The Design of a Load Balancing Mechanism for Distributed Computer Systems

Harry I. Rubin

Report No. UCB/CSD 87/362

July 1987
PROGRES Report No. 87.5

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The Design of a Load Balancing Mechanism for Distributed Computer Systems

Harry I. Rubin

Computer Science Division
571 Evans Hall
University of California, Berkeley
Berkeley, CA 94709

(415) 642-3979

harry@Berkeley.EDU -or- ...!ucbvax!harry

ABSTRACT

We describe the design and planned implementation of a load balancer for a network of computers. The mechanisms that make decisions are separated from those that carry out actions, and different levels of centralization are chosen for each. Load balancing decisions are made by one-per-machine managers; this allows better administrative control, consideration of recent history in making decisions, and reduces duplication of effort. Load balancing actions include establishing communication connections, sending task descriptions, executing a task, returning results, and so on. Load balancing actions are decentralized, each program performs its own, because these actions can be time-consuming and centralizing them could form a bottleneck.

Sending decisions include unload decisions, eligibility decisions, and placement decisions. These are made by a load balancing send manager (LBSM); there is one LBSM on each machine. The decision to accept an offered task for execution is made by a load balancing receive manager (LBRM); there is one LBRM on each machine. The LBSM and the LBRM read configuration files which instruct them on how to make their sending and receiving decisions. These configuration files provide for administrative control of load balancing.

Programs that wish to execute tasks under load balancing contact the local LBSM and send descriptions of the tasks. The LBSM makes the sending decisions and replies with the name of the machine selected for each task. The originating program then contacts the LBRMs on the selected machines and submits the tasks. The LBRMs either accept each task and execute it, or send back a refusal message. Each remote task is executed in a separate execution environment. Results are sent back to the originating program directly.

We expect that the main user of load balancing will be load balancing command interpreters (shells), but a library of routines is planned which will make it easy for other programs to use the load balancer.

Table of Contents

The Design of a Load Balancing Mechanism for Distributed Computer Systems

Harry I. Rubin

Computer Science Division
571 Evans Hall
University of California, Berkeley
Berkeley, CA 94709

(415) 642-3979

harry@Berkeley.EDU -or- ...!ucbvax!harry

1. INTRODUCTION

In a distributed computer system, it often happens that some computers are heavily loaded while others are lightly loaded or idle. Theoretical studies and simulations have promised tremendous improvements in user response time by moving tasks from heavily loaded computers to lightly loaded ones (for instance [Livny and Melman 1982], [Zhou 1986]). We use the term *load balancing* to refer to any movement of workload from one computer to another for purposes of improving performance. The issues involved in load balancing can be divided into mechanism and policy. Mechanism issues deal with the structure of the load balancer, which functions are performed by which components, how the components carry out their functions, how they communicate with one another and with clients, and so on. Policies are the algorithms (or heuristics) by which load balancing decisions are made. Policies and mechanism are not independent. Policy decisions are supported by, communicated through, and acted upon by the load balancing mechanism.

In this paper we discuss the design of a load balancing mechanism. This design is intended to be the basis for a production load balancer. We also intend to use it as a testbed for experimentation with various load balancing policies. Although to a certain extent the design of the mechanism restricts or at least influences the policies that can be supported, we do not feel this is a problem. We believe that the present mechanism is both general and flexible: it will accommodate most of the policies we will wish to try, and it has been structured in such a way that it will be easy to modify to accommodate other policies.

1.1. Objectives

Load balancing facilities can be provided at either the command level or the program level. Load balancing at the command level means the load balancing of whole commands, as typed by a user at a terminal; the user does not write any programs, only enters commands. Obviously this type of load balancing must be handled by the user's command interpreter (in Unix, the shell). At the command level, load balancing should be automatic and transparent: once load balancing has been turned on, the user should not be required

Unix is a trademark of AT&T Bell Laboratories.

This work was sponsored in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing official policies, either expressed or implied, of the Department of Defense or of the U.S. Government.

to type extra commands or special versions of commands, or even extra characters in commands (as in Hwang, et al's design [Hwang et al. 1982]). Load balancing at the program level means that programmers can use the load balancing facilities when dispatching tasks from within a program (in Unix, tasks can be dispatched by *fork/exec* or by system calls such as *system*). Our main objective was to provide convenient and powerful load balancing facilities at both the command level and program level.

One approach to providing load balancing at both the command and program levels is to install load balancing in the process control facilities of the operating system kernel, for example, a load balancing *fork*. We have rejected this approach for a number of reasons. First, it is not at all portable: neither our design nor our planned implementation would have been easily transportable to a different type of computer system. Second, at the process control level, the only information available is information about a single process. We believe that more information may lead to better load balancing decisions, therefore we wanted a design that would allow use of as much information as possible. Command lines, or lists of tasks to be dispatched from a program, can contain much information about multiple tasks to be run concurrently or serially, about communication between tasks, and so on. Performing load balancing at the process level in the operating system would not allow use of this information. Finally, making extensive modifications to the kernel of an operating system is a difficult and error-prone undertaking. For these reasons, another of our objectives was to avoid modifying the kernel of the operating system.

We believe that load balancing is an "extra" that users and programmers should be able to use, but they should not be required to use it. If they do choose to use it, it should be as convenient, automatic, and transparent as possible.

Making it possible to execute a new command under load balancing should be easy. Some load balancing systems require that a new "front end" program be written for every command that is to be executed under load balancing [Bershad 1985]. We consider such an approach unacceptable; it is too difficult to administer. Making it possible to execute a new command under load balancing should be as easy as adding the command to a load balancing list.

Administrators have a responsibility to control the use of their machines, and sometimes to control the execution of sensitive tasks. Load balancing that involves moving tasks off one machine and onto another could pose a threat to administrators' abilities to carry out these responsibilities. Administrators must be able to control which tasks execute on the machines for which they are responsible. In some cases, administrators are responsible for protecting the execution of certain tasks, in which case they must be able to control whether those tasks are executed locally or remotely, and, if remotely, to which machines the tasks could be sent. The ability of the administrator of a machine to control that machine and its tasks is called *local autonomy*. We feel that it is vital for a load balancer to provide a high level of local autonomy. Of course, it should be convenient to administer local autonomy.

In short, our objectives for this design were:

- to provide load balancing at the command level for commands typed by users at terminals
- to provide load balancing at the program level for programs dispatching tasks
- to make load balancing as convenient, automatic, and transparent as possible at both levels
- to allow the load balancer to handle any task without writing new software for each task
- to make load balancing facilities available but not mandatory
- to provide a high level of local autonomy
- to make the load balancing system convenient to install and control administratively
- to do all this without modifying the kernel of the operating system

We believe we have succeeded in accomplishing these objectives.

2. CENTRALIZATION AND DECENTRALIZATION OF DECISIONS AND ACTIONS

In the load balancer described here, decisions are made in a centralized fashion but are carried out in a decentralized fashion.

A load balancer performs two kinds of operations, the making of load balancing decisions and the carrying out of those decisions. We refer to the carrying out of a decision as an *action*.

There are several load balancing decisions, some made by a machine considering sending a task elsewhere, and some made by a machine receiving a task from elsewhere:

Sending Decisions	
unload decision	should any tasks be unloaded now?
eligibility decision	is a particular task eligible to be unloaded?
placement decision	where should an eligible task be sent or placed?
Receiving Decisions	
acceptance decision	should a received task be accepted or refused?

Similarly, there are some load balancing actions which occur on a sending machine and others which occur on a receiving machine:

Sending Actions
establish connection to receiving machine send task send any other information required (possibly) wait for task to complete receive results (if any)
Receiving Actions
accept connection from sending machine receive task set up execution environment for task execute task send any results back to sending machine clean up

Load balancing activities, both the making of decisions and the carrying out of them, can happen at any one of five levels of centralization, from completely decentralized to completely centralized:

- per task
- per user (user environment, or login session; in Unix, per shell)
- per machine
- per cluster of machines (defined on load balancing clusters)
- globally (all machines that could possibly participate in load balancing)

Centralized decision-making means that a single entity makes all decisions for the globe or cluster or machine or login session. Decentralized decision-making means that many entities are making decisions independently. Carrying out load balancing actions in a centralized way means that some central entity carries out all actions. Carrying out load balancing actions in a decentralized way means that many independent entities are carrying out actions independently.

By centralizing decision-making on a per-machine basis, load balancing actions by all users of the machine can be taken into account, and thus better decisions can be made. Centralizing load balancing decisions also eliminates certain duplications of overhead. An even more centralized approach, per-cluster or global, would stand in danger of becoming a bottleneck, would involve more off-machine communication (which is slower than on-machine communication), and would involve the greater complexity of detecting and reacting to a failure of the cluster controller or global controller. We believe that a carefully designed and implemented per-machine load balancing decision-maker will be able to make decisions quickly, so that it does not become a bottleneck. Experience and experiments will tell whether this is true.

Recent work by Zhou has led him to claim that per-cluster load balancing may deliver better performance than per-machine load balancing ([Zhou 1986]; also see [Zhou and Ferrari 1987]). Zhou's work, however, differs from our own in a number of significant respects. Zhou combined two functions, distributing load information and making load balancing decisions, into one component, and based his conclusions about centralization on the cost of distributing load information. In our design the two functions are embodied in two separate components, so deciding that load balancing decisions will be made on a per-machine basis still allows load information to be distributed on a per-cluster basis, or on any other basis. Furthermore, Zhou implicitly assumed that placement decisions for all machines would be made in the same way. Our scheme, however, includes extensive administrative controls to allow for local autonomy on a per-machine basis. This makes it awkward to have a single per-cluster decision-maker; the per-cluster decision-maker might have to make decisions according to different criteria depending on which machine was the source of the task. Such a per-cluster decision-maker would be more complex to design and implement, and would run more slowly, increasing the possibility that the decision-maker could become a bottleneck. Another problem with per-cluster load balancing is also due to local autonomy: if load balancing decision-makers run on each machine, it is easier for the administrator of each machine to control load balancing. In a cluster-based system, if the cluster-controller ran on a remote machine, the local administrator might have to depend on the administrator of the remote machine to control the movement of tasks onto and off of the local machine. For our objectives, making load balancing decisions at the per-machine level seems better.

Load balancing actions could take a relatively long time to perform since they involve communication with remote machines and waiting for actions on remote machines. Communication with remote machines is done via relatively slow off-machine communication mechanisms. Remote machines may take a long time to complete an action either because the action is complex and computationally expensive, or because the machine is slow, or both. If a remote machine is down, or crashes during an action, the originating machine may have to wait for a time-out period to discover that. If load balancing actions were centralized, even at the per-machine level, the action-manager could become a major bottleneck. By decentralizing actions, we eliminate the danger of bottleneck.

In the load balancer described here, load balancing decisions are made by one-per-machine entities, while load balancing actions are carried out on a per-user basis by per-login entities (in Unix by the shell).

3. THE ENVIRONMENT

The design proposed here is fairly general in that it makes few assumptions about the operating environment. The design could be implemented in any environment that fulfills those assumptions. This section describes those aspects of the operating environment on which the design of the load balancer depends.

3.1. File System

The load balancer assumes that files are visible and accessed by the same name from all machines (this is sometimes called *transparency*). The load balancer does not check that this is true, nor does it take any actions to ensure it. We considered designing the load balancer to copy input files to remote machines or result files back, if necessary, but our view is that a distributed system supporting a load balancer should also include a distributed file system that provides these services.

File access costs need not necessarily be uniform across machines; a file residing on one machine must be visible and accessible under the same name from a remote machine, but it might be far more costly in terms of delay to access the file from the remote machine. Load balancing policies might well take into account the relative costs of file access on different machines (as, for example, in Hac and Johnson's study [Hac and Johnson 1985]).

Although we have not dealt with it in this design, if the distributed file system allows files to be moved from machine to machine automatically, then perhaps the load balancer and the file system could negotiate about whether to move the work nearer to the files or the files nearer to the work. If the distributed file system supports replication of files, then copies of needed files could be created near the work site.

All file names must be expanded or converted to a format which is absolute, not relative, across machine boundaries, so the file naming system must include such an absolute format.

3.2. Process Migration

Our design does not rely upon process migration. Load balancing is done by initial placement of tasks; once a task is accepted by a machine, it runs to termination on that machine.

It is possible that a task that was accepted under one set of circumstances could become unwelcome on its host machine later in its lifetime if circumstances change. Nevertheless, once a task is accepted it runs to completion, even if this later becomes inconvenient or undesirable. Without process migration there is little alternative. In the case of a multi-user computer, this is a small problem; users must expect that tasks belonging to other users will affect the response times of their tasks. In the case of a single-user workstation or personal computer the problem is more significant, if only for psychological reasons ("this is *my* workstation ..."). We have considered some approaches to ameliorating the problem:

- foreign tasks which become unwelcome could be suspended for a while, or they could have their priority lowered. But such actions would increase response time, defeating the purpose of load balancing.
- tasks which become unwelcome could be aborted and an attempt made to automatically restart them from scratch elsewhere. But it may be difficult for the load balancer to know whether and how to restart a task. Alternatively, unwelcome tasks could be aborted and an error reported to the originating process, which may choose to restart the task elsewhere, but it may be difficult even for the originating process to know whether and how a task can safely be restarted. Aborting a task may leave data in an inconsistent state requiring special processing to clean up, more than just restarting the task from scratch; the possibility of this type of abortion makes load balancing unattractive to potential users. At the least, an aborted task has wasted all the processor time and all the real (wall clock) time it has used, increasing response time and defeating the purpose of load balancing. (Consider a two-hour computation aborted at one hour and fifty minutes.)

Neither of these alternatives is attractive and we have not included either in the design. Tolerating foreign tasks which have become unwelcome is part of the price one pays for participating in a load balancing scheme.

To implement process migration, if it is not already implemented, would require extensive changes to the operating system. We preferred to design a load balancing system entirely outside the operating system. This will make it possible to implement the load balancer on a wider variety of systems, it will make implementation easier, and it will make installation of the load balancer easier.

Another reason for not using process migration in the load balancer is that migrating an executing process would involve sending the process's entire execution image to the destination machine. With large physical memories and even larger virtual memories, execution images could reach megabytes or even gigabytes in size. The time delay of sending so much data through the communication medium would be significant. There is also the burden on the medium, and on other users, of sending so much data.

Because load balancing is done by initial task placement only, and does not require moving execution images, it should be possible to do load balancing across heterogeneous machines, as long as the machines support the same commands. Alternatively, load balancing decisions could take into account which machines are able to execute which commands. Either type of arrangement would require some way to refer to the different commands that would execute on the different machines by some generic command name. The Locus system uses "hidden directories" for this purpose [Walker et al. 1983], while Cronus keeps track of "execution environments" and their requirements [Hoffman et al. 1982], [Schantz et al. 1983]. Other approaches are possible [Rubin]. Load balancing across heterogeneous machines also raises the question of slightly different results obtained from different machines due to different word sizes, different arithmetic hardware, and other factors. It could be most disturbing to the user to run the same program on the same input data and obtain different results. We have not focused on load balancing across heterogeneous machines, and we do not deal further with the problems of generic versus machine-specific command names or differing results.

3.3. Multiprogramming

The load balancer consists of a number of independent (though communicating) long-running processes, sometimes called *daemons*. Any operating system on which this design is to be implemented must provide multiprogramming, the ability to have several processes running or ready to run.

3.4. The Network

The network connecting the load balancing machines need only support point-to-point communication, that is, process-to-process communication, between machines. Certain operations of the load balancer can be made more efficient if broadcast or multicast are available, but these are not required for correct operation. Of course, the network must operate with a fairly low latency or some of the benefit of load balancing will be lost. It should be possible to operate the load balancer over a long-haul network (such as the Arpanet); although the load balancer should function correctly, the longer latency would probably make it difficult to improve response time through load balancing. All communications between components of the load balancer, and between user processes and the load balancer, are reasonably short ASCII text messages; therefore the network need not provide extremely high bandwidth. (If migration were used, then entire virtual memory images, possibly several megabytes in size, might have to be sent over the network. In that case, a high bandwidth network would be more necessary.)

Subject to the above requirements, we believe it would be possible to implement our design on most any hardware, software, and network.

4. COMPONENTS OF THE LOAD BALANCER

The load balancing decisions made by a sending machine (*load balancing sending decisions*) are quite different from the load balancing decisions made by a receiving machine (*load balancing receiving decisions*); so, there are separate entities to handle the different decisions. The one-per-machine entity responsible for making load balancing sending decisions is the *Load Balancing Sending Manager* or LBSM. The one-per-machine entity responsible for making load balancing receiving decisions is the *Load Balancing Receiving Manager* or LBRM. Each machine participating in load balancing will normally run both an LBSM and an LBRM. (An administrator could choose to have a machine act only as a sender, in which case it would not need to run an LBRM. Alternatively, an administrator could choose to have a machine act only as a receiver, that is, be a compute server, in which case it would not need to run an LBSM. Generally, each machine will run both an LBSM and an LBRM and automatically adjust to being a sender or a receiver as the situation changes, for example, being a sender when its load is high and a receiver when its load is low.)

Two other software entities support the LBSM. In making its decisions, the LBSM can use information about the loads on other machines. The *Load Information Manager*, or LIM, maintains and updates load information and provides it on request to the LBSM. The LBSM can also use information about the characteristics of particular tasks. The *Task Information Manager*, or TIM, maintains and updates this information and provides it on request to the LBSM. The LBSM communicates with the LIM and the TIM via inter-process communication (IPC). The LIM and the TIM will reply to queries from any process, so other system processes or user processes may obtain information from them.

User processes that utilize the load balancer are actively involved in some of the load balancing operations, for example, they initiate the load balancing of tasks. In this sense, user processes are part of the load balancing scheme, and their part in it is described.

Figure 1 illustrates the operation of the load balancer, as explained in the following sections.

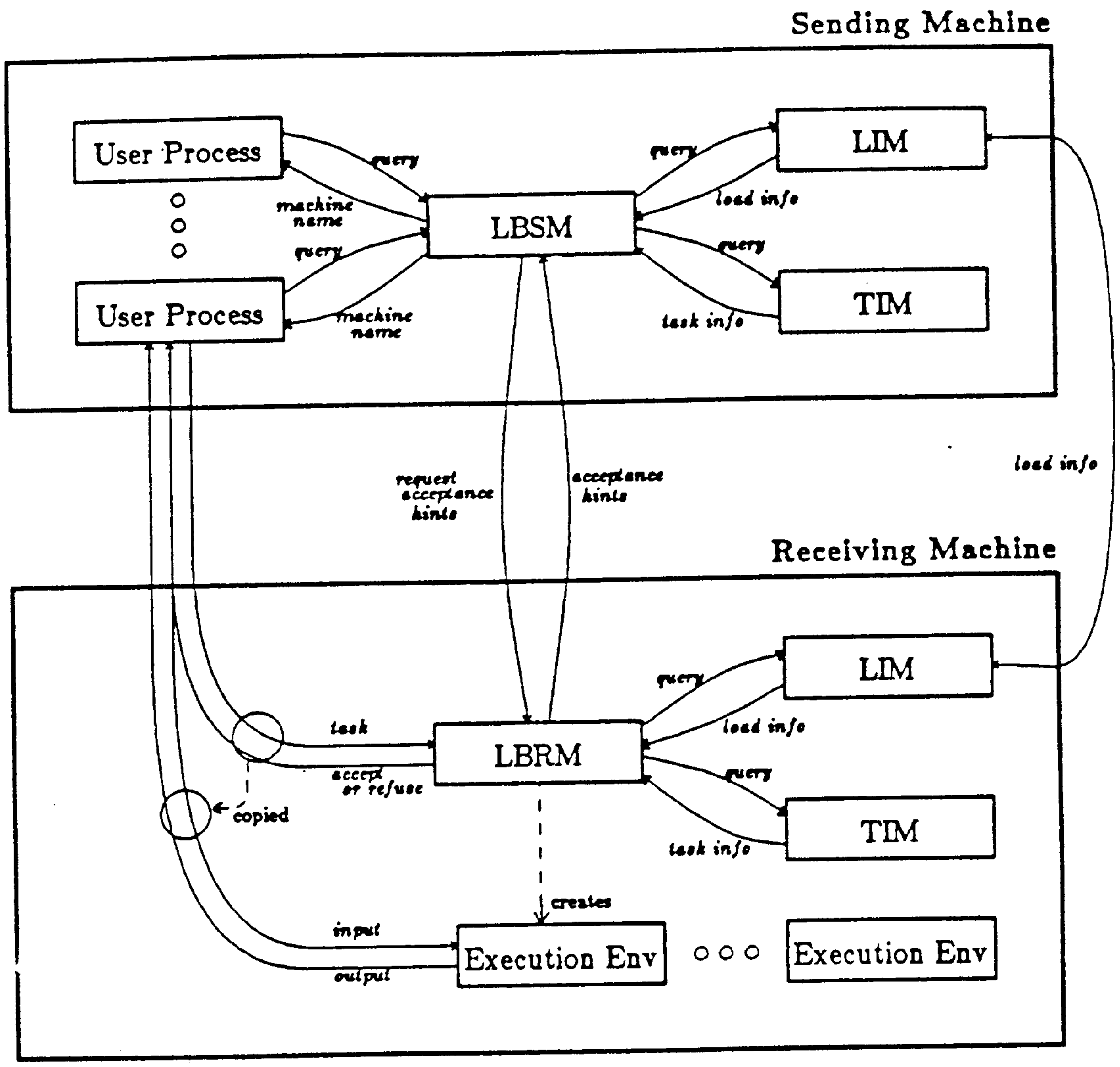

Fig. 1: The Load Balancing Mechanism — Shown for Two Machines

4.1. The User Process

Any process which executes tasks can use the load balancer. We expect that users' shells (or whatever entities represent users or login sessions on other systems) will be the main users of the load balancer, using it to improve the performance of commands typed by the user. Nonetheless, any program or process can use the load balancer. Possibilities include a load balancing Make [Feldman 1979], or a graphics program computing many picture elements. Because any user process can use the load balancer, we refer to "the user process" generically in what follows.

To make it easier for programmers to write programs that use the load balancer, a library of subroutines will be provided. The library will contain two types of routines: those which are self-contained and carry out complete (though simple) operations, and those which carry out parts of more complicated operations. An example of the first sort of library routine is a load balancing version of the Unix library call named "system," which executes a command line. An example of the second sort is a routine which establishes a connection to the local LBSM. Routines of the second sort can be used to do more complex load balancing, much like the load balancing Unix shell will do; for example, a load balancing Make could be implemented using these routines.

A user process that wants to execute some task or tasks subject to load balancing sends a query to the local LBSM. User process clients of the LBSM make no decisions (except that they want to submit certain tasks for load balancing); they know nothing about which tasks are eligible for load balancing or how placement decisions are made. This allows the clients to be simpler, eliminates certain duplications of overhead, and simplifies administrative control of load balancing. On the other hand, user processes are not required to be clients of the LBSM; if they want to make their own decisions they can do so according to any policies they wish. They can query the LIM and the TIM and contact the LBRMs on other machines. We expect that only very rarely will user processes choose not to use the LBSM for load balancing.

Clients communicate with the LBSM via whatever IPC mechanism is available and convenient; using the load balancing subroutine library makes communicating with the LBSM appear to be done as a subroutine call. The LBSM may consult the LIM and the TIM and its own information on recent placements. It makes load balancing sending decisions and replies to the user process with the names of the machines on which the tasks should be executed. Tasks which the LBSM decides should be placed on the originating machine are executed in the usual manner (in Unix, either by fork/exec or the system call named "system"). For tasks which the LBSM decides should be placed on remote machines, the user process sets up connections to the LBRMs on the machines selected and sends the tasks to them. The user process can then either wait for the results from the remote machines or proceed with other work.

If one of the selected receiving machines refuses to accept a task, it replies with a refusal message to the user process which sent the task. The user process then notifies its LBSM of the refusal and asks for a new placement decision. An alternative would be to have the refusing machine notify the appropriate LBSM directly, as well as notifying the user process, but this would place a greater load on the refusing receiver, which may be refusing because it is overloaded (or it could be for administrative reasons), so we prefer to avoid this alternative. The alternative approach might be appropriate if there was concern about user processes behaving correctly. We expect that the main client of the load balancer will be the shell, and that most other client programs will use the subroutine library provided; both the load balancing shell and the library routines will follow the proper protocols in dealing with the load balancing managers, and we will assume that any other user processes will also follow the proper protocols.

4.2. The Send Manager

The LBSM receives requests from user processes; it makes load balancing sending decisions in response to those queries (unload decisions, eligibility decisions, and placement decisions) and sends the results back to the querying processes. The LBSM maintains information about recent placement decisions it has made and can use that information in making new decisions. (How recent is recent depends on the frequency and reliability of the information the LBSM can get from the LIM.) The LBSM may query the LIM or the TIM and use their information in making decisions; the LBSM communicates with the LIM and the TIM through IPC. If the LBSM has no other duties and the local machine's load is low enough, it can try to precompute placement decisions for likely requests. (How the LBSM might precompute placements, and whether it ought to even try to do so, is a new area of investigation which we have not yet explored.)

Note that the LBSM may decide that a task or tasks should be executed on the local machine. This could be (a) because the local load is so low that it is not necessary to unload tasks, (b) because the tasks have been declared not eligible for load balancing, or (c) because the placement decision selects the local machine.

The LBSM does not fork child processes to deal with requests. Decisions must be made quickly enough to avoid forming a bottleneck, and the fork operation is fairly time-consuming. More importantly, the purpose of centralizing the LBSM is to be able to use knowledge of other load balancing decisions, and to avoid conflicting or destabilizing decisions; if decisions were made by separate processes it would be much more difficult to share knowledge and coordinate actions. (We are speaking here of "heavyweight" processes with separate address spaces. Many operating systems, including the one we plan to implement under, do not support "lightweight" processes sharing an address space, so we have not based our design upon them.)

If the LBSM encounters a problem of some sort, so that it cannot make decisions, it replies to all queries by saying all tasks should be executed locally, that is, it turns off load balancing. It would be an error for the LBSM to have a task sent somewhere it should not go, whereas turning off load balancing merely removes an enhancement to the system.

4.3. The Receive Manager

When a user process wants to execute some task or tasks subject to load balancing it consults its local LBSM, which provides the identity of a machine to which the user process should send the task. The user process then uses an intermachine IPC mechanism to establish a connection to the LBRM on the selected machine, and sends the task to the LBRM. The LBRM examines the task and decides whether or not to accept it. If the LBRM accepts the task, it sends an acceptance message back to the user process, it establishes an execution environment for the task, and it arranges for the task to execute. If the LBRM does not accept the task, it sends a refusal message back to the user process. The receiving machine has no obligation to provide service to the sending process.

For each accepted task, the LBRM creates an execution environment, if necessary obtaining information from the user process. (In Unix, for the load balancing shell, this will involve obtaining environment variables and shell variables from the originating shell.) The connection from the user process is duplicated and passed to the execution environments. The LBRM then causes the tasks to execute, each in its own execution environment. The tasks execute logically in parallel, so they may compete with each other for resources. If any task requires input data it reads it from the originating user process; any results or other output are written to the user process via the connection.

The LBRM's acceptance decision can be based on such factors as the local load, the name of the task, the characteristics of the task, the identity of the sending user, the identity of the sending machine, the number of "foreign" tasks already accepted, the number of users logged on, the time of day, and more. A file of acceptance criteria, maintained by the system administrator, specifies which factors, and what values of the factors, determine acceptability. The format and contents of this file are described in a later section. The LBRM may communicate with the LIM or the TIM running on the same machine.

The LBRM maintains a f'ᵉ of accounting information, containing a record for each foreign task accepted. Each r cord includes the originating machine and user, the real (wall clock) starting and ending times, and accounting data such as CPU time used, amount of I/O, memory occupancy, network traffic, and so on. Additional accounting data not directly associated with the execution of a task should also be recorded if at all possible; this includes data such as number of pages printed or typeset, number and type of tapes mounted, and so on.

The LBRM does not perform any accounting actions such as computing or assessing charges, it just writes out its accounting file. System administrators will probably arrange for the LBRM and accepted tasks to run under one account, for example, an account named "loadbal." System administrators could use the LBRM's accounting file to re-assign charges from this load balancer account back to the originating user's account. It is left to system administrators to arrange this, if appropriate.

When a user's task is executed on a remote machine via the LBRM, the remote machine charges the task execution to the load balancing account and the remote LBRM records the task execution in its accounting file. This design allows users to use the load balancer to execute tasks on remote machines without having to have an account on every remote machine (subject to acceptance criteria in the receivers' configuration file). Note that the accounting is carried out in this manner even if the user does happen to have an account on the remote machine. There are several reasons for this. First, doing it the same way in all cases makes the design and implementation of the load balancer simpler. Second, the accounting file is a technical record of the LBRM's activities as well as an administrative accounting record. Third, the account named "joe" on machine Alpha may belong to a different person than does "joe" on machine Beta, or the accounts may belong to the same person but be subject to different administrative or accounting arrangements (for example, they may be used for different projects billed to different clients). In any case, the accounting records should be kept separate.

4.4. The Load Information Manager

One Load Information Manager (LIM) executes on each machine. The LIM is responsible for obtaining and maintaining up-to-date information about the loads on all machines which might be considered for load balancing. (Other processes may consult the LIM about similar matters, so its duties may be a bit broader than just providing load balancing information.) To do this, it samples the load of the local machine and, when necessary, sends updates to other LIMs. It receives load updates sent by other machines and it may, if necessary, request load updates from other machines. The LIM makes its information available by responding to queries.

The LIM responds to three types of queries:

(1) query for the load of the local machine,
(2) query for the load of a single specified machine,
(3) query for all information the LIM has.

The LIM is used by the LBSM and the LBRM running on the same machine, but may also be used by other processes. Many operating systems run some sort of remote machine information facility (e.g. *rwhod* in Berkeley Unix, *rstatd* in Sun Unix). At some future time, these facilities might be merged with or replaced by the LIM.

4.5. The Task Information Manager

Different tasks consume different types of resources and cause different kinds of load. Tasks may run for a short time or for a long time, and they may consume a lot of resources during this interval or not. Some tasks may do much computation and little I/O. Some may do much disk I/O but little network communication, and so on. These are the characteristics of tasks.

The characteristics of a task depend not only on the program being run but also on the flags, options, and arguments given it. For example, using the C compiler to compile a short program will generally consume relatively less CPU than compiling a long program. Turning on code optimization will cause the compiler to run longer and to use relatively more CPU.

It is the responsibility of the Task Information Manager, or TIM, to obtain, update, maintain, and make available information on the characteristics of tasks. This information may be used by the LBSM and by the LBRM running on the same machine, and may be used by other processes as well.

The current design for the TIM is a fairly simple prototype. Some task information will be obtained through hand-done characterization studies, and this data will be loaded into the TIM. We plan to do experiments to see whether task information is actually helpful in making load balance decisions. If task information is shown to be useful in making load balancing decisions, or some other types of decisions, then a more sophisticated TIM may be designed. A highly sophisticated TIM might automatically monitor all task executions on its machine, or obtain monitored information from the operating system, and update its database based on the monitored information. It might share data with TIMs on other machines. It might try to parameterize task characteristics in terms of input file size or other determinants. We do not yet know how to do these things, and we do not know whether this is worth pursuing.

Queries to the TIM will specify a program (full path name), and as much information about flags and arguments as is available. The TIM will reply with information about the task characteristics for that program with those flags and arguments, if they are known. If the flags and arguments are not known, the TIM will send a "low confidence" or "approximation" reply telling what is known about the task, perhaps its characteristics with different flags or arguments. If nothing is known about the task, the TIM will send a "no information" reply. In a sense, all information provided by the TIM is a hint, because the TIM does not have information on every possible set of input data for every command (a possible exception being if a command was run recently and is rerun with the same inputs).

Having discussed the operation of each component of the load balancer, we next discuss the message formats by which the components communicate.

5. COMMUNICATION

The previous sections have described the components and operations of the load balancer. In this section, we discuss the communication protocols used to communicate with and between the components of the load balancer. The rationale for the design is given, and the protocols themselves are described in detail.

5.1. Communication in ASCII Text

All communication with and between load balancing managers will be by sending messages in ASCII text, as opposed to sending structures or records in some programming language's internal format. This will make it easier for user programs to communicate with the load balancing system, especially programs written in languages other than the one the load balancer is implemented in. It will also make debugging the load balancing system easier.

5.2. Formats and Keywords

To make it easier and more efficient for programs to generate, read, and understand messages, all messages must conform to a strict format. Every message begins with a message header and contains one or more sections. Each section begins with a section-header keyword, and the lines of a section have a set format. Extra spaces, tabs, and blank lines are not allowed.

All keywords are exactly four characters long. To make them into keywords, English words longer than four characters are abbreviated (sometimes drastically), and shorter words are padded with x's. In our planned Unix implementation, keywords must be in all upper case, except for padding x's which must be in lower case. As messages will be generated by programs, this strictness of format and case will not be an inconvenience to users.

5.3. Header and Ending

Every message begins with a message header identifying the sender of the message and the intended recipient of the message. This information is not strictly necessary, it is included as an error check and debugging aid. Any load balancer component receiving a message first checks to see that it is the intended receiver.

The first line of every message is the *from* line:

`FROM` *sending-component sending-machine*

Sending-component identifies the component sending the message; it is one of

`USER`, `LBSM`, `LBRM`, `LIMx`, or `TIMx`

All user processes are identified as USER. *Sending-machine* is the name of the machine (host) from which the message is being sent. The name can be in whatever format is convenient for the network being used.

The second line of every message is the *to* line:

`TOxx` *receiving-component receiving-machine*

Receiving-component and *receiving-machine* are similar to sending-component and sending-machine, only they identify the intended recipient.

Every message ends with a line containing only the keyword

`ENDx`

It is an error to have any extra text after the ENDx.

5.4. Send Manager Queries

We have previously described how user processes wishing to have tasks submitted for load balancing will send a query to the LBSM. The LBSM then makes decisions and replies with the identities of machines where the tasks should be sent. In this section we describe the format of queries to the LBSM in greater detail. We describe the format of the LBSM's replies in the next section.

A query to the LBSM consists of several sections. A *tasks* section is required; *communication*, *dependency*, and *placement* sections may be present if needed. Each section is introduced by a section-header keyword and terminated by the beginning of the next section or by the end of the message.

5.4.1. Tasks Section

The first section of an LBSM query is the *tasks* section, which lists the tasks concerning which the LBSM is to make decisions. The tasks section begins with a section-header line containing only the keyword

TASK

The LBSM is allowed to use information about the characteristics of the task to be placed (*task information*), so queries to the LBSM must identify the tasks to be placed. The tasks must be completely and unambiguously specified, with any abbreviations fully expanded and all substitutions fully performed. In Unix, this means that the name of the program to be executed and all other filenames in the command must be specified by full pathnames. For the Unix shell, this means that the names must be alias expanded, history expanded, and filename and command expanded.

Task characteristics may depend on the flags, arguments, and command line inputs or parameters to the task, so the query to the LBSM must specify these, too. As with the program name, all flags, arguments, and parameters must be fully expanded.

A user process may have several tasks to be placed. In this case, the tasks are listed one after the other, one per line. The task lines are numbered, beginning with zero; the task number appears first on the line, followed by a single space, followed by the task name, then the flags, arguments, and so on. (If there is only one task, it is numbered with the number zero.) Each line, containing a task number, task name, and parameters, is terminated with a newline (or carriage return) character.

5.4.1.1. No-Placement Tasks (NOPL)

In some situations, a program knows that some tasks are to be run now and some are to be run later. This could be due to the user's instructions, as in the Unix command line

```
cc program.c -o a.out;  mv a.out program
```

or due to some inherent dependency (see below). The user process could ask the LBSM to make a placement for the "for later" task, but since the situation could change before the task is executed it would probably be better for the user process to make another query and obtain a placement decision later. On the other hand, the LBSM may be able to make better decisions if it is informed of tasks that are to be executed in the near future (we plan to do experiments to find out whether this is true). These tasks "for later" are included in the task list so that the LBSM can take them into account, but a special notation is used to tell the LBSM not to make a placement for the task. (If the user process does in fact want a placement decision now for a task to be executed later, it just omits the special notation. This is not an error, but it is probably not very wise either.)

The notation to instruct the LBSM not to make decisions about a task is this: immediately following the task number (no space) is a hyphen, immediately followed by the keyword NOPL (for "no placement"). For the example above, with the "cc" command to be done immediately and the "mv" command to done later, the tasks section would be:

```
TASK
0 /bin/cc /usr/tom/program.c -o /usr/tom/a.out
1-NOPL /bin/mv /usr/tom/a.out /usr/tom/program
```

NOPL tasks may be listed in the communication or dependency sections (see below). The LBSM may take NOPL tasks to be executed later into account in making decisions, but it does not reply to the user process concerning them. The LBSM does not save information about "later" tasks, as it has no way to know when, or even whether, they will actually be executed. If the user process later wants LBSM decisions concerning these tasks, it sends another query to the LBSM.

5.4.1.2. Already-Placed Tasks (more NOPL)

Use of the load balancer is optional, and other processes may make their own load balancing decisions (possibly after consulting the LIM or the TIM). Or outside decision-makers may make decisions about some tasks and ask the LBSM to make decisions about others. In either case, tasks for which placement decisions have already been made by outside decision-makers are called "already-placed" tasks. It would be helpful to the LBSM if outside decision-makers informed the LBSM of their decisions, although this is not required of them. This is done by listing the already-placed tasks as NOPL tasks and including in the query a placements section to describe where the already-placed tasks have been placed. The placements section is usually part of LBSM reply messages (see below for description of placements section); in this case the section only includes placements for the already-placed tasks. For example, if a placement has already been made for one C compilation, but placement of a second compilation has been left to the LBSM, the tasks section of the LBSM query would contain:

```
TASKS
0-NOPL /bin/cc /usr/tom/program1.c -o /usr/tom/p1.out
1 /bin/cc /usr/tom/program2.c -o /usr/tom/p2.out
```

The LBSM may take already-placed tasks into account in making decisions, but it does not reply to the user process concerning them. If an LBSM query contains only NOPL tasks, the LBSM replies with a message containing only a header and ending, to serve as an acknowledgement.

5.4.2. Communication Section

Among several tasks to be placed, some may communicate among themselves. Knowing the expected intertask communication patterns may help the LBSM to make better decisions. For example, communicating tasks may be placed on machines which are close together in terms of communication costs, or they may even be placed on the same machine, sacrificing parallelism for ease of communication. If a multiprocessor is available and has a low enough load, the communicating tasks may be placed there to get both parallelism and ease of communication. The algorithms to be used to make these decisions are still to be researched.

The second section of an LBSM query describes which tasks are expected to communicate with which other tasks. This section is optional and should be omitted if there is no communication or if the originating user process has no information about intertask communication. It is an error for the query to include a communication section if there is only one task listed in the tasks section. The communication section is introduced by a section-header line containing

```
COMM
```

Following this is a sequence of lines, each line containing the task numbers of two tasks that are expected to communicate with each other; the task numbers are separated by a space. Direction of communication is not considered, so "3 5" means the same as "5 3". If several tasks communicate with each other, all the communication paths must be listed. For example, if tasks 3, 5, and 6 all communicate with each other, the query would contain

```
COMM
3 5
5 6
6 3
```

An alternative we considered was to allow groups of communicating tasks to be listed all on one line. With this alternative, the example above with tasks 3, 5, and 6 communicating would have a communication section of

```
COMM
3 5 6
```

We believe the pairwise format will be easier for programs to generate and read.

If directionality were important, the format could be changed to indicate direction of communication. So if, for example, task zero sends information to task one and tasks one and two communicate back and forth, the communication section could have

```
COMM
0->1
1->2
2->1
```

Direction of communication is not important in any network we are familiar with, and we do not believe it is important for the LBSM to know direction of communication, so we have not adopted this alternative format.

The load balancer has no way to know which tasks are expected to communicate unless it is told by the user process. The load balancer does not cause tasks to communicate or even allow them to communicate, it merely takes into consideration in making its load balancing decisions the fact that they are expected to communicate. It is not an error for the user process to omit communication information, or to supply information on expected communication which turns out be wrong, it just means that the load balancing decisions may not be as good as they might have been.

5.4.3. Dependency Section

The third section of an LBSM query deals with execution-order dependencies. If one task must be completed before a second can be executed, then the second task *depends* on the first. A group of tasks may include complex sequences of dependencies, including both fan-out ("task 0 must precede both 1 and 2") and fan-in ("both tasks 4 and 5 must complete before 6 runs"). As with the communication section, this section is optional and should be omitted if there are no dependencies or if the originating user process has no information about intertask dependencies. It is an error for the query to include a dependency section if there is only one task listed in the tasks section.

The dependency section of a query begins with a section-header line containing

```
DPND
```

Following this is a sequence of lines, each line containing the task numbers of two tasks, separated by a space, where the first task must precede the second. In the example above, in which the user's command means that "cc" must precede "mv", the query would contain

```
TASK
0 /bin/cc /usr/tom/program.c -o /usr/tom/a.out
1-NOPL /bin/mv /usr/tom/a.out /usr/tom/program
DPND
0 1
```

We considered formats such as

```
0 -> 1 2
4 5 -> 6
```

but decided that it would be easier for programs to generate and to read the simpler pairwise format. Another possibility was to use a format similar to that used by the Unix utility Make [Feldman 1979]: for each dependent task, listing the tasks it depends on:

```
DPND
1 : 0
2 : 0
6 : 4 5
```

Here again, we decided that the simple pairwise format would be easier to use.

As with communicating tasks, the load balancer has no way to know which tasks must wait for which other tasks unless it is told by the user process. The load balancer does not ensure that one task completes before a dependent task begins, it merely takes the dependency into account in making load balancing decisions. It is not an error for the user process to omit dependency information, or to supply information on expected dependencies which turns out be wrong, it just means that the load balancing decisions may not be as good as they might have been.

5.4.4. Summary of Send Manager Queries

The end of a query to the LBSM is marked by a line containing only ENDx

We summarize with an example of a query sent by a user process to the LBSM:

```
FROM USER ernie
TOxx LBSM ernie
TASK
0 /usr/local/dtbl -Pdp /usr/harry/document.tbleqnms
1 /usr/local/deqn -Pdp
2 /usr/local/ditroff -Pdp -ms
3-NOPL  /bin/cc /usr/harry/program.c -lm -O -o /usr/harry/program.out
COMM
0 1
1 2
DPND
0 3
1 3
2 3
ENDx
```

This query would be generated from the Unix shell command line:

(dtbl -Pdp document.tbleqnms | deqn -Pdp | ditroff -Pdp -ms); cc program.c -lm -O -o program.out

but the query format defined is capable of expressing more complex relationships than the shell syntax.

Note that, in the example command line, there is no inherent reason why the C compilation cannot execute in parallel with the document formatting. Nevertheless, the user specified that they be done sequentially by using the shell sequence operator ";", so the sequentiality is expressed in the dependencies (and by the NOPL); the load balancing shell will not try to out-smart the user. If the user had used the shell parallel execution operator "&" instead of ";", then the document formatting and the compilation could execute in parallel and the NOPL and the dependencies section would have been omitted from the query.

5.5. Send Manager Replies

When it receives a query, the LBSM makes unload, eligibility, and placement decisions for the tasks in the query. If the unload policy decides that tasks are not to be unloaded, then the LBSM replies that all tasks should be executed on the local machine. If some tasks are not eligible for load balancing, then the LBSM replies that those tasks should be executed on the local machine. The placement decision may decide that a task should execute on the local machine or on some remote machine. Having made these decisions, the LBSM sends a reply back to the querying user process. The reply consists of a message header and a *placement* section.

5.5.1. Placement Section

The placement section's purpose is to inform the recipient about which machines tasks are to be executed on. For each task in the tasks section of the query (except those flagged NOPL), the placement section of the reply has a line with the same task number followed by the name of a machine or by "LOCL" (for LOCAL). "LOCL" means that the task should be executed on the machine where it was originally submitted, that is, the machine the task would execute on without load balancing. If a machine name is given, then the task should be executed on the named machine. Machine names are given in whatever format is appropriate and easily used by the network connecting the machines. The LBSM may return the name of the local machine for some tasks; this is the same as if it had returned "LOCL." An example of an LBSM reply message is:

```
FROM LBSM ernie
TOxx USER ernie
PLAC
0 LOCL
1 vangogh
2 ernie
ENDx
```

When a placement section is part of a query to the LBSM, then placements are only listed for those tasks which have already been placed.

5.6. Receive Manager Requests

We have described above how user processes wishing to submit tasks for load balancing send the tasks to the LBRM of the machine chosen by the LBSM. In this section and the following one, we describe the interactions between user processes and LBRMs in greater detail.

The method a user process uses to establish a connection with a remote LBRM depends on the existing local networks and software. In Berkeley Unix, the user process looks up the network address of the remote machine, looks up the on-machine address (port number) for the LBRM service, opens a socket and connects it to the machine address and port number. User programmers should not have to deal with this sort of detail to use the load balancer, so we plan to provide library routines which will establish connections.

Once a connection has been established, the user process sends a message to the LBRM, consisting of a message header, an execute section, and an ending. Messages to the LBRM do not include communication (COMM) or dependency (DPND) sections; the LBRM does not need the communication or dependency information either to make acceptance decisions or to execute the tasks. It is up to the user process and the tasks themselves to establish the communications and enforce the dependencies.

5.6.1. Execute Section

The execute section of a request to the LBRM lists the tasks its sender would like to have executed on the remote machine. The execute section's format is identical to the format of the tasks section, except of course for the section-header keyword, EXEC, and except that there are no NOPLs:

```
EXEC
0 /usr/local/dtbl -Pdp /usr/harry/documentfile.tbleqnms
1 /usr/local/deqn -Pdp
2 /usr/local/ditroff -Pdp -ms
3 /bin/cc /usr/harry/programfile.c -lm -O -o /usr/harry/programfile.out
```

5.6.2. Pipes

A feature of the Unix shell is piping, in which the output of one command becomes the input of another command. Piping is arranged by the shell, without the command processes themselves taking any action; indeed, processes are not even aware that their input or output goes to or comes from a pipe. When commands are executed remotely through the load balancer, the LBRM is responsible for setting up the execution environment. In Unix, for commands connected by pipes, setting up the execution environment must include setting up pipes, because the command processes themselves are not aware of the pipes.

Shell level pipes were formerly implemented using an operating system feature also called pipes. In 4.1 and earlier versions of Berkeley Unix, pipes between processes had to be created by a common ancestor process; usually this was a shell. In 4.2 and later versions, pipes are merely pairs of sockets, which can communicate if they know each other's port number. The port number can be handed down from a common ancestor, or it can be shared in some other manner.

In order for the LBRM to set up pipes between tasks, it must know which tasks are to communicate by pipes. This information must come from the originating user process. If the originating user process is a shell, then it must get the information from the command line typed by the user. The communication section (defined above) does not meet the needs of piping for several reasons. First, the communication section lists all communication between tasks, whereas pipes are only one type of communication. Pipes are established by the parent shell and processes are unaware of them, but task processes can communicate by establishing communications themselves (by creating sockets, sending messages, and so on). So tasks may be listed in the communication section even if they have nothing to do with piping. Second, the information supplied by the communication section is fairly general, it says only that a task is likely to communicate in some manner with another task. To establish piping, the LBRM must know which task's output is to become which other task's input. So when tasks are to be piped, the communication section does not supply all the necessary information. Finally, a communication section can only refer to tasks listed in the associated tasks section or execute section, and the execute section sent to an LBRM lists only tasks to be executed on that LBRM's machine, but the source or sink of a pipe might be a task sent to a different machine. In order to supply the LBRM with the information needed to establish pipes, the Unix implementation of the load balancer will add to LBRM requests (requests sent by user processes to LBRMs) a special pipes section. The format of the Unix pipes section is defined next.

The Unix-specific pipes section begins with a section-header line containing

```
UPIP
```

(for "Unix PIPes"). Following this is a series of lines, each line describing a pipe the LBRM is to set up.

There are two kinds of pipes the LBRM must deal with: (a) local pipes between two tasks to be executed on the LBRM's machine; and (b) nonlocal or intermachine pipes between a task to be executed on the LBRM's machine and a task to be executed on another machine. A nonlocal pipe is either an incoming pipe or an outgoing pipe, that is, either the input to a task to be executed on the LBRM's machine comes from a pipe from some other machine, or the output from a task to be executed on the LBRM's machine goes to a pipe to some other machine (Unix shell-level pipes carry information in only one direction).

In the case of a local pipe, both ends of the pipe are tasks listed in the execute section of the LBRM request. It is sufficient in this case to specify that the pipe is local and to identify the tasks to be connected by the pipe:

```
LOCL 2 3
LOCL 5 8
```

Order is significant and denotes the direction of the pipe: "LOCL 2 3" means the output of task 2 is to be piped as the input to task 3. Nothing special is done to the non-piped inputs and outputs; in the example above, to the input to task 2 and the output from task 3. Multi-stage local pipelines are allowed, and are denoted in the following manner:

```
LOCL 2 3
LOCL 3 4
```

In the case of a nonlocal pipe, for each of the two LBRMs involved, one end of the pipe is a task listed in the execute section, but the other end is an intermachine communication path. The two LBRMs must establish the communication channel between them. (Clearly it is undesirable to have the communication flow through the originating machine.) If there were some sort of rendezvous server or switchboard facility, then communication could be established by that means. Unfortunately, Unix does not provide any such facility, so, in the Unix environment, the load balancer must establish communication by an *ad hoc* method. The originating user process will send to each of the two LBRMs at the ends of the pipe the name of the other machine and a rendezvous identifier. The rendezvous identifier is used to make sure that, of several possible connections being established between two machines, the two pipe ends that are connected are the two that are supposed to be connected. The rendezvous identifier must be unique on the two machines until the pipe is established; it may be easier to make it globally unique for all time. The rendezvous identifier could be generated from some combination of the originating machine name or id, the originating user process's id, a sequence number maintained by the originating user process, and a timestamp (in Unix the number of seconds since midnight, January 1, 1970 is returned by the "time" and "gettimeofday" system calls). So a line in the pipes section sent to an LBRM for a nonlocal pipe must include the following information: (a) the fact that a nonlocal pipe is to be established, denoted by the keyword NLCL; (b) whether the task on the LBRM's machine is to be the source of the pipe or the sink, denoted by one of the keywords SORC for the source end or SINK for the sink end; (c) the identity of the local task, that is, the number of a task in the execute section; and (d) the information needed to establish the communication channel for the nonlocal end of the pipe, that is, the name of a machine and a rendezvous identifier. For example,

```
NLCL SORC 4 monet ernie.3788.1.546496078
NLCL SINK 6 joshua ernie.3788.2.546496282
```

The LBRM at the source end of the pipe decides whether it is willing to accept the source-end task, but does not reply to the user process yet. If it is willing to accept the task, it creates an endpoint for communication; in Unix, this endpoint is called a *socket*, and in the internet protocol used by Unix a socket can be identified by a machine address and a *port number*. The source-end LBRM then sends a message to the LBRM at the sink end (LBRMs are at "well-known addresses" on each machine, which in Unix means that the port number can be looked up in the "services" database by the "getservbyname" system call), including the rendezvous identifier and the port number. The LBRM at the sink end connects to the specified port and sends back the rendezvous identifier and ACPT (for ACCEPT), and the pipe communication channel is established. The two LBRMs then send acceptance messages back to the originating user process, create execution environments for the two tasks, manipulate the input or output of the environments so that they use the pipe, and cause the tasks to execute in the environments.

If the source-end LBRM is not willing to accept the task it sends a refuse message to the originating user process, and it does not create the port.

If the sink end LBRM refuses the task that was to have been the sink end of the pipe, then it sends a refuse message back to the originating user process, and when it receives the pipe-establishing message from the source end LBRM, it can either send back a refuse message (consisting of the rendezvous identifier and RFUS) or ignore the message. The source-end LBRM will take the pipe to have been refused either if it receives a refuse message or if the pipe-establishing message is not replied to within some timeout period. If the pipe is refused, then the source-end LBRM sends a refuse message back to the originating user process with an error indication that the task was refused because the pipe could not be established to the specified sink-end.

When the sink-end LBRM receives the request with the pipe section, it decides whether it is willing to accept the task. If not, then, as described above, it sends a refuse message back to the originating user process, and, when it receives the pipe-establishing message from the source end LBRM, it can either send back a refuse message or ignore the message. If the sink-end LBRM is willing to accept the task, it does not reply to the user process yet, but waits for the pipe-establishing message. If the message does not arrive within a reasonable time, the sink-end LBRM sends a refuse message back to the originating user process, with an error indication that the task was refused because the pipe could not be established to the specified source-end.

In the approach we have chosen, if either end of the pipe refuses its task, both ends report refusal to the originating user process and forget about the tasks. The user process, if it still wants the tasks executed, must then choose a new machine for the refusing end, possibly with the help of the LBSM, and send new messages to the machines at both ends, including the machine that was willing to accept the task. Another approach would be to have the accepting machine "hold" its task and wait for the user process to send a "revise pipe" message naming a new other end. This alternate approach might be more efficient in the case of refusals, but it would greatly complicate the situation. There would have to be protocols, and code, to handle such cases as the holding LBRM, or the machine it is running on, crashes, or the originating process, or the machine it is running on, crashes. We feel that the greater complexity is not worthwhile; efficiency is increased only for refusals and refusals should be rare, so the increase in efficiency would be small. The added complexity of the design and of the code would make it more expensive to implement and maintain the load balancer, and would increase the likelihood of problems.

5.7. Receive Manager Replies

When the LBRM receives an execution request, it decides, for each task, whether to accept the task or refuse to accept it. In the present design, the LBRM considers each task separately, it does not make acceptance decisions on groups of tasks. The LBRM informs the requesting user process of its decisions by sending a message which lists, for each task, whether the task is accepted or refused, and, for each refused task, an indication of the reason for refusal. (The exact format of the reason for refusal remains to be determined.)

Reply messages from the LBRM contain a message header followed by an acceptance section. The acceptance section begins with a section-header line containing the keyword ACPT (for ACCEPTANCES) and continues as shown in the following two examples:

```
FROM LBRM vangogh
TOxx USER ernie
ACPT
0 ACPT
1 ACPT
ENDx
```

```
FROM LBRM vangogh
TOxx USER ernie
ACPT
0 ACPT
1 RFUS reason
ENDx
```

In the first example, both of the tasks submitted in the execution request have been accepted. In the second example, the first task has been accepted and the second task refused.

If an execution request includes several tasks and they are all accepted, then the LBRM sends an acceptance reply and executes all the tasks. If they are all refused, the LBRM sends a refusal reply. If, however, some of the tasks are accepted and some are refused, then a more complex situation has arisen.

It may be that the user process originating an execution request which includes a number of tasks wants all the tasks executed on the same machine (perhaps on the LBSM's advice). In other words, if any tasks are refused, none of them should be executed. We considered a number of approaches for handling this situation. One possibility was to have the LBRM in this situation hold the accepted tasks without executing them and ask the originating user process whether to execute or not. The user process would have to reply with lists of which tasks to execute and which to drop. Another possibility was to add a notation for "all-or-nothing" groups to all execution requests. These approaches have drawbacks: they cost extra time, they complicate and hence slow down message processing, they complicate the code and make faults more likely and maintenance more difficult, they lower reliability (what if the originating machine crashes while the LBRM is waiting for instructions about what to do with "held" tasks?). We believe that most execution requests will contain single tasks and that most multiple-task execution requests will either be completely accepted or completely refused. Accordingly, we have dealt with this problem in the following simple manner. If this approach turns out to be inadequate, one of the more complex alternatives can be implemented. In our simple approach, an LBRM that has received a multiple-task execution request and refuses any task will refuse all tasks, even if the remaining tasks are otherwise acceptable. The *reason* returned for the otherwise acceptable tasks will indicate that they were acceptable but were refused only because they were part of a multiple-task execution request of which one or more other tasks were refused. The originating user process can then resubmit the execution request without the unacceptable task(s), or it can attempt to find another machine that will accept the whole set of tasks.

This concludes the discussion of messages to and from the LBSM and LBRM. We turn now to messages to and from the LIM and the TIM.

5.8. Load Information Manager Communication

The LIM accepts three types of queries: local, one-machine, and all-machine queries. Local queries request load information for the machine on which the LIM is running. One-machine queries request the load information for a single named machine. All-machine queries ask for the load information for all machines for which the LIM has data, in other words, for all the data the LIM has. The queries have the formats shown:

local	one-machine	all-machine
FROM USER *machine-name*	FROM USER *machine-name*	FROM USER *machine-name*
TOxx LIMx *machine-name*	TOxx LIMx *machine-name*	TOxx LIMx *machine-name*
LOCL	MACH *machine-name*	ALLx
ENDx	ENDx	ENDx

The reply begins with one of

local	one-machine	all-machine
FROM LIMx *machine-name*	FROM LIMx *machine-name*	FROM LIMx *machine-name*
TOxx USER *machine-name*	TOxx USER *machine-name*	TOxx USER *machine-name*
LOCL	MACH *machine-name*	ALLx

followed by the appropriate information or an error message, followed by ENDx.

The obvious method for the LIMs to use to distribute load information is for each LIM to send an update to every other LIM every interval. This approach would work in any network supporting point-to-point messages. If broadcast or multicast is available, then clearly a certain amount of overhead in message sends can be saved. An unsophisticated use of broadcast would be the flooding broadcast approach with some sort of border constraints. A highly sophisticated approach would have the LIM obtain a list of peer machines, perhaps from a configuration file, then find out either from a configuration file or from a network manager or routing manager which machines could be reached by broadcast or multicast and which could not. In a different vein, instead of sending out updates at regular time intervals, the LIM could send out updates whenever, but only when, the load has changed significantly.

We plan to do two implementations of the LIM, one using point-to-point messages and the other using single-broadcast. (Actually it will be one implementation with the communication mechanism selected by a compile time option.) Single-broadcast will be less expensive, but will only work if all machines participating in load balancing are on the same cable and the network supports broadcast. Point-to-point will be able to function in any type of network and with any network topology.

5.9. Task Information Manager Communication

As described above, queries to the TIM specify a program by giving a full path name, and as much information about flags and arguments as is available. The TIM replies with information about the task characteristics of that program with those flags and arguments, if they are known. If the flags and arguments are not known, the TIM sends a "low confidence" or "approximation" reply telling what is known about the task, perhaps its characteristics with different flags or arguments. If nothing is known about the task, the TIM sends a "no information" reply.

Further specification of the TIM and TIM communication protocols and formats awaits construction of, and experimentation with, a prototype TIM.

6. CONFIGURATION INFORMATION

In order for the load balancing system to work to the satisfaction of the users and administrators of the machines involved, it must be given instructions concerning which tasks should be sent to remote machines, where they should be sent, which tasks should be accepted from remote machines under what circumstances, and so on. This information is called *configuration information* (others might call it *access control.*)

A load balancer without configuration information might be technically successful in that it could make good decisions and carry them out efficiently, but the question remains, would people be willing to use it? An administrator who runs a load balancer without configuration information loses control of the machine, because the load balancer will move tasks onto the machine and onto other machines with no administrative constraints. As there are nearly always administrative constraints which must be observed, very few people would be willing to run such a load balancer on their machines, and software that no one will use is useless. We believe that the success of a load balancer, indeed of any software project, depends at least as much on the administrative aspects of installing and using the software as it does on technical strengths and weaknesses. (This is somewhat akin to the idea that it is important to provide a well-designed user interface as well as good functionality in a software product.) Accordingly, our design includes fairly extensive configuration information.

Configuration information concerning sending tasks to remote machines is called *sending configuration information*; configuration information concerning receiving tasks from remote machines is called *receiving configuration information*.

The load balancing managers read configuration information from files called *configuration files*, in particular, the *sending configuration file* and the *receiving configuration file*.

6.1. Sending Configuration Information

When an LBSM begins execution it has no information, not even a list of names of machines to which it might send tasks or which it might query for acceptance hints (described below). The LBSM reads its *sending configuration information* from the *sending configuration file*.

The LBSM makes three kinds of decisions: unload decisions, eligibility decisions, and placement decisions, so the sending configuration information is divided into three sections: unload information, eligibility information, and placement information. Each of the three could be very complex: the unload information could specify a variety of load limits and times of day; the eligibility information could specify that only certain commands from certain users at certain times of day or under certain load conditions are eligible for load balancing; the placement information could specify which tasks from which users should be sent to which machines at particular times of day or under particular load conditions, and so on. The sending placement information would have to be combined with the acceptance hints from receiving machines (see below). It is not clear that all this complexity is necessary or useful, and allowing all this complexity in making sending decisions would increase the amount of time required to make the decisions, which is exactly what we are seeking to minimize. For these reasons we have adopted a design with simpler sending configuration information. In circumstances where more complex sending configuration information would be useful, a more complex design could be adopted.

In our design, the sending configuration information is made simpler by restricting the kinds of sending configuration information in each of the three sections. In the unload information section, a machine administrator is allowed to specify only a single range of a single load index. For example, in Unix, the administrator could choose the Unix one minute load average for the load index and "four or greater" for the range (that is, if the Unix one minute load average is four or greater then tasks should be unloaded, otherwise they should not be unloaded). Eligibility information is allowed to consist only of a list of commands eligible for load balancing. Placement information is allowed to consist only of a list of machines which might be willing to accept tasks.

In addition to the sections for unload, eligibility, and placement information, an LBSM maximum update period for acceptance hints is specified in the sending configuration file. See "Acceptance Hint Periodic Update," below, for further discussion.

6.1.1. Sending Configuration File Format

As opposed to the inter-component messages, which are normally never seen by humans, the configuration files are meant to be written, read, and maintained by administrative personnel (presumably human). The formats of the configuration files therefore are not strict about extra spaces, allow either upper or lower case, and do not insist on four-character keywords; comments are allowed, too. The configuration files begin with a line identifying what type of file it is, sending information or receiving information. The sending configuration file then continues with sections for unload, eligibility, and placement information. The format of the sending configuration file is as follows. For clarity, we have shown keywords in upper case and items to be supplied by the administrator in lower case, but in fact, case is ignored when the configuration file is read.

```
# comments from '#' to end-of-line
LBCONFIG SEND
UNLOAD
TOTALLOAD index:min-max
ELIGIBLE
COMMAND command command ...
PLACEMENT
MACHINE machine machine ...
UPDATEPERIOD
minutes
END
```

See the "TOTALLOAD" section under "Receiving Configuration File Format," below, for the format of index, min, and max.

An administrator editing the file can choose the load index and its range, the commands eligible to be off-loaded, and the machines to which they can be off-loaded. Naturally, each machine can have different configuration information, including a different load index for unload, but the LIMs on all machines will handle the same set of load indices, and the load index specified must be one of those reported by the LIM.

When the LBSM begins execution, it attempts to read the sending configuration information file. If for any reason it cannot, it generates an error message. Until it knows of at least one machine which might be willing to accept tasks, the LBSM responds to all queries from user processes with LOCL (for LOCAL), in effect turning off load balancing.

When the LBSM has a list of machines which might be willing to accept tasks (candidate machines), it tries to obtain acceptance hints for each candidate machine on the list. It obtains these hints by sending a query to the LBRM on each candidate machine. If the LBRM on a candidate machine ever refuses a task, the LBSM may query that LBRM for new acceptance hints (see Acceptance Hints, below).

6.2. Receiving Configuration Information

A receiving machine must make the final, authoritative determination of whether to accept an offered task, otherwise the administrator responsible for a machine could lose control of what work is executed on the machine. This determination must be made at the time the task is offered to it by the sending machine, because the administrator might have changed the rules since the sending machine's last update. Thus, receiving configuration information must be available to each receiving machine.

When the LBRM starts execution, it tries to read the receiving configuration information file. If it cannot read the file, it issues an error message and refuses all tasks. The reason for the receiving configuration information is to prevent a machine from accepting and processing tasks it should not process. Without receiving information the LBRM does not know which tasks to accept, so, to be safe, it does not accept any.

6.2.1. Receiving Configuration File Format

The format of the receiving configuration file is

```
# comments from '#' to end-of-line
LBCONFIG RECEIVE
MACHINE machine machine ...
     NONE
     USER username username ...
     ALLUSERS
     NOTUSER username username ...
     TIME start-end start-end ...
     ALLTIMES
     LOCALACCT
     LOCALGROUP groupname groupname ...
     TOTALLOAD index:min-max index:min-max ...
     REMOTELOAD index:min-max ...
     COMMAND command command ...
     ALLCOMMANDS
     NOTCOMMAND command command ...
MACHINE machine machine ...
     ...
END
```

The file is made up of any number of acceptance-rule groups. Each acceptance-rule group starts with a MACHINE line that names one or more machines to which the acceptance-rules in the group apply. It is an error for a machine name to appear in more than one acceptance-rule group: to reduce the possibility of confusion about what is being allowed, all the acceptance rules for a machine can be looked at in one place only. (The alternative would be to allow machines to be named in more than one acceptance-rule group, with the final acceptance criteria applying to the machine being the union of all permissions specified, and the most lenient permission in case of overlap.)

Following the MACHINE line there are one or more lines specifying acceptance rules, one rule per line. The administrator of a machine may specify various combinations of acceptance rules; for a task to be accepted, it must meet all the acceptance rules in the appropriate acceptance-rule group. The receiving configuration file specifies allowed acceptances, not limitations; anything not specifically allowed is prohibited. The LBRM will not accept a task unless it has been specifically allowed to. We do not take the more lenient approach in which the LBRM accepts all tasks unless it has specifically been told not to. The following describes each of the available acceptance rules.

NONE
: no tasks are to be accepted from the applicable machines. The same effect can be obtained by not mentioning the machines at all, but this provides a way to explicitly bar certain machines.

USER
: only the named users are allowed to send tasks from the applicable machines.

ALLUSERS
: all users on the specified machines are allowed to send tasks. Either USER or ALLUSERS must be specified in each acceptance group.

NOTUSER
: the named users are not allowed to send tasks from the applicable machines. Useful in conjunction with ALLUSERS: NOTUSER overrides ALLUSERS to give an effect of "all except ...."

TIME
: tasks from the applicable machines are only accepted during the specified times of day. Times of day are in the format *hhmm*. where *hh* specifies hours in the range 0-23 and *mm* specifies minutes in the range 0-59.

ALLTIMES
: tasks from the applicable machines are accepted at any time of day. Either TIME or ALLTIMES must be specified in each acceptance group.

LOCALACCT
: tasks will be accepted only from users on remote machines who also have accounts (logins, user-ids) on the local machine. (Note that it is possible for there to be a user on a remote machine with user-id "joe" while a different user has user-id "joe" on the local machine. User-ids should be consistent across machines before using LOCALACCT.)

LOCALGROUP
: this item is strongly Unix-flavored: Unix users are members of one or more *groups*. If one or more LOCALGROUPs are named, then a user on a remote machine must have a local account, and the local account must be in one of the specified local groups. LOCALGROUP implies LOCALACCT.

TOTALLOAD
: tasks from the applicable machines are accepted only if the load on the local machine is within specified limits as measured by specified indices. This is the total load on the local machine due to both locally originated tasks and remotely originated tasks. *Index* names a load index and *min* and *max* specify the range of values within which tasks will be accepted, either min or max (but not both) may be "*" which means "any value." A Unix example would be

```
TOTALLOAD LA1:*-4 nusers:*-20 freemem:2000-*
```

which specifies that tasks are to be accepted only if: (a) the Unix one minute load average is in the range from anything up to four, and (b) the number of users logged in is in the range from anything up to 20, and (c) the amount of free memory is 2,000 blocks or more. The load indices specified must be among those reported by the LIM.

REMOTELOAD
: tasks from the applicable machines are accepted only if the load on the local machine due to tasks accepted from remote machines is within specified limits as measured by specified indices. *Index*, *min*, and *max* are as in TOTALLOAD. In the current design, the only index allowed is `ntasks`, which is the number of remote tasks that have been accepted but have not yet completed execution. Note that the LBRM itself maintains the value of ntasks, it does not get it from the LIM. Note also that ntasks is the total number of remote tasks executing, not just the number from the machines in the current acceptance-rule group; there is no way to set max for one machine or group. There is also no way to set a cumulative or periodic ntasks limit (for example, 20 tasks per hour or per day).

COMMAND
: only the named commands are accepted from the applicable machines.

ALLCOMMANDS
: all commands are accepted from the applicable machines (subject to other acceptance rules). Either COMMAND or ALLCOMMANDS must be specified in every acceptance group.

NOTCOMMAND
: the named commands are not accepted from the applicable machines. Useful in conjunction with ALLCOMMANDS: NOTCOMMAND overrides ALLCOMMANDS to give an effect of "all except ...."

6.3. Acceptance Hints

If a sending machine were to send a task to a machine which refused to accept it, that would cost time and delay the eventual completion of the task. Let us call this a *refused send*. Refused sends are probably costly enough, in terms of lost time, that the sender should try to avoid them, even at the cost of increased complexity and hence longer decision times in the sending machine. Without such avoidance the sender might encounter several refusals before finding an accepting machine (or giving up and processing the task locally). So, clearly, sending machines should know something about the likelihood of acceptance. This can be accomplished by having sending machines know something about receiving machines' receiving configuration information.

Receiving configuration information on the receiving machine must be authoritative, whereas information on the sending machine can be incorrect. Incorrect information on the sending machine may result in (a) excluding from consideration machines which would accept tasks, or (b) in refusals and hence delay, but it will never result in a machine processing a task it should not process (because the receiving machine will use the authoritative configuration information to correctly accept or refuse the offered task).

As a practical matter, it is extremely difficult to manually keep multiple copies of information up-to-date with each other. Trying to manually maintain copies of each receiving machine's configuration information on each sending machine or on each shared file area would no doubt be a frustrating and futile exercise.

To solve this problem the following design approach is used. Receiving configuration information for each receiving machine is kept on that machine; this receiving configuration information is authoritative. Sending machines obtain some receiving configuration information for machines they are likely to send to. This information is obtained from the LBRM on the receiving machine. Receiving configuration information for candidate machines held on sending machines is considered *hints*, that is, probably correct, but not necessarily correct. Since acceptance hints will usually be correct, sending machines should seldom ignore usable machines or encounter refusals. Even when refusals are encountered, they will not cause incorrect functioning. A refusal may cause the incorrect hint to be updated, reducing the chances of further refusals (see below).

6.3.1. Acceptance Hints at Start-Up

The LBSM has in its configuration file a list of machines which might be willing to receive tasks. When the LBSM begins execution, it sends a query to the LBRM on each machine on the list, asking for acceptance hints.

6.3.2. Acceptance Hint Updates Due to Refusals (Hard and Soft Refusals)

Whenever a task is refused by a receiving machine, the LBSM of the sending machine may send a query to the receiving machine's LBRM asking for new acceptance hints. When an LBRM refuses a request, it may be a "hard" refusal, that is, one that will not change, or a "soft" refusal, which may change very soon. When a request is refused because the originating user is not allowed on the receiving machine, that is a hard refusal; if the same request were resubmitted some time later it would still be refused (unless the LBRM got new configuration information). When a request is refused because a load index on the receiving machine is outside the allowed range, that is a soft refusal; if the same request were resubmitted some time later it might be accepted.

When refusing a request, the LBRM gives the reason for refusing (see Receive Manager Replies, above). Based on the reason supplied, the LBSM can classify refusals as hard or soft. When a request is refused for a hard reason, that means that the sending LBSM's acceptance hints for that candidate machine are incorrect; the LBSM, when informed of the refusal by the user process, will send a query to the refusing LBRM asking for an update on acceptance hints. A soft refusal does not necessitate an acceptance hint update.

(Note that a refusal due to time of day is a hard refusal. The sending LBSM's acceptance hints for the refusing candidate machine must indicate that the candidate is willing to accept tasks at that time of day, or else the LBSM would not have placed the task on that candidate. If the candidate is in fact not willing to accept tasks at that time of day, then the LBSM's hints are wrong and should be updated.)

6.3.3. Acceptance Hint Periodic Updates

In order to ensure that its acceptance hints are up-to-date, the LBSM queries the LBRM of each candidate machine on its candidate machine list (see Sending Configuration Information, above) at least once every period of time. The LBSM records, for each candidate machine, when it last received an acceptance hint update. Updates are obtained from all candidates when the LBSM begins execution and when it is reconfigured (see below). An update is obtained from a particular candidate when the LBSM encounters a hard refusal from that candidate. If an update from a particular candidate has not been obtained for some other reason for an entire time period, the LBSM queries that candidate's LBRM for acceptance hints.

The importance of periodic updates can be illustrated by the following scenario. The local LBSM has acceptance hints which indicate that a certain candidate machine is not willing to accept tasks from the local machine. In fact, the candidate has recently been given new receiving configuration information and is now willing to accept tasks from the local machine. Without periodic updates, the LBSM will not discover that the candidate is now available until the LBSM is restarted or reconfigured. Since the LBSM believes that the candidate is unavailable, it will not place tasks there, and there will be no updates due to refusals. Clearly, it is undesirable for the LBSM to continue to exclude from consideration machines which are willing to accept tasks, so periodic updates are important.

Choosing the length of the update period involves a tradeoff between the cost of obtaining updates and the penalty of ignoring usable machines. Since the importance of these two factors will be different in different environments, the optimal update period will also be different in different environments. Therefore we allow the machine administrator to set the update period. The sending configuration file (see above) contains a section labelled "UPDATEPERIOD" which consists of a single line specifying the length of the update period. In our planned implementation, the length of the period will be specified by an integer number of minutes.

6.3.4. Acceptance Hint Version Stamps

To improve efficiency and reduce overhead, LBSMs and LBRMs will use *version stamps* with acceptance hints. The LBRM will generate or obtain the appropriate version stamp for its acceptance hints when it starts up and whenever it is reconfigured (see below). The LBSM will keep a version stamp associated with the latest set of acceptance hints it has received from each candidate machine. The LBSM will send the old version stamp when it queries an LBRM for new hints, and the LBRM will either reply that the old version is still current or send a new set of hints with a new version stamp. In the Unix implementation, the version stamp will be the last-modified time of the receiving configuration file. This should improve the efficiency of the load balancing managers and reduce communication overhead by reducing the sending of duplicate sets of acceptance hints.

6.3.5. Alternatives

If the LBSM's machine and the candidate machine both have access to the same file system, the LBSM could try to read the receiving configuration file directly. This would take some load off the LBRMs, but it would complicate the code of the LBSMs. Also, there are some questions of appropriateness. We have decided not to include this in the design, but if acceptance hint queries and replies turn out to be a heavy cost, the issue might be explored further.

We considered having LBRMs send acceptance information to potential senders as follows:

> send configuration information to all likely senders at start-up and whenever the information changes; send configuration information to a particular sender whenever a task from that sender is refused or in response to a query

We decided against this for the following reasons: first, it is the LBSM's responsibility to gather and use information needed for making sending decisions; the design is simpler and cleaner if that responsibility is kept in the LBSM and not partially spread into the LBRM. Second, the administrator on the sending machine must have control over where his machine sends tasks; the LBSM should not consider sending tasks to other machines even if they declare themselves available, unless those machines are in the sending machine's sending configuration information.

7. OTHER MATTERS

7.1. Load Balancer Control Program

The load balancer consists of the LBSM, the LBRM, the LIM, and the TIM. A machine which only unloads tasks to other machines need not run the LBRM; a machine which only receives tasks (a compute server) need not run the LBSM and may not need to run the LIM or the TIM. An administrator may wish to start or stop load balancing, to reconfigure one or more of the components, or to check the status of one or more of the components. To make it easier for administrators to do these things, we plan to provide a program called the *load balancer control program*, or *lbc*, which will interactively accept commands to do these things.

Lbc will also accepts commands on the command line. So it will be possible to use lbc to start all the load balancing components at machine boot time from a script of boot commands (in Unix, from the `/etc/rc.local` file).

The lbc program will look for a file named `lbrc` containing information about which components to run by default, the locations of the executables for each component, the locations of the configuration files, and perhaps information about security level, debugging level, an acceptance level, and so on, at which the components should run. ("`Lbrc`" stands for "load balancer reboot commands.")

7.2. Files

The LBSM and the LBRM will accept command line arguments specifying the files to use as configuration files; in Unix these will be of the form "-f *filename*". If there is no command line file argument, then the lbrc file will be checked. If the lbrc file does not specify a configuration file (sending or receiving, respectively), then a compiled-in default file name will be used. If the default file does not exist, then the manager reports an error.

In Unix, most system administrative information is kept in the directory `/etc`. The default file names for the load balancer configuration and administration files are:

lbrc	`/etc/lbrc`
send configuration file	`/etc/lb.config.send`
receive configuration file	`/etc/lb.config.receive`

These default names are compile-time constants and can be changed by editing the source files and recompiling.

7.3. Reconfiguration

It may be useful to be able to reconfigure the components of the load balancer without terminating and restarting them. Reconfiguration means the old configuration information is discarded, and the configuration file is read and processed as described above.

The planned Unix implementations of the LBSM, the LBRM, the LIM, and the TIM will reconfigure when they receive the hangup signal (SIGHUP). Reconfiguration can be accomplished through the lbc program, which will send the signal.

ACKNOWLEDGEMENTS

Many thanks are due Domenico Ferrari for encouragement, suggestions, and support. Thanks also to David Anderson, Keith Bostic, Larry Carter, Mike Karels, Kirk McKusick, Mike Meyer, Colin Parris, Joe Pasquale, Stuart Sechrest, Mark Sullivan, Stefano Zatti, and Songnian Zhou, for comments, discussions, and suggestions.

8. REFERENCES

Brian Bershad, Load Balancing with Maitre d', UCB/CSD 85/276 (PROGRES Report No. 85.18), Computer Science Division, University of California, Berkeley, December 1985.

S. I. Feldman, Make - A Program for Maintaining Computer Programs, *Software—Practice & Experience 9*, 4 (April 1979).

Anna Hac and Theodore Johnson, A Study of Dynamic Load Balancing in a Distributed System, Report JHU/EECS-85/15, Department of Electrical Engineering and Computer Science, Johns Hopkins University, 1985.

Morton D. Hoffman, William I. MacGregor, Richard E. Schantz, and Robert H. Thomas, Cronus, A Distributed Operating System: Functional Definition and System Concept, Report No. 5041, Bolt Beranek and Newman, Inc., June 1982.

Kai Hwang, William J. Croft, George H. Goble, Benjamin W. Wah, Faye A. Briggs, William R. Simmons, and Clarence L. Coates, A Unix-Based Local Computer Network with Load Balancing, *Computer Magazine 15*, 4 (April 1982), 55-66.

Miron Livny and Myron Melman, Load Balancing in Homogeneous Broadcast Distributed Systems, *Proceedings of the ACM Computer Network Performance Symposium*, April 1982, 47-55.

Harry I. Rubin, Load Balancing Considerations in the Design of an Operating System, (in preparation).

R. Schantz, B. Woznick, G. Bono, E. Burke, S. Geyer, M. Hoffman, W. MacGregor, R. Sands, R. Thomas, and S. Toner, Cronus, A Distributed Operating System: Interim Technical Report No. 2, Report No. 5261, Bolt Beranek and Newman, Inc., February 1983.

Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, The LOCUS Distributed Operating System, *Proceedings of the 9th Symposium on Operating System Principles*, October 10-13, 1983, 49-70.

Songnian Zhou, A Trace-Driven Simulation Study of Dynamic Load Balancing, UCB/CSD 87/305 (PROGRES Report No. 86.4), Computer Science Division, University of California, Berkeley, September 1986.

Songnian Zhou and Domenico Ferrari, An Experimental Study of Load Balancing Performance, UCB/CSD 87/336 (PROGRES Report No. 86.6), Computer Science Division, University of California, Berkeley, January 1987.