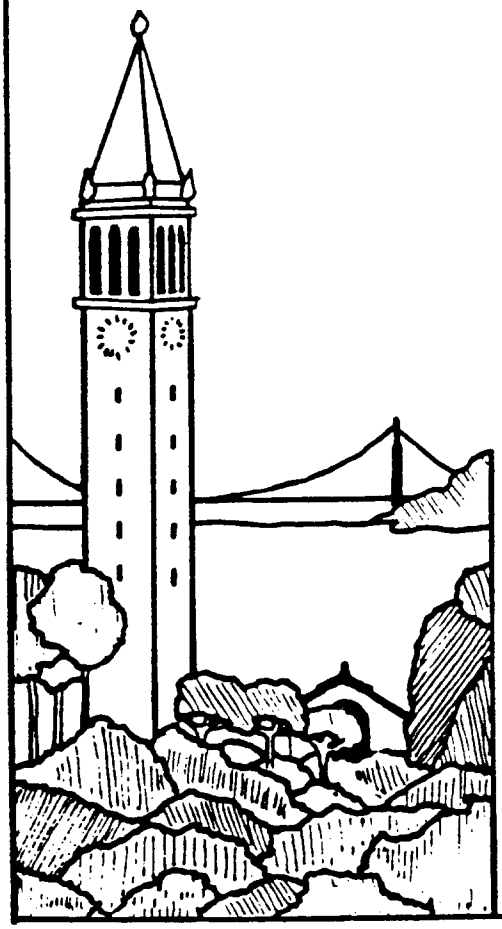
An Experimental Study of Load Balancing Performance

Songnian Zhou and Domenico Ferrari



Report No. UCB/CSD 87/336

January 1987
PROGRES Report No. 86.8

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

An Experimental Study of Load Balancing Performance

Songnian Zhou and Domenico Ferrari

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley†

ABSTRACT

The design and implementation of a prototype load balancer on a loosely-coupled distributed system are discussed, and the results of a large number of measurement experiments performed on the system under artificial workloads we constructed using frequently executed system commands are presented. The impacts on the system's performance of the load balancing algorithms, as well as of the values of their adjustable parameters, and of the various types of workloads, are evaluated. The effects of load balancing on the performances of individual hosts and on each type of job are also quantitatively investigated using measurements. The results of our study show that automatic load balancing at the job level can have very beneficial effects on the mean and standard deviation of job response times while causing little overhead and requiring no modification to the system kernel or to applications programs. This is the case even when only a relatively small fraction of the jobs can be executed remotely, and the reduction in response time is uniform across all job types, including those that are not moved for execution to another machine.

1. INTRODUCTION

As distributed computing systems become increasingly popular, resource sharing among a number of computers connected by communication networks becomes feasible and desirable. Some of the possible resources to share are computing power, data, and hardware devices. The sharing of computing power is usually in the form of *load balancing*, or *load sharing*‡, and has been studied in various forms for several years. To be more

† This work was partially sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089, and by the National Science Foundation under grant DMC-8503575. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

‡ The term load balancing has sometimes been used to imply the objective of equalizing the loads of the hosts, whereas load sharing has simply been taken to mean a redistribution of the workload. We will use the term *load balancing* in the rest of this paper, but not necessarily with the stronger connotation.

specific, load balancing is the process of redistributing the workload submitted to a network of computers to avoid the situation in which some of the hosts are congested, while others are underloaded. As a result, system performance, e.g., job throughput rate or job response time, is likely to improve.

A variety of approaches have been employed to study load balancing, including graph-theoretical methods [Stone77, Stone78, Bokhari79, Wu80], queueing network analysis [Chow79, Livny82, Rama83, Wang85, Eager86a, Eager86b], simulation [Bryant81, Livny82, Leland86, Zhou86], experimental implementation [Hwang82, Bershad85, Hagmann86, Ezzat86], and measurements [Ezzat86]. In [Zhou86], one of the authors describes a trace-driven simulation study of load balancing. In that study, job information collected from production systems was used to drive the simulator of a distributed system containing a number of hosts. Load balancing was found to be very effective in reducing both the average and the standard deviation of job response times. Comparisons were also made among the performances of a number of load balancing algorithms. Encouraged by the results of this study, and in order to gain further insights into a number of issues, we implemented load balancing on a cluster of diskless SUN-2 workstations running in our distributed systems laboratory under the SUN/UNIX† operating system, connected by an Ethernet and supported by file servers. This experimental environment enables us to conduct measurement studies of load balancing.

Our purposes in implementing a prototype load balancer and doing measurements on it were several-fold. First, we were interested in investigating the feasibility of load balancing in a UNIX environment. We wanted to experiment with various load indices proposed by other researchers as well as by us, in order to determine one or a family of load indices suitable for load balancing. We wanted to compare, in a more realistic environment, the algorithms we had studied using simulation in order to validate our results, and to tune our simulation model so that it can be used, with more confidence, in exploring parts of the design space unreachable by our implementation. We also wanted to assess quantitatively the amount of overhead introduced by load information exchanges and job transfers between the hosts. Finally, we hoped that our experimental implementation can serve as a base for a production load balancing system.

While a number of implementations of load balancing have been reported, as will be discussed in the next section, measurements have not been used as an important approach to the study of load balancing performance. In [Bershad85], some comparisons of the job response times before and after the installation of the load balancer were provided. Hagmann presented a set of measurements of the costs of remote job executions [Hagmann86]. Ezzat reported some performance figures of load balancing in the NEST system with 3 hosts driven by benchmark programs [Ezzat86]. We intended to use measurements as our primary method of study in this research.

The design and implementation of our load balancing system, and some of the measurement experiments we performed on it, are reported in this paper. The important results

---

† UNIX is a trademark of AT&T Bell Laboratories.

from our study include the following:

- transparent, flexible load balancing at the job level can be achieved at a low cost, and without modifying either the system kernel or any of the existing application programs;
- load balancing is capable of substantially reducing the mean of the process response times (up to 30-40%), and their standard deviation (up to 40-50%), especially when the system is heavily loaded, and/or the instantaneous loads on the hosts are appreciably unbalanced;
- a number of "reasonable" load balancing algorithms using periodic load information exchanges or acquiring such information on demand produce comparable performance improvements;
- load balancing can still be highly effective when only a small fraction of the workload (down to 20%, in terms of CPU time consumption) can be executed remotely;
- the relative (percentage) reduction in response time is uniform across all classes of jobs, mobile or immobile, big or small.
- load balancing at the job level has limited ability in reducing the temporal fluctuations in the load, mainly due to the generation of multiple processes by some single jobs; balancing at a finer level (e.g., process or task) may be able to reduce temporal load fluctuations further, at the expense of increased communication overhead;

The rest of the paper is organized as follows. The design and implementation problems we dealt with are discussed in Section 2. In Section 3, we describe the design of the measurement experiments, including the artificial workloads we constructed. The results of the experiments are presented in Sections 4, 5, and 6, with Section 4 comparing the algorithms used for load balancing and assessing the importance of their adjustable parameters, Section 5 studying the effects of the workloads on load balancing performance, and Section 6 discussing the influence of load balancing on individual hosts and job types. The major results are summarized in Section 7.

2. DESIGN AND IMPLEMENTATION

We briefly review existing load balancing systems first. This prepares us for a presentation of the basic design and implementation of our prototype load balancer. Next, we describe the load balancing algorithms we have implemented and studied, and, finally, we provide some results of our overhead measurements.

2.1. Related Works

A number of load balancing implementations have been reported in the literature [Hwang82, Hagmann85, Bershad85, Ezzat86]. In almost all of them, a special syntax for command submission has been introduced to inform the system that the command is eligible for load balancing. In some cases, a specially constructed version of applications software is needed for remote execution. The operating system had to be changed in many cases in order to make remote execution possible.

The earliest implementation known to the authors was done at Purdue University in a UNIX environment [Hwang82]†. Special versions of compilers, assemblers, and text processing programs were constructed that called a system scheduling routine, *rxe*, to determine a "lightly loaded" destination host for execution. A modified form of the UNIX *load average*††, with considerations for the machine heterogeneity, was used as the load index.

Bershad implemented a load balancer for the Berkeley UNIX 4.2 BSD operating system [Bershad85]. Like the Purdue system, only a few programs with large CPU time demands ("CPU hogs") were considered, and the program and data files had to be explicitly moved to the execution host due to the lack of a distributed file system. System servers (daemons, in UNIX terminology) were used to exchange and maintain load information represented by load averages, and to create remote jobs upon user requests.

The Process Server implemented at Xerox PARC was targeted for a workstation environment [Hagmann86]. A collection of personal workstations are supported by Process Servers that may be permanently dedicated compute servers or workstations donated by their owners when they are not using them. A central agent (the Controller) is used to collect load information and perform job placements for the entire system. Each command has to be modified to make its remote execution possible.

Very recently, a load balancer for the NEST system of AT&T Bell Laboratories has been reported [Ezzat86]. The load balancer is implemented on a number of workstations connected by an Ethernet-like local network. The name of a special program, *rexec*, must be used as a prefix to any command string to be load balanced. *Rexec* obtains the hosts' loads, measured by their respective normalized response times, and transfers the command to the most lightly loaded host. Care was taken to use the software and to create temporary files on the execution host (rather than on the initial host) as much as possible, in order to improve performance.

2.2. System Basics

While the above load balancers have provided much knowledge about load balancing design and implementation, the requirements of our research were quite different from the ones of those systems. In designing our load balancer, we felt the following characteristics to be highly desirable:

1) *transparency*: no special syntax should be introduced, unless the user has some specific requirements; the placement of a job should be done automatically on the basis of the system's load conditions and the job's resource demands;

2) *no or little change to the system kernel**: the cost of installing and maintaining the load balancer should be minimized;

---

† Both the AT&T Bell Laboratories and the Berkeley versions of UNIX were present in the system.

†† Load average in UNIX is an estimate of the number of "active" processes in the system, averaged over a given period, e.g., 1 minute.

* In our implementation, we had to *add* a small amount of code to the system kernel to generate and maintain the load index used by the load balancing algorithms, and to provide enough precision for our measurements. No functional changes, however, was made to the kernel.

3) *no modifications to commands and applications*: the code of no existing command should have to be modified to adapt it to load balancing;

4) *general applicability*: we are interested in considering *all* types of jobs, at least in principle, rather than only a specific category, e.g., text processing commands; also, the design should not assume some specific system architecture; the same design should be suitable for time-sharing systems and compute servers, as well as personal workstations, provided that certain basic requirements are met, namely, a communication system and the availability of a distributed file system.

Like the designers of the other load balancers, we were also concerned with the overhead of load balancing; the remote execution of a job should not incur high overhead in terms of extra processing and real time delay. Since our implementation is experimental in nature, we are less concerned with issues such as remote process management and control, and user interface facilities, which we plan to add to the system for a production version.

There are two basic issues in the design of a load balancing system. The *policy* issue is concerned with the algorithm used to determine which jobs or processes should be executed remotely, and where. The *mechanism* issue is concerned with the physical facilities to be used for remote execution, i.e., with the way a job is transferred to a remote host and its results sent back. Before these two issues can be studied, however, we have to decide the level at which load balancing takes place. There are several choices. At the job, or command, level, the user interface can be changed so that some of the jobs submitted by the user may be redirected to some remote host for execution. Alternatively, load balancing can be done at the process level. In that case, the process management module of the system kernel must be modified to identify processes to be executed remotely. A third choice is to modify individual applications and incorporate remote execution facilities there [Johnston86]. However, considering our requirements discussed above, the second and third approaches are to be ruled out.

After the level of load balancing is determined, we still have to decide whether the jobs or processes are to be transferred during their execution (process migration), or only at start-up time (initial placement). Process migration has been suggested by a number of researchers as potentially more capable of improving system performance [Leland86, Cabrera86]. On the other hand, it is also likely to incur higher overhead, and is very difficult to implement in such systems as UNIX. In addition, since we decided to do load balancing at the job level, and multiple processes may be created by a single job, we would have to consider the interactions between the processes explicitly. These considerations led us to restrict ourselves to initial job placement in our experimental load balancer.

Our implementation is based on a modified C shell† implemented at Berkeley by Harry Rubin and Venkat Rangan for the Berkeley UNIX 4.3 BSD system running on VAX machines [Joy83, McKusick85]. This modified C shell intercepts user commands and executes certain types of commands remotely when the local host is heavily loaded, using the *rexec* daemon available in the system. The structure of our system is depicted in Figure 1‡.

---

† C shell is the name of the command interpreter in Berkeley UNIX operating system [Joy80].

‡ To distinguish our modified C shell from the standard one, we call it *C-shell*. The *R-shell*, to be

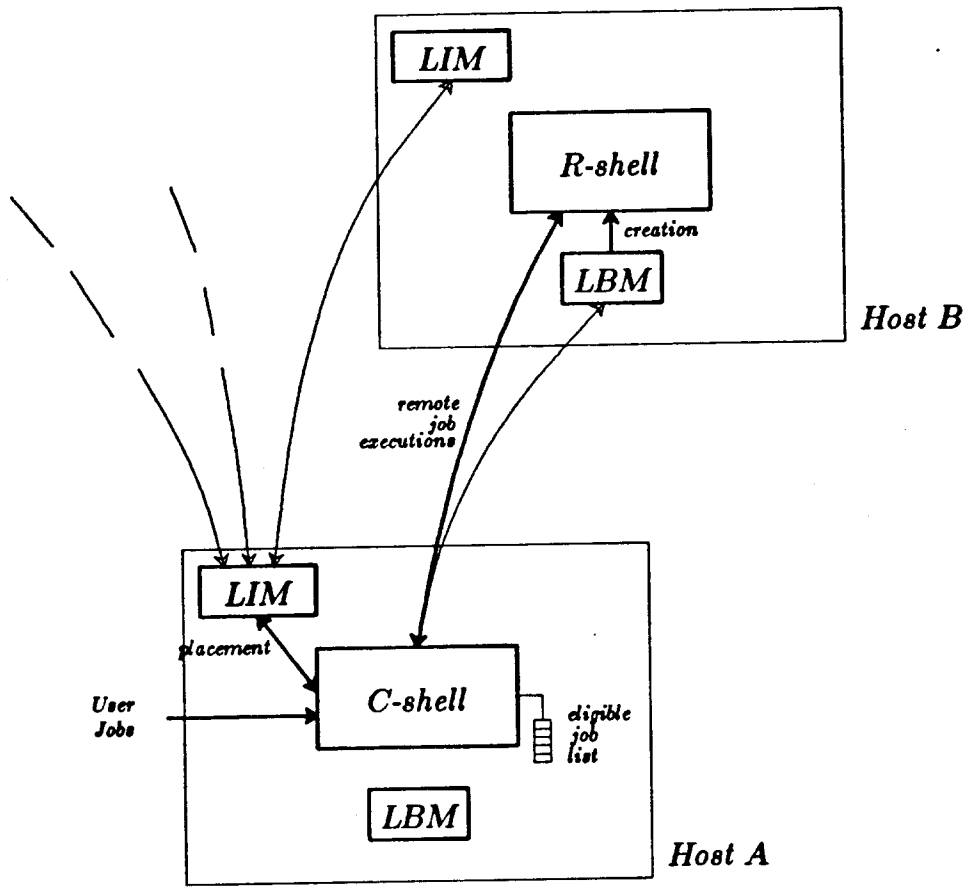


Figure 1. Structure of load balancing implementation.

At startup time, the C-shell reads in a configuration file that specifies a list of names of jobs that are eligible for remote execution*. When an eligible job is submitted by the user to the C-shell, the C-shell contacts a *Load Information Manager* (LIM), a software module that constantly monitors the loads of the hosts in the system and performs job placements. If the initial host is heavily loaded, while some other hosts are not, one of the remote hosts is selected as the destination for the job. In any case, the placement decision is returned to the C-shell. For remote execution, the C-shell contacts the *Load Balance Manager* (LBM)† on the destination host, which starts up an R-shell and establishes a stream connection between it and the home C-shell. The command line is transmitted over this connection to the R-shell after the user's identity has been authenticated and an appropriate user environment is set up there. Access control to files and other resources in the system is automatically enforced as the R-shell assumes the same user identity as that of the home

described below, shares the same software with C-shell, but is only to receive remote jobs and execute them.

* This list is part of the context of each user, just like command aliases, and may be dynamically modified by the user to suit his or her needs.

† Note that there is one LIM and one LBM on each host.

C-shell. Since starting an R-shell is an expensive operation as we will see below, we keep such a shell alive after the execution of the first job so that if a later command from the same user login session is placed on the same host, we do not have to go through the same process described above. The R-shells on remote hosts act as agents for the home C-shell, and are terminated when the home C-shell exits. This scheme has the potential problem of a proliferation of R-shells. However, the code segments of all C-shells and R-shells on each host are shared, so that, when an R-shell is not active, almost no resources are consumed.

Thus, our load balancing system design is at the job level, and stresses a clear separation of policy from mechanism. The collection and management of load information, and the job placement decision-making are performed by the LIMs, one on each host, and cooperating among themselves in ways dictated by the load balancing algorithm. Only the initial filtering of jobs by their names is performed in the C-shell to avoid querying the LIM too frequently, and to allow personalized selection of jobs. The load balancing mechanism, on the other hand, is provided by the LBM on each host, with the cooperation of the C-shell and R-shell. The separation between policy and mechanism makes it easy to experiment with different algorithms, as only the LIM needs to be changed. In fact, the LIM software can be constructed so that the load balancing algorithm may be changed dynamically as the system's size and load change.

We assume that the distributed system includes a distributed file system supported by one or more file servers. The program and data files of a job do not have to be fetched from the originating host, but from a file server, no matter where the job is executed. Thus, we assume that the cost of accessing the files is the same for all hosts. While this is basically true for the environment in which our load balancer is implemented (a number of diskless workstations supported by file servers), the location of the program and data is an important factor to consider in systems where files are scattered on a number of hosts (which are not dedicated file servers). We decided not to consider this problem in order to concentrate on the issues we are most concerned with now.

2.3. Algorithms

A large number of algorithms have been proposed in the literature (see [Wang85] for a taxonomy). The problem domain we are concerned with in this research (i.e., initial job placement in a loosely-coupled network environment for general-purpose computing, with distributed job submissions), and our desire to *implement* the algorithms make many of the proposed algorithms unsuitable. Also, we are not particularly interested in studying specific algorithms, but rather in comparing different approaches to load information exchange and job placement. Among dynamic algorithms, in which the current system's load conditions are considered, two large categories are commonly recognized. In *source-initiated* algorithms, the overloaded hosts actively seek hosts to transfer their jobs to. In contrast, the underutilized hosts actively look for jobs on overloaded hosts and execute them in *server-initiated* algorithms. Since server-initiated algorithms are most suitably supported by process migrations, as pointed out by Eager *et al.* [Eager86a], we will only consider source-initiated algorithms in this paper.

A load balancing algorithm consists of three component policies. The *information policy* decides what kind of information about a host's load (load index) is to be used to make job placement decisions, and the way such information is to be made available to the decision-makers. The *transfer policy* decides what jobs are eligible for remote executions, and under what local load conditions. Finally, the *placement policy* specifies the method with which a remote host is selected for the execution of a job. These three policies interact in various ways: the placement policy utilizes the load information supplied by the information policy, and acts only on the jobs determined to be eligible by the transfer policy.

We implemented and studied five load balancing algorithms that differ in their information policies and in the corresponding placement policies. The load index used by all the algorithms is the same, and will be described in the next section. All the algorithms use the same transfer policy, which is based on the job name, as described in the previous section, and on a local load threshold. More specifically, if the name of the job (i.e., the command) is on the user's list, the job is considered eligible for remote execution, and one of the LIMs is contacted for placement. The LIM checks the load on the job's originating host, and will consider the job for remote execution only if the load is above a specified threshold T_r. Notice that, even in this case, the job is not necessarily executed remotely; if the LIM fails to find a suitably lightly loaded host, the job will be executed locally. The algorithms we studied are described below.

DISTED

Periodically, the LIM on each host extracts load information from the local kernel to compute the load index. If the new value of the load index is significantly different from the previous one, it is broadcast for every other LIM to update its record of this host's load†. When a LIM receives a placement request from one of the local C-shells, it first decides whether the local load is above the threshold. If this is the case, the LIM searches through its list of host load records, selects as the destination the host appearing to have the lightest load, and informs the C-shell of the decision. The job will be executed locally if all the hosts are heavily loaded.

GLOBAL

The information exchange method used by the DISTED algorithm above is straightforward, but generates a large number of broadcast messages. The GLOBAL algorithm attempts to cut down the number of messages by employing a master LIM that receives load information from all the other (slave) LIMs periodically, and periodically broadcast the load vector containing the loads of all the hosts. The placement policy of GLOBAL is the same as that of DISTED.

CENTRAL

---

† Some refinements to this basic scheme have been implemented. For example, the local load is broadcast once in a while even if the local host's load has not changed much, so that other hosts will not assume that this host is unavailable. Also, the local load is not broadcast if it stays above an upper threshold T_u.

This algorithm goes one step further than GLOBAL in centralizing the placement decision-making. Not only is the load information collected by the master LIM, but also all the placement requests are directed to it. Consequently, load information flows only from the slaves to the master, and the role of the slave LIMs is reduced to that of periodically reporting local load information to the master. If communication is fast and the system's scale is not too large, this algorithm can support the system with a low volume of information exchanges. The placement policy of CENTRAL coincides with that of DISTED except that the master LIM, instead of the local LIMs, performs all the placements. This algorithm is used in the Process Server [Hagmann86].

LOWEST

The above algorithms rely on periodic load information exchanges to provide the LIMs with reasonably up-to-date load information. The LOWEST algorithm acquires such information on demand in a distributed fashion. When a placement request arrives at a local LIM, this LIM polls a number of hosts up to a limit L_p specified by the information policy, and selects the host with the lightest load. If the system is large, it becomes impractical to poll every host, and the placement may be suboptimal. On the other hand, unlike the above three algorithms, the overhead incurred by the information policy of LOWEST is independent of the system's size. This algorithm and the next one were proposed by Eager *et al.* [Eager86b]

RANDOM

This algorithm does not need any load information other than that for the local host. If a job is determined to be eligible, a remote host is picked at random, and the job transferred there. Because of implementation difficulties, no retransfer of jobs is allowed in our version of this algorithm.

For comparison, we also ran measurement experiments with load balancing disabled. We call this the NoLB case.

It is recognized that there exist other algorithms that can potentially produce good performance. The above five algorithms were chosen because they are implementable, and they represent different approaches to the load information exchange (periodic versus on-demand), to the job placement (system-wide selection, subset, random), and they are of varying levels of complexity, with RANDOM being the simplest.

2.4. Overhead Assessment

We measured the additional CPU processing and job delays due to load balancing, that is, to the exchanges of load information, the job placements, and the remote executions. Table 1 shows some of the results for Sun-2 workstations with 2 MB of memory and a 3Com Ethernet board. Note that all times in the table are real time delays and are averages of a few hundred to a few thousand repetitions. The measurements were taken on empty hosts. When the system is loaded, the delays become longer and their variance increases. For locally executed jobs, the average overhead is very low, typically 5-10 milliseconds, and is mainly due to searching the job list in the C-shell, and, in case the job

Table 1. Load balancing overhead measurements

extract load info. from kernel and send out a message (500 bytes)	14.5 ms
receive a load message and store into load vector (500 bytes)	5.7 ms
placement request by C-shell to LIM (round-trip) to local LIM to remote LIM (for CENTRAL)	 23.8 ms 52.9 ms
remote job execution overhead (incl. placement by local LIM, assuming R-shell already set up)	325 ms
start an R-shell (setup)	5 sec

name is on the list, to querying the LIM. The delay due to a LIM query plus the overhead of remote execution is highly variable, depending on the loads of source and destination hosts. On the average, it is a few hundred milliseconds. This assumes that an R-shell has already been set up on the destination host. Otherwise, several seconds of additional delay may be incurred. Overall, the overhead of load balancing seem to be quite low. With an exchange period of 3 seconds, load information updates cost from one to a few percent of CPU time on Sun-2 workstations. The delay due to remote execution is hardly perceivable by an interactive user, and is very small compared to the average job response time, which is in the range of a few tens of seconds.

3. EXPERIMENT DESIGN

Before we discuss our experiments, we need to describe the performance index used to assess and compare the performance of different systems or algorithms. Since we are interested in an interactive computing environment, the mean response time of all the jobs executed during a measurement session seems to be an appropriate performance index. However, the response times of jobs executed remotely in the background turned out to be difficult to obtain in our implementation. Instead, we made use of the system accounting facility to obtain the response times of all the *processes* executed during a measurement session, and used the mean process response time as our performance index. For the execution of most of the jobs, only one process is created, so the two indices are the same, except for the command line processing in the C-shell, which is not accounted for in the process response time. For a few commands (namely, *cc*, *lint*, and *ditroff*), however, several processes are created, and their response times are all considered in computing the mean. The overhead of load balancing is accounted for by measuring it during the experiment run and adding it to the process response times. Another important concern in system performance usually is the predictability of the process response times. In many cases, making the response time more predictable is at least as important as reducing the mean. We use the standard deviation of the process response times as a measure of predictability.

We identify four major factors that affect the performance of a load balancing system. First, load indices that capture the current load conditions and are, preferably, capable of predicting host load in the near future are of crucial importance. A poor load index may cause job transfers that do not contribute to balancing the load of the system, and might even make things worse. Secondly, the algorithm used for load balancing determines the cost of distributing load information, and the quality of job transfers. Thirdly, the performance improvements due to load balancing are dependent also on the workload the system is subjected to. The workload will be characterized along two dimensions, which will be considered as independent factors: that of its *intensity*, i.e., its magnitude, and that of its *mobility*, i.e., the fraction of the workload (as defined in Section 5.2) that can to be executed remotely. Lastly, the underlying implementation of the load balancer certainly impacts load balancing performance, but since the implementation is fixed in our case, our measurement experiments only explore the remaining three dimensions. More specifically, we vary one factor at a time and study its influence. A number of levels or values are assigned to each of the factors, as listed below.

• Load index:	instantaneous CPU queue length time-averaged CPU queue length linear combination of averaged CPU, paging/swapping, and I/O queue lengths averages of different intervals
• Algorithm:	NoLB, DISTED, GLOBAL, CENTRAL, LOWEST, RANDOM each algorithm has a number of adjustable parameters
• Workload intensity:	each host uses one of three types of artificial workloads: light (L), moderate (M), and heavy (H). A system workload is a combination of the host workloads. So, for a system of six hosts, we studied the following combinations: 2H, 2M, 2L (*canonical workload*); 5H; 6M; 6L
• Workload mobility:	several values of the *immobility factor* (see Section 5.2)

The problems concerning load indices are very important. In [Ferrari86], a linear combination of resource queue lengths, with the corresponding job resource consumptions as coefficients, is proposed as a load index based on mean value analysis, and an experimental evaluation of that index is presented. That work is carried forward in [Ferrari87] where a measurement-based evaluation of a wide range of load indices using the implementation described in this paper is presented. Instead of repeating the results of that research, we shall just use the load index that we found to be among the best, that is, the sum of the process queue lengths of the CPU, the paging system, and the I/O system, averaged over a 4 second period. Note that this equals the total number of processes ready to run and executing, being paged/swapped, and doing file I/O, respectively. We have described the algorithms we study in Section 2, and Section 4 will present the results using the canonical workload.

The construction of workloads accounted for most of our efforts in the design of the experiments. On the one hand, since a high degree of repeatability of the experiments was felt to be absolutely necessary, we used artificial workloads. On the other hand, we want these workloads to represent real workloads reasonably well, so that we can have confidence in the realism of the results. We traced a production VAX-11/780 running under the Berkeley UNIX 4.3BSD system [Joy83, McKusick85] for an extended period of several months and analyzed the types and frequencies of the commands executed by the system. On the basis of such an analysis, we selected a number of frequently executed commands, as listed in Table 2, and used them to construct scripts, i.e, streams of commands.

Table 2. Commands used in scripts and their eligibilities for remote execution

command	elig.	function	command	elig.	function
cat	N	*view a file*	*ls*	N	*directory listing*
cc	Y	*C compiler*	*man*	Y	*manual page viewing*
cp	N	*file copying*	*mv*	N	*move a file*
date	N	*current time*	*nroff*	Y	*text formater*
df	N	*file system usage*	*ps*	N	*process checking*
ditroff	Y	*text formater*	*pwd*	N	*current directory*
du	N	*disk usage*	*rm*	N	*delete a file*
egrep	Y	*text pattern search*	*sort*	N	*file sorting*
eqn	Y	*equation formater*	*spell*	Y	*spelling checker*
fgrep	Y	*text pattern search*	*tbl*	Y	*table formater*
finger	N	*user information*	*troff*	Y	*text formater*
grep	Y	*text pattern search*	*uptime*	N	*system uptime*
grn	Y	*graph printing*	*users*	N	*list of current users*
lint	Y	*C program checker*	*wc*	N	*word count in a file*
lpq	N	*printer queue check*	*who*	N	*user information*

To obtain various levels, or intensities, of load, such as those characterizing multi-user systems, we ran a variable number of the jobs in the background. Also, we simulated user think times by the "sleep" command. The scripts are classified into three levels: light, moderate, and heavy, with a number of distinct scripts constructed for each level so that hosts subjected to the same level of workload can use different scripts. The ranges of CPU utilizations and mean load index values of the three levels of scripts are shown in Table 3. Each script runs for about 30 minutes on a Sun-2 workstation. Job and system performance statistics, such as resource demands, response times, resource utilizations, and resource queue lengths, were measured throughout each run.

Table 3. Characterization of the workload levels

type	CPU utilization	average load index
light	30-45%	0.3-0.7
moderate	60-70%	1.0-1.8
heavy	70-85%	1.8-3.0

As in any measurement experiment, we must consider the variability of the experimental environment, and, therefore, that of the measurement results. In dynamic load balancing, the placement of each job may vary from one run of the experiment to the next, because of the unavoidable variations in the timings of the events. (This problem was further complicated in our experiments by the fact that we had to share the file server and the network with other parts of the research community. We tried to minimize this impact by running the experiments during the night.) Thus, we repeated the same experiment a number of times (typically 6), and computed the mean and the 90% confidence interval (CI) of the performance indices over these replications.

4. COMPARISONS OF ALGORITHMS

We first compare the performances of the algorithms, then study the effects of the adjustable parameters for the algorithms.

4.1. Basic Comparisons

To compare the performances of the five algorithms described in Section 2, we applied each of them to a system of six Sun-2 workstations running the canonical workload described in Section 3. With this workload, two of the six workstations were subjected to heavy job scripts, two to moderate scripts, and the remaining two to light scripts. For each of the algorithms, we varied the adjustable parameters (considered as secondary factors), such as the local load threshold T_r, the load exchange period P for the periodic policies, and the probe limit L_p for the non-periodic policy (LOWEST), in order to achieve the best performance under that algorithm. For algorithm, Table 4 shows the mean response time and its 90%† confidence interval, the percentage improvement in response time relative to the NoLB case, the standard deviation of the response times and its percentage improvement, and the values of the adjustable parameters used in the run.

The first observation one can make about the results in Table 4 is that load balancing can indeed improve system performance substantially. The canonical workload was constructed to reflect a loading situation commonly observed in production environments: some workstations are loaded, while others are not. By transferring jobs from heavily loaded hosts to lightly loaded ones, the mean job response time can be improved. Comparing the

† All confidence intervals in the tables and figures of this paper have been computed with a 90% confidence level.

Table 4. Performance of the algorithms (all times are in seconds)

replication count: 6
total number of jobs per run: 501
total number of eligible jobs per run: 254 (50.7%)
total number of processes per run: 766 (1.53 processes/job)
average process execution time: 7.45
approximate average CPU utilization for NoLB case: 60%

Algorithm	Resp. Time	Improv.	Std. Dev.	Improv.	Parameters
NoLB	53.3 ±0.83	0.0%	90.1	0.0%	---
DISTED	36.4 ±0.09	31.7%	50.6	43.8%	$P=15$, $T_r=0.8$
GLOBAL	32.6 ±0.67	38.9%	43.6	51.7%	$P=5$, $T_r=0.8$
CENTRAL	33.7 ±0.54	36.8%	48.5	46.8%	$P=10$, $T_r=0.8$
LOWEST	31.8 ±0.37	40.3%	42.8	52.5%	$P_r=4$, $T_r=0.8$
RANDOM	39.9 ±1.21	25.2%	62.0	31.2%	$T_r=1.0$

improvements in mean response time and those in the standard deviation of the response times, we notice that the latter is reduced more substantially. This means that the job response times are more predictable with load balancing than without.

The performances of the algorithms, except that of RANDOM, are quite close to each other. In Section 2, we described the algorithms and pointed out that, in the periodic algorithms, the information is ready when a job is to be placed, and the "best" host in the system is selected. However, the periodic updates incur higher computation and communication overhead than the polling method used by LOWEST, and the load information used in placements tends to be less current than that in LOWEST. Comparing DISTED and GLOBAL, we see the adverse effect of the excess use of broadcast messages, as the two algorithms are the same except that, in GLOBAL, a master is used to collect and distribute load information. As a result, only the master has to handle N messages per period P, where N is the number of hosts, while all the other hosts need only to send one message and receive one during each period. For more discussion of the overheads of distributed and centralized load information exchanges, the reader is referred to [Zhou86].

A complete evaluation of the qualities of the algorithms cannot be done using a system of only six hosts. However, since our measurements agree well with our simulation results [Zhou86], we feel confident about those results. We simulated systems with 7, 14, 21, 28, 35, 42, and 49 hosts, and found that the scalability of the GLOBAL and CENTRAL algorithms is very good. Their performances are comparable to that of LOWEST throughout the range of system sizes. On the other hand, the performance of DISTED becomes worse as the system grows larger, since, in DISTED, the information exchange overhead per host is linear in the number of hosts.

4.2. Adjustable Parameters

The performance of load balancing is dependent on the parameters used in the algorithms. While it is impractical to explore all the possible variations, or even to present in this paper all the experiments we performed, we show the effects of the three most important parameters, namely, the load exchange period P, the local load threshold T_r, and the probe limit L_p, on three of the algorithms, GLOBAL, RANDOM, and LOWEST, respectively. For all cases, the canonical workload is applied to the six-host system, and the brackets around the data points show the 90% confidence intervals.

The mean process response times of GLOBAL using various values of P is shown in Figure 2.

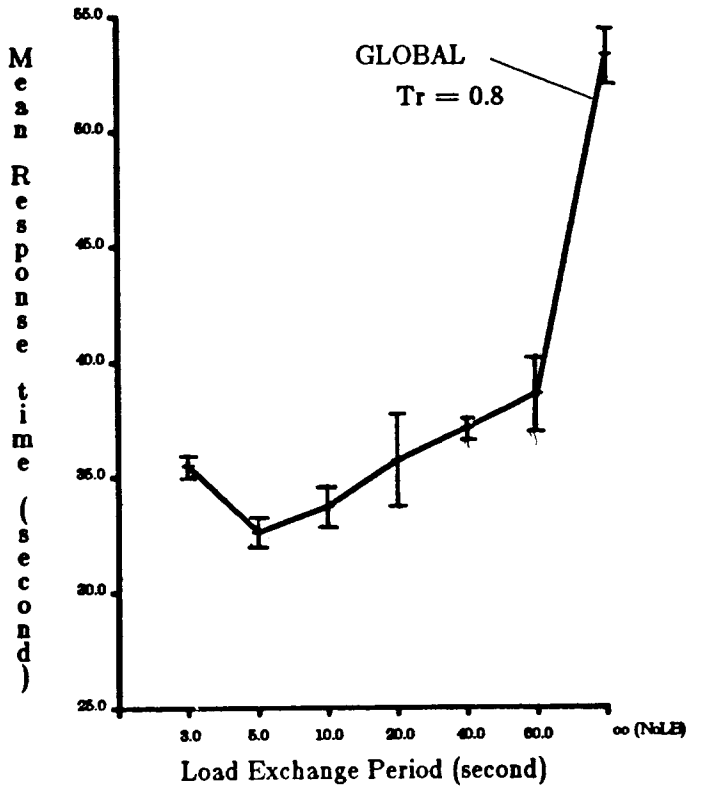


Figure 2. Mean process response time under various load exchange periods P (Canonical workload, GLOBAL, T_r=0.8).

When the exchange rate is too high, the overhead outweighs the benefit of up-to-date information. On the other hand, if the rate is too low, the information may get too stale, and performance suffers. The optimal exchange rate is also dependent on the workload. Specifically, the rate should be higher if the job arrival rate is high and the average resource demands of the jobs are low. This is the case in our simulation studies for multi-user time-sharing systems. It is remarkable, however, that substantial performance gains are still achieved with an exchange period as long as 60 seconds. At that point, it becomes quite possible that multiple jobs are transferred during the period to a host that used to be lightly loaded, and actually make it overloaded. This form of system instability is called *host overloading* [Zhou86]. The message here seems to be that a load balancing system can

tolerate a certain level of host overloading without suffering substantial performance degradation.

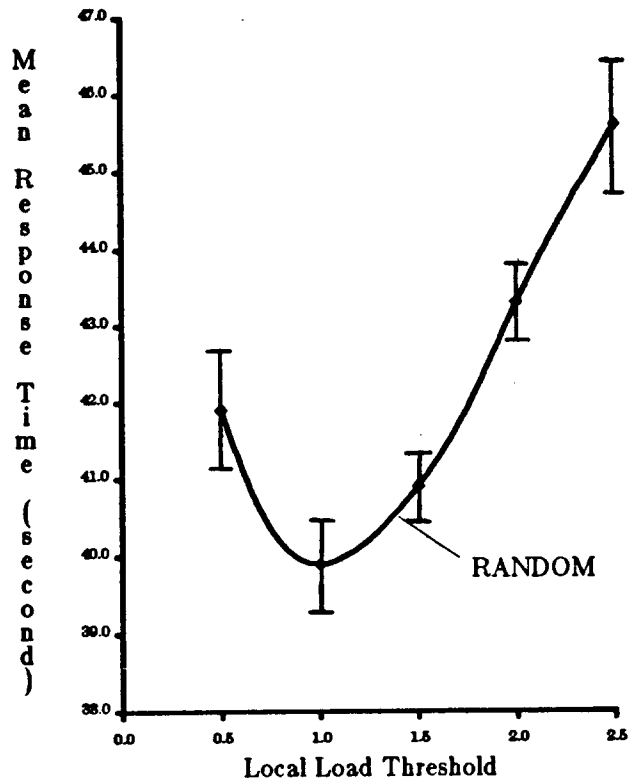


Figure 3. Mean process response time under various local load thresholds T_r (Canonical workload, RANDOM).

Similarly, there are conflicting requirements for the local load threshold T_r. On the one hand, a sufficient number of jobs have to be transferred between the hosts in order to balance their loads. On the other hand, however, an excessive amount of job transfers will increase system overhead, and may even cause severe host overloading. This tradeoff is illustrated by Figure 3, which shows the relationship between the mean response time and the local load threshold for the RANDOM algorithm, which uses T_r as its sole parameter. Again, the optimal threshold is dependent on the load level of the system. If all the hosts are subjected to heavy workloads, T_r should be set relatively high to avoid unproductive job transfers.

We also studied the performance of LOWEST with various values of the host probe limit L_p. The results are displayed in Figure 4, which shows a minimum like those in Figures 2 and 3.

From Figures 2, 3, and 4, it is clear that the parameter values of the algorithms should be dynamically adjusted as the system load conditions change over time, in order to keep obtaining most of the performance gains of load balancing. Load balancing algorithms that dynamically adjust their parameters may be called *adaptive algorithms*. Such adjustments require system wide load information. It seems likely that the algorithms that

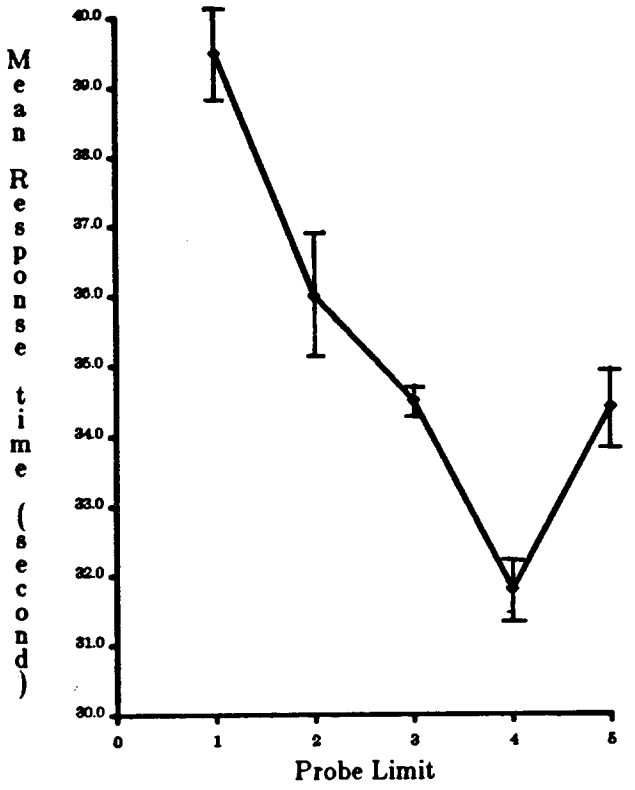


Figure 4. Mean process response time under various probe limits L_p (Canonical workload, LOWEST, $T_r = 0.8$).

actively exchange such information among the hosts will be better capable of making parameter adjustments. Furthermore, the introduction of a central agent should facilitate this task, as parameter adjustment decisions can be made by the central agent quickly and effectively, and propagated to the other hosts. In this regard, GLOBAL and CENTRAL seem to be more suitable for *adaptive load balancing* than LOWEST and DISTED.

5. PERFORMANCE UNDER DIFFERENT WORKLOADS

The previous section compared the performance of the algorithms using the canonical workload. In this section, we study load balancing performance under different workloads. We first study workloads of different intensities, then study those with different levels of mobility. The GLOBAL algorithm was chosen for this part of the study.

5.1. Different Intensities

Tables 5, 6, and 7 show the values of the performance indices and their improvements relative to the NoLB case when all hosts in the system are subjected to heavy, moderate, and light load, respectively. Although the load level is the same for all the hosts, separately constructed scripts are used so that no synchronization effect will occur.

Load balancing can provide performance gains due to two factors: long-term system load imbalances and short-term load imbalances. For the canonical workload, significant differences in host loads over the entire run (long-term imbalances) exists, so the

Table 5. Five hosts with heavy loads ($P = 10.0$ sec, $T_r = 1.0$)

Algorithm	Response Time	Improvement	Std. Dev.	Improvement
NoLB	87.0 ±2.03	0.0%	121.4	0.0%
GLOBAL	59.4 ±0.15	31.7%	75.9	37.5%

Table 6. Six hosts with moderate loads ($P = 10.0$ sec, $T_r = 0.8$)

Algorithm	Response Time	Improvement	Std. Dev.	Improvement
NoLB	49.5 ±0.27	0.0%	72.4	0.0%
GLOBAL	39.4 ±0.44	20.5%	57.5	20.6%

Table 7. Six hosts with light loads ($P = 10.0$ sec, $T_r = 0.6$)

Algorithm	Response Time	Improvement	Std. Dev.	Improvement
NoLB	28.7 ±0.65	0.0%	38.7	0.0%
GLOBAL	25.2 ±0.52	12.2%	31.4	18.9%

performance gains can be easily explained. For the workloads used in this section, however, the hosts are similarly loaded, yet sizable reductions in response times are observed for the heavy and moderate workload cases. These gains can only be attributed to the short-term host load imbalances. At any particular point in time, some hosts are likely to be significantly less loaded than others, hence transferring jobs to them will reduce the overall mean job response time. The distinction between senders and receivers is not clear here; a host may be overloaded at one time and transfers jobs out, and underloaded later and receives jobs from other hosts.

A comparison between the response time reductions in the three cases show that the higher the system load, the greater performance improvement may be expected. This is intuitive, but also highly desirable. Also, it should be noted that the reductions in the standard deviation of the process response times when the hosts are evenly loaded are not as large as in the long-term unbalanced case in Section 4.

The reader may have noticed that, while six workstations were used for the moderate and light workloads, only five are used for the heavy workload. This is because in the latter case the file server was heavily congested by file requests. In our system, all the workstations get their files, and all but two of the workstations do remote paging and swapping, from a single file server, which is also shared by other workstations, and is simply another Sun-2 workstation configured with disks. When the six workstations are active, the load on the file server becomes higher than that on the workstations, even for the moderate workload case. With a heavy workload, the file server can be overwhelmed with

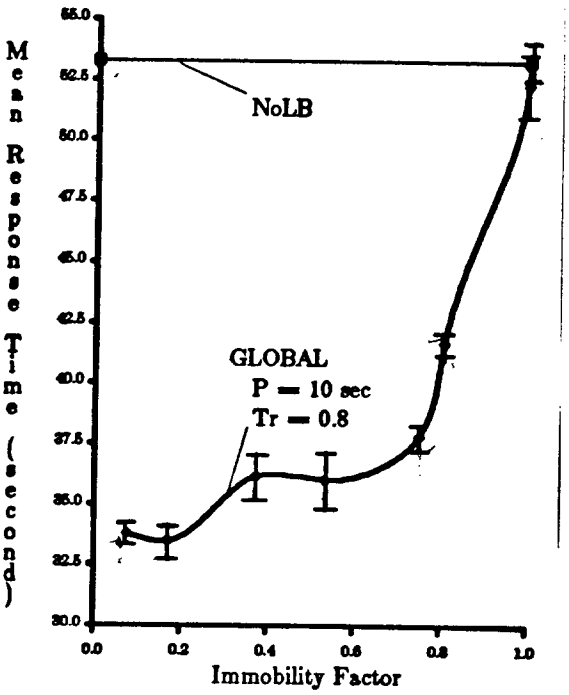
file access and paging requests, with its average load index going up to 6 and over. Our experience agrees well with the results of a performance study of diskless workstations by Lazowska *et al.*, in which the authors concluded that the file server's CPU tends to be the first resource in the system to saturate [Lazowska84]. With the file server's CPU being the focus of contention, the system is no longer correctly configured, and the potential benefits of load balancing are overshadowed by the negative impact of a major I/O bottleneck. We conjecture, therefore, that greater performance gains are possible if more powerful and/or multiple file servers are provided. A load index value of 3 is considered to represent a heavy load in our workstation environment, but may be considered quite normal in compute servers or time-sharing systems. With the possibly higher loads in those types of environment, the utility of load balancing should be greater.

5.2. Different Mobilities

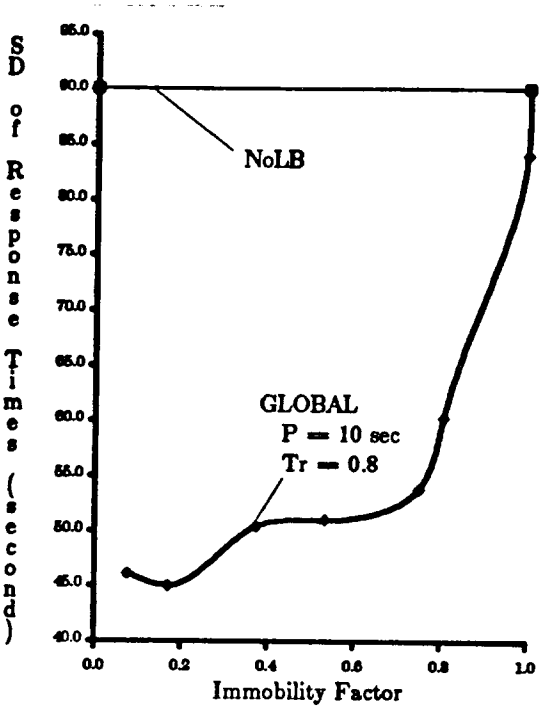
In any computer system, there exist jobs that perform local services and/or require local resources, and hence cannot or should not be transferred by the load balancer. Examples include system servers, login sessions, mail and message handling programs, and highly interactive jobs such as command interpreters and text editors. These jobs are bound to have an adverse impact on load balancing performance, as the choice of jobs to be transferred is now limited. In [Zhou86], we studied this problem and defined as *immobility factor f* the percentage of jobs that cannot be transferred. In this paper, however, we find it more convenient and accurate to define the immobility factor as the percentage of CPU time consumed by the immobile jobs over all jobs. The impact of immobile jobs on the mean response time is depicted in Figure 5. The different values of the immobility factor shown in the graph were obtained by changing the list of eligible jobs in the configuration file, as we can easily measure the total amount of CPU times consumed by each type of jobs, and compute their respective percentages of the total. Note that the canonical workload used in all the previous sections corresponds to an immobility factor of 0.17. As we have observed in [Zhou86] using trace-driven simulation, the curves are distinctively concave. Even when the immobility factor is as high as 0.8 (i.e., 80% of the workload is immobile), most of the performance gains of load balancing are still retained. This seems to suggest an characteristic of load balancing: only a small percentage of the jobs need to be transferred among the hosts to achieve effective load balancing. For a wide range of immobility factor values and other adjustable parameters, we have observed that only less than half of the eligible jobs are actually transferred.

6. EFFECTS ON INDIVIDUAL HOSTS AND JOB TYPES

In the above two sections, we have studied the influences of the two major factors, namely, the algorithms and the workloads, on load balancing performance. We go into more detailed studies in this section by examining the impact of load balancing on the loading and performance of the individual hosts and on the response times of each type of jobs.



5.1. Response time vs. immobility factor.



5.2. Standard deviation of response times vs. immobility factor.

Figure 5. The influence of immobile jobs (Canonical workload, GLOBAL).

6.1. Effects on Individual Hosts

Although it is now clear that load balancing can improve system-wide performance, its impact on the loading of individual hosts is equally important, especially in a workstation environment. Figure 6 shows the average load index value of each host throughout a run, and with different values of the immobility factor f. We see a significant reduction in the loads of all the hosts except those that were originally very lightly loaded. This is a confirmation of the reduction in the average response times we observed, and is in agreement with the Little's result. We also notice a strong equalization of the hosts' loads: as the immobility factor goes from 1.0 down to 0.17, the hosts' loads are compressed into a narrow range. Thus the term "load balancing" is truly appropriate in our case, even though none of the algorithms we studied takes it as its explicit objective.

The fact that the loads of the hosts tend to become balanced on the average does not necessarily mean that they are balanced during shorter intervals, which would be highly desirable though. Indeed, this is shown not to be the case by Figure 7, where the 20 second average load index (instead of the 4 second average used earlier in this paper) is plotted as a function of the time during a run. Several comparisons may be made using the plots. Comparing the loads of the hosts without load balancing, we see significant differences in loads. These differences are substantially reduced by load balancing. However, there still exist load fluctuations in each host. Our load balancer operates at the job level, and several processes may be created by a single job. As long as those processes are treated as an

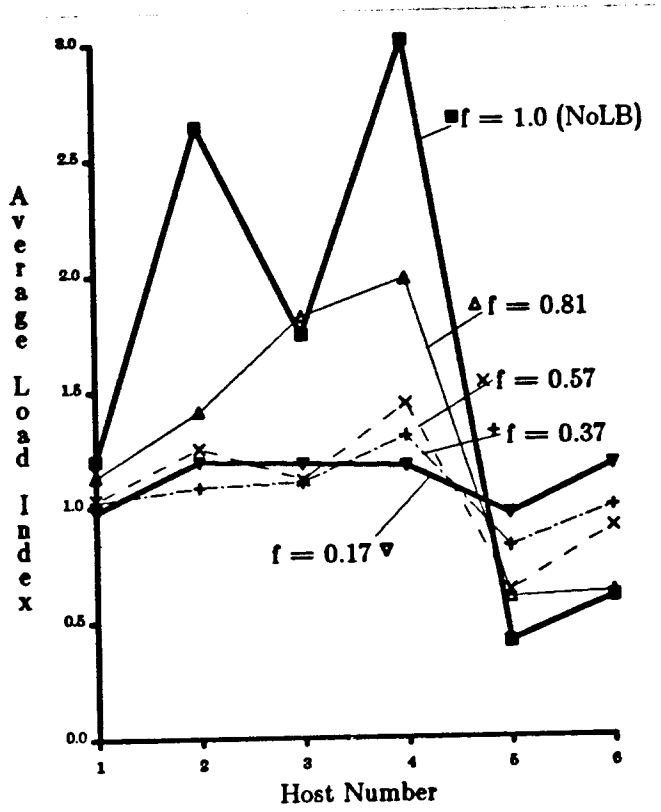


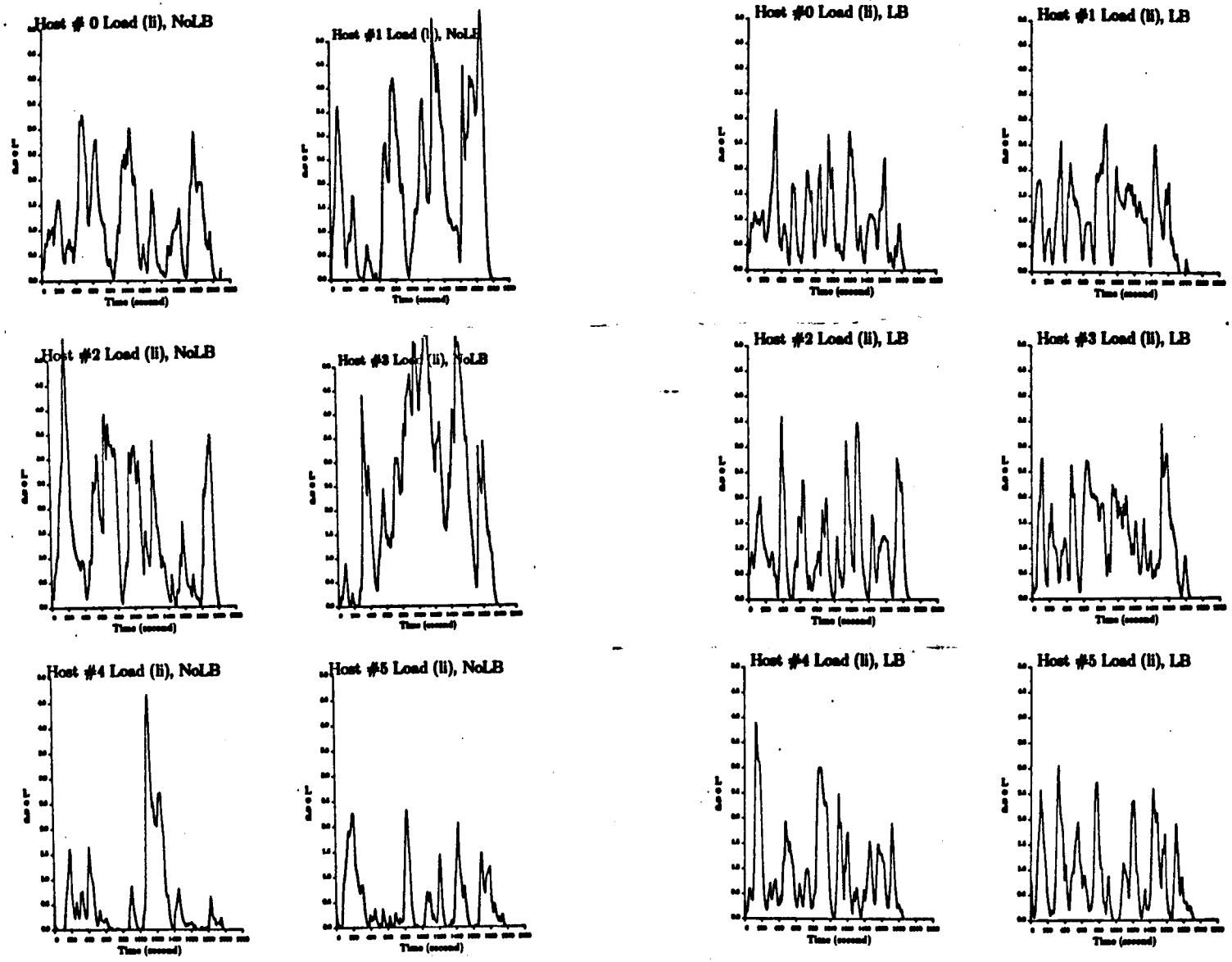
Figure 6. Loads on hosts, with various immobility factors
(Canonical workload, GLOBAL, $P = 10.0$ sec, $T_r = 0.8$).

inseparable group, temporal fluctuations in load seem unavoidable. Since smoothing the hosts' load over time is highly desirable, we conclude that load balancing at the job level using initial placement only has the drawback of not being able to eliminate temporal fluctuations. On the other hand, it is questionable whether the performance gains due to further reductions in temporal load fluctuations provided by load balancing at a finer granularity will more than offset the additional communication and computation overhead. More research is called for here.

6.2. Effects on Each Type of Jobs

The conjecture could be made that, while the mobile jobs will generally benefit from load balancing, the immobile jobs will not benefit much, or not at all. Our measurements contradict this conjecture. Table 8 lists the mean response times of each type of jobs executed during the runs with and without load balancing. All times are in seconds, and the percentage improvements are provided following the response times for the load balancing case.

The average response times of *all* types of jobs are reduced, and, with only a few exceptions (*cp*, *date* and *finger*), the reductions are uniform across the board. There is no clear difference in improvements between different classes of jobs, big or small, mobile or immobile. While the response time of a job to be transferred will improve because it will be executed on a more lightly loaded host, those of the jobs already running on the initial host



7.1 Hosts' loads without LB

7.2 Hosts' loads with LB

Figure 7. Time diagrams of the hosts' loads with (7.2) and without (7.1) load balancing (GLOBAL, Canonical workload, $P = 10.0$, $T_r = 0.8$).

Table 8. Average response time of each command type with and without LB (Canonical workload, GLOBAL with $T_r = 0.8$, $P = 5$ sec).

cmd	elig.	count	NoLB	LB	cmd	elig.	count	NoLB	LB
cat	N	33	5.19	3.53 (32.0%)	*ls*	N	53	52.7	30.3 (42.5%)
cc	Y	54	89.1	53.8 (39.6%)	*man*	Y	8	20.2	6.78 (66.4%)
cp	N	3	2.34	2.30 (1.7%)	*mv*	N	2	3.61	1.72 (52.4%)
date	N	22	1.81	1.46 (19.3%)	*nroff*	Y	17	181	102 (43.7%)
df	N	9	6.22	3.61 (42.0%)	*ps*	N	23	22.5	14.1 (37.3%)
ditroff	Y	7	324	194 (40.1%)	*pwd*	N	18	4.26	3.02 (29.1%)
du	N	6	82.6	55.1 (33.3%)	*rm*	N	0	-	-
egrep	Y	7	22.1	6.07 (72.5%)	*sort*	N	30	105	66.8 (36.4%)
eqn	Y	5	103	64.2 (37.8%)	*spell*	Y	45	117	73.6 (37.1%)
fgrep	Y	10	19.2	11.9 (38.0%)	*tbl*	Y	2	109	55.9 (48.7%)
finger	N	25	92.6	80.4 (13.2%)	*troff*	Y	12	110	65.8 (40.2%)
grep	Y	3	12.1	6.56 (45.8%)	*uptime*	N	34	7.85	4.18 (46.8%)
grn	Y	7	277	158 (43.0%)	*users*	N	4	7.08	3.20 (54.8%)
lint	Y	24	78.6	42.0 (46.6%)	*wc*	N	15	12.3	5.13 (58.3%)
lpq	N	12	29.8	15.2 (49.0%)	*who*	N	11	4.78	2.29 (52.1%)

will also improve because they will not have to compete with the new-comer.

7. CONCLUSIONS

In this paper, we described the design and implementation of a load balancer for a loosely-coupled distributed system, and presented some of the results of a large number of measurement experiments performed on the system. On the basis of our findings, we believe that transparent, flexible load balancing at the job level can be achieved at low cost, and without modifying either the system kernel or any of the existing application programs. Our design emphasizes a clear separation between the mechanism and the policies for load balancing, thereby allowing the particular load balancing algorithm, along with its adjustable parameters (e.g., the load exchange period, the local load threshold, and the probe limit) to be dynamically changed in response to changing system load conditions.

Measurements show that load balancing can indeed significantly reduce the mean process response time, and that the corresponding reduction in the standard deviation of process response times is usually even greater. Furthermore, the improvements are largely uniform over all classes of jobs, big or small, mobile or immobile, and most of the

improvements can still be retained when up to 80% of the workload cannot be transferred between the hosts. While we observed that load balancing has strong equalization effects on the individual hosts' loads over the entire measurement runs, there still exist temporal fluctuations in host loads. We attribute this drawback to the fact that several processes may be created by a single job, and suggest that load balancing at a finer granularity be studied to see whether this conjecture is correct, and whether such fluctuations can be advantageously reduced.

Five source-initiated load balancing algorithms were studied that used different methods to distribute load information and perform job placement. We find that algorithms using periodic load exchanges and those acquiring such information on demand provide comparable performances. For the former class of algorithms, the use of a central agent to collect and distribute load information reduces the computation and communication overhead, and hence provides better performance. The centralized algorithms are also better suited for adaptive load balancing, in which the algorithm and/or its parameters may be changed dynamically. On the other hand, distributed algorithms such as LOWEST generally impose lower overhead, scale better, and are more reliable. We also find that the performance of load balancing is, to various degrees, sensitive to the algorithms' parameter values.

As well as load balancing algorithms and their parameters, workloads also have a strong impact on performance. Generally speaking, the higher the load, the greater the imbalances in the hosts' loads (both long-and short-term), the greater the performance improvements that may be expected. Short-term imbalances can be as profitably exploited as long-term imbalances, as demonstrated by the performance gains when all the hosts are subjected to similar levels of loads.

ACKNOWLEDGEMENTS

The authors are grateful to Harry Rubin and Venkat Rangan for making their load balancing C shell available to us. Sincere appreciation is due to Keith Sklower and Joan Van Horn for tirelessly explaining the system to us and repairing software and hardware problems in the system. We are also indebted to the research community at Berkeley for tolerating the additional loads we managed to generate on the file servers and the network unproductively. Last, *but not least*, we thank our six workstations, *Bluefairy*, *Dopey*, *Jiminy*, *Joshua*, *Lampwick*, and *Thumblina*, for all the sleepless nights they spent compiling programs that will not be run and formatting papers that will not be published. We will soon forget all the miseries they caused us by crashing at the most inconvenient moments.

REFERENCES

[Barak84]
A. Barak and A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer," Technical Report, Department of Computer Science, The Hebrew University of Jerusalem, 1984.

[Bershad85]
B. Bershad, "Load Balancing with Maitre d'," Tech Report, UCB/CSD 85/276,

Computer Science Division, University of California, Berkeley, December 1985.

[Bokhari79]
S. H. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment," IEEE Trans. Soft. Eng., Vol. SE-5, No.4, pp. 341-349, July 1979.

[Bryant81]
R. Bryant and R. Finkel, "A Stable Distributed Scheduling Algorithm," Proc. 2nd International Conf. on Distributed Computing Systems, pp. 314-323, 1981.

[Cabrera86]
L. F. Cabrera, "The Influence of Workload on Load Balancing Strategies," Proc. 1986 Summer USENIX Conference, Atlanta, GA, pp. 446-458, June 1986.

[Chow79]
Y. Chow and W. Kohler, "Models of Dynamic Load Balancing in a Heterogeneous Multiple Processor System," IEEE Trans. Comp. Vol. C-28, No.5, pp. 354-361, May 1979.

[Eager86a]
D. Eager, E. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Dynamic Load Sharing," Performance Evaluation, Vol.6, No.1, pp. 53-68, April 1986.

[Eager86b]
D. Eager, E. Lazowska, and J. Zahorjan, "Dynamic Load Sharing in Homogeneous Distributed Systems," IEEE Trans. Soft. Eng., Vol.SE-12, No.5, pp. 662-675, May 1986.

[Ezzat86]
A. Ezzat, "Load Balancing in NEST: A Network of Workstations," Proc. 1986 Fall Joint Computer Conference, Dallas, TX, pp. 1138-1149, November 4-6.

[Ferrari86]
D. Ferrari and S. Zhou, "A Load Index for Dynamic Load Balancing," Proc. 1986 Fall Joint Computer Conference, Dallas, TX, pp. 684-690, November 4-6.

[Ferrari87]
D. Ferrari and S. Zhou, "An Empirical Investigation of Load Indices for Load Balancing Applications", to be submitted, 1987.

[Hagmann86]
R. Hagmann, "Process Server: Sharing Processing Power in a Workstation Environment," Proc. 6th International Conf. on Distributed Computing Systems, Cambridge, MA, pp. 260-267, May 1986.

[Hwang82]
K. Hwang, W. Croft, G. Goble, B. Wah, F. Briggs, W. Simmons, and C. Coates, "A UNIX-based Local Computer Network with Load Balancing," IEEE Computer, Vol.15, No.4, pp. 55-66, April 1982.

[Johnston86]
W. Johnston and D. Hall, "UNIX Based Distributed Printing in a Diverse Environment," Proc. 1986 Summer USENIX Conference, Atlanta, GA, pp. 514-528, June 1986.

[Joy80]
W. Joy, "An Introduction to the C Shell," Computer Science Division, University of California, Berkeley, November 1980.

[Joy83]
W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual," Computer Systems Research Group, University of California, Berkeley, July 1983.

[Lazowska86]
E. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel, "File Access Performance of Diskless Workstations," ACM Trans. on Computer Systems, Vol.4, No.3, pp. 238-268, August 1986.

[Leland86]
W. Leland and T. Ott, "Load-balancing Heuristics and Process Behavior," Proc. Performance '86 and ACM SIGMETRICS Conf on Measurement and Modeling of Computer Systems, pp. 54-69, May 1986.

[Livny82]
M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," Proc. ACM Computer Network Performance Symposium, pp. 47-55, April 1982.

[MuKusick85]
K. McKusick, M. Karels, and S. Leffler, "Performance Improvements and Functional Enhancements in 4.3 BSD," Proc. Summer USENIX Conference, June 1985, Portland, OR, pp. 519-531.

[Rama83]
K. Ramakrishnan and A. Agrawala, "A Resource Allocation Policy using Time Thresholding," Proc. Performance '83, pp. 395-413, May 1983.

[Stone77]
H. S. Stone, "Multiprocessor Scheduling with the Aid of of Network Flow Algorithms", IEEE Trans. Soft. Eng., Vol.SE-3, No.1, January 1977, pp. 85-93.

[Stone78]
H. S. Stone, "Critical Load Factors in Two Processor Distributed Systems", IEEE Trans. Soft. Eng., Vol.SE-4, No.3, pp. 254-258, May 1978. pp. 47-55.

[Theimer85]
M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System," Proc. 10th SIGOPS Symp. on Operating Systems Principles, Orcas Island, WA, pp. 2-12, December 1985.

[Wang85]
Y. Wang and R Morris, "Load Balancing in Distributed Systems," IEEE Trans. Comp. Vol.C-34, No.3, pp. 204-217, March 1985.

[Wu80]
S. Wu and M. Liu, "Assignment of Tasks and Resources for Distributed Processing," Proc. COMPCON Fall 1980, pp. 655-662.

[Zhou86]
S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," Tech. Rept No. UCB/CSD 87/305, September 1986, also submitted for publication.