software system), at which point the shared file server becomes the central bottleneck for this I/O intensive application. Linking of object files into an executable or a library is a sequential process. We observe that, with increasing parallelism, the linking time remains stable (while the parallel compile time shrinks), accounting for an increasing portion of the total execution time and dragging down the speedup curve.

The speedup of *lsmake* is substantially higher than any other parallel make facility reported so far. For instance, the *pmake* facility in Sprite using kernel-supported process migration [9] and the load build system (more sophisticated *make*) in Apollo Domain [20] do not exceed speedups of 5 or 6. As discussed in Section 4.5, we believe that the higher performance of *lsmake* is attributable to the lower overhead of remote execution in UTOPIA, and the asynchronous nature of task dispatching adopted in UTOPIA. These measures not only keep all the remote hosts busy most of the time, but also increase the throughput of the central dispatcher, which could be a bottleneck when the task granularity is small, and the number of hosts employed large. In contrast, in Sprite, each placement request and system call to migrate a process at *exec* time takes just under 400 milliseconds, whereas a remote task takes 3 seconds to execute on the average.

The above comparison raises the interesting issue of the efficiency of user- versus kernel-level remote execution. Whereas a kernel facility such as that of Sprite provides full transparency, no application context is kept inside the kernel (and probably should not be). Consequently, each remote execution involves all of the remote environment setup as well as the transmission of the state information, such as process execution state and internal kernel state. The call does not return until the remote execution has started. In comparison, at the user level, connection and remote environment setup can be performed asynchronously before it is needed, and used for a sequence of remote tasks. Ultimately, only a request message to the remote host needs to be sent before the application can go back to other work. Our current implementation of *lsmake* still uses the remote operation model with a shared stub for full transparency. For many parallel and distributed applications, such as *lsmake*, however, such a level of transparency is not needed, and the dispatching cost can be further lowered by reducing the level of transparency supported. More generally, the efficient remote execution mechanism in UTOPIA makes it feasible performance-wise to start relatively fine-grained tasks on remote heterogeneous hosts, thus expands the range of parallel and distributed applications that can benefit from load sharing.

6 Related Work

Load sharing has been studied extensively over the last decade, both analytically and experimentally. While numerous studies using analytical models and simulations exist, few practical systems have been built.

The Condor system at the University of Wisconsin provides load sharing of sequential batch jobs [22]. The jobs that can benefit from Condor are assumed to take substantial amounts of CPU time to finish. In Condor, great emphasis is placed on workstation autonomy; remote jobs can be stopped and resumed at another host from their most recent checkpoint, which allows graceful preemption of remote jobs whenever a user reclaims ownership of her workstation. UTOPIA only supports initial job placement, thus jobs cannot be migrated to other workstations once they are started. Unlike UTOPIA, no assumption of uniform file name space is made in Condor; instead, file system calls are intercepted and sent back to the home host for data access. Condor aims at making use of idle workstations scattered around the network. In contrast, UTOPIA load sharing system supports both batch and interactive jobs and makes use of not only idle workstations, but also non-idle hosts that are relatively lightly loaded. Not only lengthy jobs can benefit from load sharing, short jobs such as the shell commands which take a few seconds can benefit as well, this makes slow hosts appear faster even for the jobs taking small amounts of time. The Condor jobs must be relinked with the Condor library in order to be executable remotely. UTOPIA applications, such as *lstcsh*, *lsmake*, or *lsbatch*, enable tasks executed remotely without recompilation or relinking.

The NEST project at AT&T Bell laboratories presents an implementation of a decentralized control dynamic load balancing facility [3]. It supported both batch and interactive jobs. However, changes to the Unix kernel were deemed necessary in order to support transparent remote execution and to get a more accurate measure of host load. We decided to base our design outside the kernel since changing the kernel is very undesirable, particularly in a heterogeneous environment.

The Process Server developed at Xerox PARC supports load balancing among workstations running CEDAR, and uses centralized algorithm for load information collection and job placement [14]. No mention is made of the load information used. Applications must be modified for remote execution.

The Butler system at CMU starts a butler server on an idle workstation to handle remote execution requests, which are submitted by specifying *rem* in front of eligible commands [25]. Only idle workstations are used for remote commands, and, if a workstation is reclaimed by its owner, the butler warns and kills any guest process on it.

Several distributed operating systems, such as Locus [28], Sprite [9] and V [24], support transparent remote execution by migrating processes dynamically. However, none of the systems is intended for large-scale and heterogeneous environments. The policies used by these systems for task placement are quite primitive, without considering applications' resource requirements, system heterogeneity, and scalability.

In contrast to other existing systems, UTOPIA load sharing facility provides more comprehensive and efficient support for resource sharing. It supports a much wider range of parallel and distributed applications than any system reported so far.

7 Concluding Remarks

Dynamic and transparent load sharing in large-scale distributed systems makes the vast amount of computing resources scattered around the system available to the users. Through transparent remote execution, remote powerful hosts may be used to improve application performance, and those applications available only on some of the hosts may be invoked from any where in the system. As the resource demands of the applications escalate, incremental upgrade (i.e., upgrade only some of the hosts in the system at a time) becomes an economical and practical approach to keeping up with the technology advances, since, with load sharing, the new, more powerful hosts are transparently accessible throughout the system. Parallel applications may provide dramatic increase in performance of resource intensive applications without impacting the interactive users.

In this paper, we discussed the design, implementation, and performance of UTOPIA, a load sharing system for large, heterogeneous distributed systems. Viewed at a higher level, UTOPIA provides four functions important to integrating a distributed system into a coherent, efficient system:

1. efficient distribution of dynamic state information in large-scale systems,

2. transparent and efficient remote execution on heterogeneous hosts,

3. intelligent and adaptive task placement advice, and
4. flexible load sharing support for a diverse collection of applications.

Such services can be used in isolation or in combination to exploit remote resources transparently.

Besides producing a general-purpose load sharing system, UTOPIA, we made two research contributions to the field of resource sharing in distributed systems. First, we proposed algorithms for load information distribution scalable to systems with thousands of hosts. A combination of centralized algorithm within the load sharing clusters and decentralized algorithms among the clusters based on directed graph and virtual clusters are used to achieve scalability. Second, we designed and implemented a wide range of remote execution mechanisms at the user level that offers several models of remote execution (such as remote operation, client-server, and batch). Application progrmamers may then select the most efficient mechanism that satisfies the application's requirements for performance and ease of use. Compared to kernel-based remote execution facilities, our system appears to be more efficient and flexible due to our ability of performing remote operations asynchronously and before they are needed to amortize the cost of remote execution context setup over multiple tasks.

UTOPIA allows virtually all Unix commands and user tasks to be executed on remote hosts transparently, without change to their code or recompilation/relinking, and provides support for the development of parallel and distributed applications. Since the system is implemented at the user level, heterogeneous hosts can share load with full transparency, and the configurational differences among the hosts are fully exploited to improve performance. UTOPIA currently runs on Ultrix, Sun-OS, and HP-UX, and is easily portable to other Unix systems. Early measurements show that the system overhead is low, and applications performance can be substantially improved. Various versions of UTOPIA have been in use in our Distributed Systems Laboratory at the University of Toronto for several years. We are currently conducting an evaluation study at on a system of several thousand hosts at a large technology company.

Acknowledgments

This work has been partially funded by an External Research Grant from Digital Equipment Corporation, by Bell Northern Research Inc., and by a University Research Incentive Fund from the Government of Ontario. Xiaohu Zheng was partially supported by an Ontario Graduate Scholarship. Xinguan Lin worked on the implementation of *lsbatch*. We are grateful to Sean Gates, Jane Hearnden, and Dan Willis of Bell North Research for their feedback on the experience in using UTOPIA in their systems.

References

[1] *Tcsh User's Manual.* Cornell University.

[2] R. Agrawal and A. Ezzat. Location independent remote execution in NEST. *IEEE Transactions on Software Engineering*, 13(8):905–912, August 1987.

[3] R. Agrawal and A.K. Ezzat. Processor sharing in NEST: A network of computer workstations. In 1st *International Conference on Computer Workstations*, pages 198–208, November 1985.

[4] G. Champine, Jr. D. Geer, and W. Ruh. Project athena as a distributed computer system. *IEEE Computer*, 23:40–51, Sep 1990.

[5] E. D. Lazowska D. L. Eager and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12:662–675, May 1986.

[6] E. D. Lazowska D. L. Eager and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. *Performance Evaluation Review, ACM*, 16:63–72, May 1988.

[7] R. Dannenberg and P. Hibbard. A butler process for resource sharing on spice machines. *ACM Trans. on Office Information Systems*, 3(3):235–252, 1985.

[8] P. Delisle and J. Wang. *Lstcsh Man Page.* August 1991.

[9] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software — Practice and Experience*, 21(8):757–785, 1991.

[10] D. Notkin et al. Interconnecting heterogeneous computer systems. *Communication of the ACM*, 32(3):258–273, March 1989.

[11] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In *Proceedings Performance '87*, pages 515–528, Brussels, Belgium, December 1987.

[12] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31:48–59, Jan 1982.

[13] K. Goswami, R. Iyer, and M. Devarakonda. Load sharing based on task resource prediction. In *22nd Hawaii International Conference on System Science, Software Track*, pages 921–927, January 1989.

[14] Robert Hagmann. Process Server: Sharing processing power in a workstation environment. In *Proceedings of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 260–267, Cambridge, MA, USA, May 1986.

[15] John Howard and et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, Feb 1988.

[16] Kai Hwang, W. J. Croft, et al. A Unix-Based local computer network with load balancing. *IEEE Computer*, 15(4):55–66, April 1982.

[17] W. Johnston and D. Hall. Unix based distributed printing in a diverse environment. pages 514–528, June 1986.

[18] W. N. Joy. *An Introduction to the C Shell, 4.3 BSD Unix User's Manual*, chapter 4, page USD:4. 1986.

[19] W. N. Joy, R. S. Fabry, and K. Sklower. *4.3 BSD Unix Programmer's Manual*, volume 1. 1986.

[20] David B. Leblang and Jr. Robert P. Chase. Parallel software configuration management in a network environment. *IEEE Software*, 20(11), Nov. 1987.

[21] S. J. Leffler, R. S. Fabry, W. N. Joy, and P. Lapsley. *An advanced 4.3BSD Interprocess Communication Tutorial, 4.3 BSD Unix Programmer's Supplementary Documents*, volume 1, page PS1:8. 1986.

[22] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the* 8^{th}*International Conference on Distributed Computing Systems*, pages 104–111, San Jose, California, Jun 1988.

[23] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. In *Proceedings of ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[24] K. A. Lantz M. M. Theimer and D. R. Cheriton. Preemptable remote execution facilities for the v-system. In *Proceedings ACM-SIGOPS* 10^{th}*ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[25] D. A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the* 11^{th}*ACM Symposium on Operating Systems Principles*, pages 5–12, November 1987.

[26] R. Sandberg and et al. Design and implementation of the sun network file system. In *Proceedings of USENIX Conference*, May 1985.

[27] M. StJohns. Authentication server. *RFC 931*, Jan 1985.

[28] Bruce Walker et al. The LOCUS distributed operating system. In *Proceedings of the* 9^{th} *ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.

[29] J. Wang, X. Zheng, and S. Zhou. *Lstcsh User's Manual.* Sept 1991.

[30] J. Wang, X. Zheng, and S. Zhou. Utopia applications interface. Technical report, Computer Systems Research Institute, University of Toronto, Aug 1991.

[31] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, November 1988.

[32] S. Zhou and D. Ferrari. A measurement study of load balancing performance. In *Proceedings of* 7^{th}*International Conference on Distributed Computing Systems*, pages 490–497, October 1987.

[33] S. Zhou, M. Stumm, D. Wortman, and K. Li. Heterogeneous distributed shared memory. *to appear, IEEE Transactions on Parallel and Distributed Systems, also as Technical Report No. 244, Computer Systems Research Institute, University of Toronto*, September 1990.