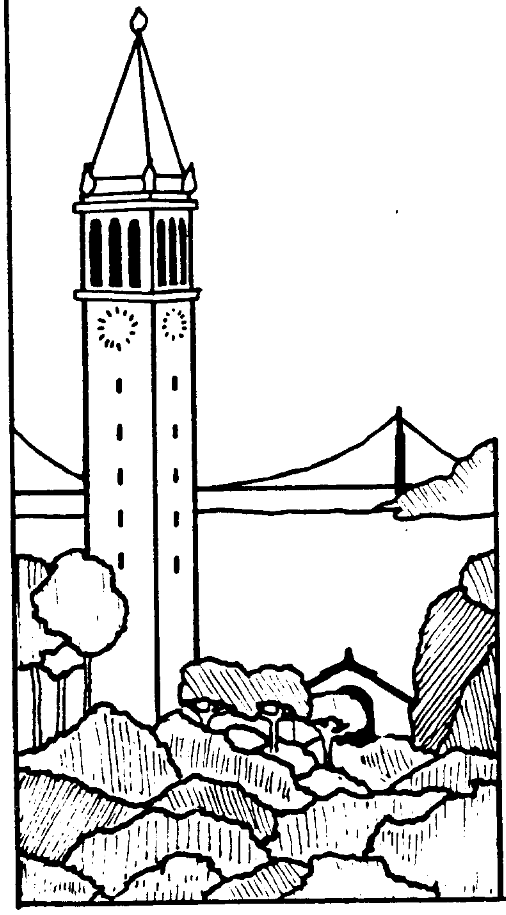
Load Balancing With Maitre d'

Brian Bershad



Report No. UCB/CSD 86/276

December 1985
PROGRES Report No. 85.18

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

Load Balancing With Maitre d'

Brian Bershad

Computer System Support Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

December 17, 1985

Abstract

As the number of machines in a computer installation increases, the likelihood that they are all being equally used is very small. We have implemented a load-balancing system to increase the overall utilization and throughput of a network of computers. With this system, a busy machine will locate an underutilized one and attempt to process certain types of CPU intensive jobs there. We present here a complete functional description of the system and an analysis of its performance.

1. Introduction

In any multi-machine computing installation, there is a need to evenly distribute the workload over the participating machines. Otherwise, some machines will remain idle while others become overloaded. The Berkeley UNIX environment, with its networking capabilities, provides for the inter-connection of powerful processors. The busier machines may move tasks to the less busy machines, offering a more even distribution of workload across the entire system, and a decrease in overall command response time.

This paper describes one load-balancing system currently in use in the EECS Department of the University of California at Berkeley called *Maitre d'*[1]. For a given class of relatively expensive jobs, *Maitre d'* will attempt to locate a lightly loaded machine and process the job at that machine. In this way, imbalances in processor demand across machines can be smoothed through an automatic redistribution of the load.

This paper is broken into several sections: background, functional description, operational considerations, implementation notes and performance analysis.

2. Background

Maitre d' was originally proposed by several students at Berkeley to relieve peak usage demands on the machines used by the Computer Science Division of EECS, particularly those dedicated to instruction. These machines had always been VAX 11/750s and 11/780s, characterized by extremely uneven workloads. Research and administrative machines alike suffered from high daytime loads, while being relatively idle at night. Instructional machines would go unused for long periods of time, but would become so loaded in the days prior to an assignment being due that they would become almost unusable.

In December 1984 the university received a gift of six VAX 11/750's from Digital Equipment Corporation[3] as part of a grant earmarked for undergraduate research and instruction. These machines were not ready for assignment to formal classes when the school semester began, but they were accessible through a 10 megabit ethernet [1,6] connection with the rest of campus. Consequently, the new VAXes were designated as remote process servers for overloaded instructional computers being rented by the Department, with *Maitre d'* acting as the agent.

2.1. Functional Description

Maitre d' operates around a modified state-broadcast algorithm. Every potential client maintains a list of known server machines. Associated with each server is a binary value representing that server's availability, as determined and advertised *by* the server. When a user invokes an application modified to run under *Maitre d'*, a decision is made as to whether the job should be performed remotely or on the user's machine by comparing the UNIX five minute load average against a minimum load threshold. [2] If it is determined that a remote machine should be used (local load average > sendoff threshold), the list of known servers is consulted, and the least recently used available server in the list is chosen to perform the job. That prospective server is contacted, informed of its selection and then told to perform the service.

The process of selection and contact can be made entirely transparent to the user. This is especially necessary in an instructional environment where most users are naive and unable to handle exceptional conditions (such as failure). They care only that their job gets done, and not where.

[1] *Maitre d'* comes from the French term *maitre d'hotel*, meaning hotel or dining room manager.

[2] The UNIX five minute load average is defined as the average number of jobs in the run queue exponentially smoothed over the past five minutes. It is a limited metric, but very cheap to obtain. For a more complete evaluation of load metrics, see [2].

Maitre d' can be used to off-load almost any type of non-interactive job. The initial version of *Maitre d'* exported pascal and C compiles. It has since been expanded to include typesetting (*troff*), circuit simulation (*spice*), a true Pascal compiler (*pc*), and others (see details in Appendix B). New applications are being added as need and opportunity arise.

3. How It Works

This section describes the operational details of *Maitre d'*. Readers uninterested in the mechanics of the system should skip to section four.

3.1. Establishing Connections

Before considering the total operation of *Maitre d'*, it is important to review some of the basics of interprocess communication. This section describes the establishment of connections and the relationship between the client and server machines. It assumes no familiarity with UNIX Interprocess Communications (*IPC*). We present a brief review of *IPC* under UNIX in the following paragraph. The reader unfamiliar with UNIX *IPC* may wish to consult either [5] or [7] for a more thorough description.

Separate processes may communicate with one another using any of several alternative methods. We describe here only the user-level protocol for a connected, bi-directional stream communication capable of crossing machine boundaries. The terms *client* and *server* are used to describe the parties during the connection phase. The server is generally referred to as a *daemon*, or a process which runs indefinitely, usually blocked while waiting for an event. The server daemon listens at some well-known address [3] waiting for requests for service. The client calls the server via some high-speed communications medium (in this case, the ethernet) and requests a connection. Implicit in this client's call is the requester's originating address. The server accepts by completing the connection with the client. Once the connection has been established, general practice is to have the server create a duplicate server process which interacts only with the initiating client, leaving the original server free to continue listening for further requests. Once the connection is established, both the client and server may read from and write to their common connection as though it were a standard UNIX file. This makes data transfer between the two processes extremely simple.

3.2. System Components

Load balancing under *Maitre d'* comprises four distinct components:

(1) *maitrd* [4]
All machines which are to be able to off-load jobs to other processors run a *maitrd* daemon. This process maintains the list of all server machines, including status information as to whether or not they are currently willing to accept jobs. For the remainder of this paper, the terms *maitrd* and *client* may be used interchangeably.

(2) *garcon*
This is the server daemon running on each machine which is to be a compute server. It has two functions: maintaining status connections with client machines, and accepting jobs from application programs. *Garcon* and *server* may be used interchangeably.

[3] An address is the combination of the machine's internet address and a local port number.

[4] *Maitre d'* (with a capital *M* and spelled correctly) is the name of the system. One of the component programs is called *maitrd* (with a lower case *m* and modified spelling).

(3) *application programs*

The *maitrd* and *garcon* components provide only a control environment through which the execution of programs may actually be distributed among many processors. The software that provides the interface between the user's requested task and *Maitre d'* is referred to as the *application.*

(4) *miscellaneous*

This includes the black-box library routines used to interface with *maitrd* (described later), and a dynamic control program called *mdc* used to tune parameters while the system is running (appendix C).

When a *maitrd* process is started, it tries to create control channels with the *garcon* daemons running at a pre-designated set of servers. This set is given in a startup configuration file (appendix B) kept at the *maitrd*'s machine. Through these control channels, the *maitrd* process can determine which machines are up and accepting jobs, which are up and not accepting, and which are down.

The counterpart to a *maitrd* process is the *garcon*. This daemon listens at its well-known control port for requests. When a connection from a *maitrd* is accepted, the *garcon* daemon lets the *maitrd* know whether the *garcon* host is willing to accept jobs. A server declares itself ready if its UNIX five-minute load average is less than some threshold *and* there are fewer than a given number of active users logged in to the server machine. Both of these thresholds can be set from a configuration file. After the connection is made, *garcon* checks its own status every 30 seconds, informing each of its connected clients (i.e., multicasting) whenever availability changes.

The typical configuration for *Maitre d'* is to have a set of machines in which each runs both the *maitrd* and *garcon* daemons. Each machine in the cooperative would look for help from others whenever it became too busy, and in return would be willing to take on jobs from its peers when it would otherwise be idle. One alternative to this is to introduce dedicated servers into the system (machines running only *garcon*), which accept jobs, but never send them out. A second alternative involves running a *clearinghouse maitrd* and is discussed below.

For each client/server pair, there is an open stream connection maintained for the duration of the relationship. This imposes a limit on the number of servers that may be kept track of by a single maitrd process[5]. Although not particularly cumbersome for a small number of machines, the total number of status connections for N clients and M servers grows as NM, and is order N^2 when all N machines are accommodating both *garcon* and *maitrd.* To slow this growth, *Maitre 'd* has the capability for *clearinghouse* clients. Instead of running a *maitrd* on every machine that is to offload, applications can request an available machine from a remote *maitrd.* This is most effective in clusters of diskless workstations. It is sufficient to have a single *maitrd* running on the file server handling requests from all of the file server's clients. Any reliability gained from having redundant *maitrd's* would be pointless, as the workstations aren't very useful when the file server is down.

Certainty of state is the main reason why open connections are used for the control channels. A status update from a *garcon* is guaranteed to arrive at each connected *maitrd.* If an update is undeliverable, *garcon* assumes the intended *maitrd* no longer exists and removes it from its multicast list. Similarly, if a *garcon* disappears, UNIX *IPC* enables each connected *maitrd* daemon to recognize this *immediately* and mark the associated server machine as unavailable. The *maitrd* daemon attempts to re-establish connections to downed servers every five minutes.

3.3. Selecting A Machine

The *maitrd/garcon* connection performs no real work. Its only purpose is to give the local *maitrd* a pool of processors from which it may choose a server. In addition to maintaining

[5] In 4.2BSD, this limit was set by the operating system at 20. The newer 4.3BSD allows 64.

connections with remote servers, *maitrd* also listens in on a second, local socket for requests from an application program, which is any program that has been modified to run under *Maitre d'*. These applications first connect to the local *maitrd* process, asking for an available machine. If the local load is less than the sendoff threshold, or no remote servers are presently available, the application is told to perform the job locally. Otherwise, *maitrd* does a round-robin traversal of its list until it finds a machine where the *garcon* has advertised a willingness to accept jobs. It passes back to the application program the internet address of this server and terminates the local connection with the application.

3.4. Program Execution

In addition to listening on its control port, *garcon* also listens on a data or service port. It is this address that *maitrd* gives to the application, and it is the responsibility of the application to create the connection. Once the connection is created, the *garcon* process creates a copy of itself. This copy communicates with the local application and executes the requested task on the remote machine, leaving the original *garcon* free to handle further requests from other applications. All communication between the application and the server is done through this data port. Commands and data are passed from the application process to the remote machine; results and an exit status are passed back from the server to the application process.

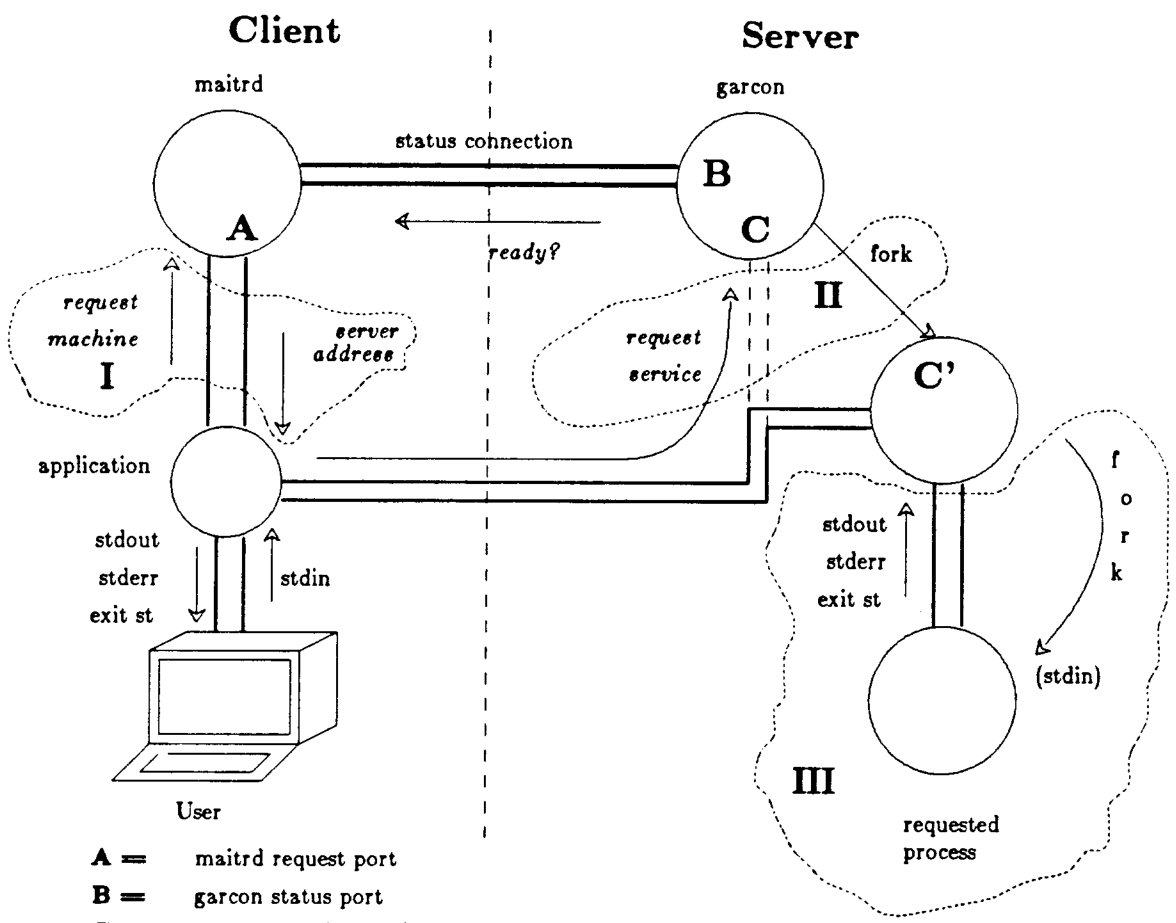
Figure I shows a typical *Maitre d'* interaction occurring in three stages. Permanent communication streams are shown in thick black; temporary ones in thin. In stage I the user invokes some application (compiler, formatter, etc.) which connects to the local *maitrd* and requests an available server. This *maitrd* has been maintaining status connections with many *garcons* (only one shown in the picture). The client daemon passes a server address out of its *maitrd* port (**A**) to the application program and terminates its connection with the application. Stage II shows the application connecting to the *garcon* port (**C**) that was returned by the local *maitrd* and requesting a service. If the application and request are valid (see sec. 4.1.2), the server creates a copy of itself (called *forking* in UNIX). Note that the service port is passed down to the newly created dedicated child process (**C'**). The dashed lines leading into the *garcon* port indicate that the parent stops attending to the application on the other end of the channel once the child *garcon* has taken over. Stage III has the dedicated *garcon* creating the process to perform the requested task and acting as a data buffer between the application program and that task. When the requested process finishes, its exit status is returned to the application at the originating machine, and the child *garcon* terminates.

3.5. I/O Handling

The C-shell provides every process with three default *file descriptors* or data channels: standard input *(stdin)*, standard output *(stdout)* and standard error *(stderr)*. *Stdout* and *stderr* are both output channels; *stdin* is the only input channel. Many UNIX programs are capable of taking their input from *stdin*, and placing their output on *stdout*. Convention has error messages going to *stderr*. All processes return an 8 bit status value upon termination.

Stdin to a *garcon* task is passed directly from the originating machine. But, on the return path of the service channel, *Maitre d'* provides only one data stream. The extra bandwidth needed to handle *stdout, stderr* and the exit status is obtained by returning all data from the server in finite packets, and prepending each packet with a four byte header. The first byte indicates the type of data being returned: *stdout*, *stderr* or exit status. If the header indicates an exit status, the second byte indicates how the process exited, and the third byte is the exit status. Otherwise, the final three bytes in the header give the size of the packet to be sent. The dedicated *garcon* process, acting as a buffer between the application and the remote task, handles the data encoding.

Figure I



A =	maitrd request port
B =	garcon status port
C =	garcon port (request)
C' =	garcon port (service)

3.5.1. The Black Box

Although the sequence of connection, selection, and output decoding required by every application program appears complicated, there is a function called **RemoteRun** which does it all:

```
RemoteRun(inFD, outFD, cmdp);
int inFD;
int outFD;
struct pipepiece *cmdp;
```

where

```
struct pipepiece      {
        char   *pp_name;
        char   **pp_args;
}
```

and **inFD** and **outFD** are the input and output file descriptors that should be used for input to and output from the remote task. There are no provisions for redirecting *stderr*. Note that **cmdp** is a *pointer* to **struct pipepiece**. A list of these structs indicates a sequence of piped commands, allowing multiple tasks to be piped together on the remote end. Appendix A shows a typical interaction with **RemoteRun.**

4. Operational Considerations

4.1.1. Error Handling

Maitre d' itself does not have any provision for handling errors other than reporting them to the application. Some common error situations are:

- lost connection with remote host
- remote *garcon* prematurely exited
- remote machine could not execute process

Whenever possible, an application should recover from the error and accommodate the user's task as quietly and politely as possible. This may be done either by finding another host, or by processing the job on the user's machine. All of our applications attempt to run the job locally if there is a remote failure.

Because of redirection facilities and pipes in UNIX, a difficult situation arises if the remote server is the recipient of a local process's output, and one of the errors listed above occurs. The process cannot be restarted in any way. For example:

tbl file.me | matfront nroff -me > file.out

where *matfront* is a generic front end that attempts to process its arguments on a remote machine using *stdin* as input. Once data from *tbl* is passed to *matfront*, it cannot be retrieved if the remote *nroff* terminates due to some error related to *Maitre d'*. Even if *matfront* kept a copy of *tbl's* output (which would be prohibitively expensive), output already generated by the remote *nroff* would have gone to *file.out*. Any attempt to restart the job might cause duplicate output to appear in *file.out*. Because *Maitre d'* operates at the user level, above the C-shell and UNIX kernel, no simple solution exists for this problem. Consequently, if an error occurs after a remote job has started executing, and the job is receiving its input from a pipe, the user's task terminates with an error message. Processes not using redirection do not have this problem, and can be restarted

4.1.2. Security

When a machine services users on other machines, various security problems arise. Some of the concerns are:

Unauthorized Use
Administrative barriers must be respected.

Unauthorized Access
Use of spare cycles should not allow access to restricted files.

Unauthorized Execution
Not all machines should have to provide all services to all clients.

Maitre d' solves these security problems with client verification, task verification, non-privileged users and logfiles.

When a connection is requested to either of *garcon's* two ports, the originating host is checked against a list of authorized hostnames contained in the startup configuration file. The request is ignored if the host is not authorized and the illegal access attempt is recorded in a log file.

If a request for service arrives, *garcon* guards against unscrupulous applications by checking to see that it has actually advertised itself as *available.* If not, the request is ignored. The request is then checked against a list of *reasonable* services that *garcon* has been told about in the configuration file. If it is not in the list, *garcon* informs the application that it doesn't know about the service. It is then up to the application to either choose another server or process the job locally.

Associated with each process under UNIX is a user name governing that process' access rights. When a task runs under *garcon,* the privileges are first set to those of some named user given in the configuration file. In Berkeley's implementation, all service tasks run as *nobody,* which is an actual entry in the password file originally created for system administration functions. *Nobody* has no password, home directory or shell and can only read public files. If process accounting is being run on the server machines, it would be worthwhile to create a dummy account used only by *garcon.* In this way, the standard accounting software could determine the percentage of resources which are being used on behalf of remote requests.

Once a server has begun a remote job, and if its load has risen above the acceptance threshold, it is possible to have this job's priority lowered or re-*niced* during the period that the server is not accepting new jobs. Active jobs from other machines will then not impinge upon jobs coming from a server machine's own users. The priority is raised again once the server's load falls below the threshold.

5. Implementation Notes

All programs that are to operate under the *Maitre d'* system require a front end (using **RemoteRun**) on the client machine to decode the returned data. Those applications that use *stdin, stdout,* and *stderr* for I/O do not need a front end on the remote machine, and may simply use *matfront* as an interface. Since *Maitre d'* supports only these three channels of data transfer, a few programs did not easily integrate into the system.

5.1. The Compilers

5.1.1. C

Creating the application to handle C compiles was relatively trivial due to the structure of the compiler, which is broken down into four distinct parts: pre-processing, compilation, assembly and loading. Pre-processing and loading are always performed on the local machine. The compilation and assembly, which comprise almost 70% of the total cycles, are done on the remote machine.

5.1.2. Pascal

A good example of a package that did not lend itself well to running under *Maitre d'* was the Berkeley Pascal interpreter (*pi*). There were problems on both the client and server end.

Server

Berkeley Pascal (*pi*) does not use *stdin* for anything, but instead requires that its input come from a file (or several files). The executable image produced by *pi* goes directly to a file and cannot be directed elsewhere (such as *stdout*). Things are further complicated by the fact that *pi* uses *stderr* for only one error message. All other error messages go to *stdout*. This causes problems for *garcon* which expects remote tasks to communicate back to the application via *stdout* & *stderr*.

Client

Because of the way *#include*[6] files are handled in *pi*, it is semantically legal to concatenate all of the files and compile them as a single stream. Originally, we ran a very simple Pascal pre-processor over all of the user's source, scanning for *#include* directives and including them as we found them, sending over all of the files as one large program. Unfortunately, all of the debugging and diagnostic errors produced by Berkeley Pascal have line numbers relative to the beginning of *each* source file, so this was not a satisfactory solution. Students were being told that they had a syntax error on line 1237, when their largest file contained only 250 lines.

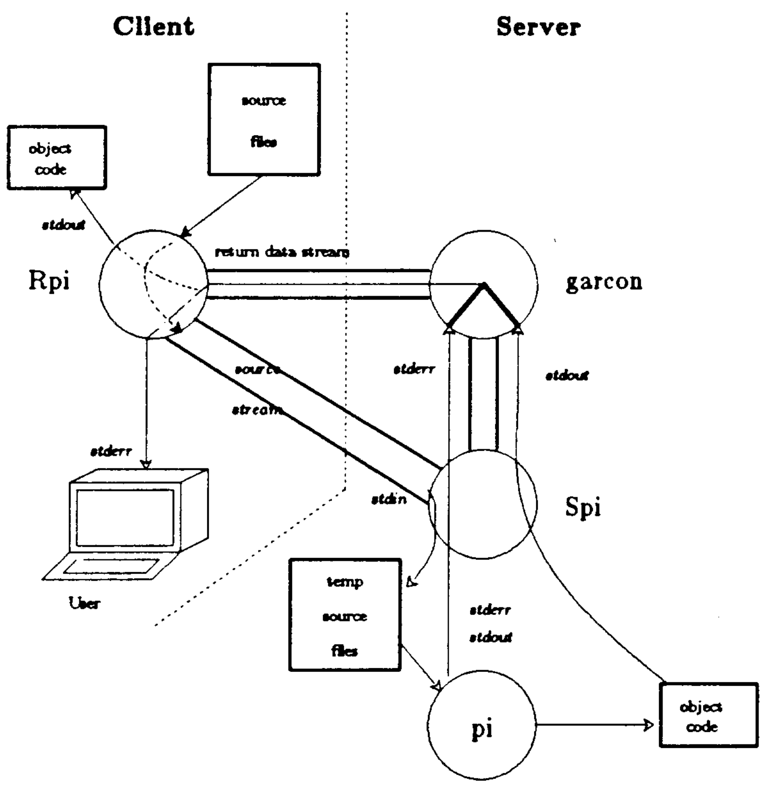
These problems were solved by building a simple file transfer protocol on top of the three data channels already provided for by *Maitre d'*. This extra level required another layer of pre-processors on both the client and the server machines. A user runs *Rpi* on the local machine which reads the user's program, looking for *#include* directives. Instead of starting up *pi* on the server machine, *Rpi* instead requests that *Spi* (server *pi*) be executed. Data going from *Rpi* to *Spi* includes a header giving the size of the file, the length of the file name, the file name, and finally the file. This is done for every file that is needed in the compilation. *Spi* reconstructs the original source code in a temporary directory on the server machine. When all files have been received, *Spi* executes *pi* on those files in the temporary directory and the compilation is performed. *Spi* takes care of transferring *pi*'s *stdout* to *stderr*. If the compilation completes successfully, *Spi* reads the executable image left behind by *pi* and sends it to *stdout*. This is picked up by *garcon* and sent to *Rpi*, which knows that anything coming to *stdout* from the remote machine belongs in an executable file on the local machine. The relationships between the processes, machines and files are shown in figure II.

Maitre d' provides only a primitive connection mechanism through which a task on a remote machine may communicate with its calling process on the local machine. For tasks flexible enough to operate using only three data channels, the mechanism is sufficient. The point of this discussion on the Pascal application is to show that those tasks requiring more complicated file access *can* be accommodated by building on top of the interface provided. It may seem cumbersome and expensive to transfer all of a user's source files from client to server for each application's invocation. But, until a reliable remote file system becomes widely available, this type of explicit data transfer is necessary.[7]

[6] When the interpreter encounters a line of the form "*#include filename*" in the source file, it reads the named file into its input stream as though it were coded in-line by the user.[4]

[7] Even with a remote file system, the data would still need to be moved between machines. The transfer would just be completely transparent.

Figure II



6. Evaluation & Performance

6.1. Design Criteria

Alonso[2] describes six points for choosing a good load-balancing strategy: stability, implementability, cost, autonomy, transparency and tunability. *Maitre d'* meets all of these criteria.

(1) *Stability.* A server machine could become inundated with requests from clients if it had recently announced its availability. As instantaneous state information is very hard to come by, use of an algorithm that avoids processor flooding is very important. *Maitre d'* relies on a round-robin selection mechanism of available hosts to minimize the possibility that any one client might overload a server. Although it is possible for several clients to all simultaneously request the same server, this type of selection synchronization is highly unlikely.

(2) *Implementability*. *Maitre d'* runs at the user level and requires no modifications to the UNIX kernel. It is running on VAX 750s, 780s, 785s, micro VAX IIs, and a Sequent parallel processor. The basic package is about 4000 lines of well-commented C code (approx. 50K bytes object) and is portable to any machine running Berkeley UNIX 4.2 or 4.3BSD.[8]

(3) *Cost*. The overhead in running *Maitre d'* is minimal. There are three considerations for cost: client overhead, network traffic and server overhead. In the worst case, when no servers are available, the user's process usually requires less than .5 seconds of real time to determine that the job must be performed locally. Since the class of jobs running under *Maitre d'* are typically long-lived, this time is unnoticeable (but consistent). Traces show that it is rare for jobs to require more than a few hundred kilobytes of data to be transferred between client and server machines. On a 3 or 10 megabit ethernet, the impact of the extra traffic is minimal. Since servers only broadcast *changes* in state information and spend most of their time blocked waiting on requests, the cost of running the server is also low. Using acceptance load thresholds of 5 and 2 on our VAX 785's and 750's respectively[9], each server was averaging about 40 state changes a day.

(4) *Autonomy*. The decision to accept or reject jobs is *completely* up to the server, so no machine can be forced to take on work from outside clients. In addition, the types of jobs a server will accept, and the client machines from which those jobs may arrive is definable by the system administrator in a configuration file. The server machine may also impose a pre-emption policy on remote jobs, always giving priority to its own users.

(5) *Transparency*. The fact that a process is being executed on a remote machine *can* be made completely transparent to the user. Users need never know that their tasks are being performed elsewhere.

(6) *Tunability*. System parameters can be set and reset through a configuration file and a dynamic control program (*mdc*). Thus, *Maitre d'* can be run in various environments without the need for recompilation.

6.2. Performance

Several factors contribute to the success of *Maitre d'*. First, the decision to export only high-cost, CPU intensive jobs allows a machine, regardless of how busy it might be, to provide swift response time for those jobs. As the less expensive, non-ported jobs are no longer competing with the more costly ones for resources, they too enjoy an improvement in response time. Table I shows comparative statistics for April 1984 and April 1985 taken on a VAX 11/780 (ucbcory). In 1984, the machine ran without *Maitre d'*. In April 1985, the only jobs being offloaded were Pascal and C compiles. About 3000 compiles per week were being exported to two VAX 750's operating as dedicated remote servers. Given are the UNIX five minute load averages; the times to start up the editor on a trivial file; and the times (in seconds) to compile and execute locally the following short CPU intensive program:

[8] There were some problems initially with the Sequent, but it turned out to be a bug in their 4.2 release and not in *Maitre d'*.

[9] Experience has shown these loads to be comfortable operating points. Below these values, the machines tend toward idleness. Above them, response time becomes intolerable.

```
main()
{
        int i,j,m;
        for(i=0 ; i<1000 ; i++)
              for (j=0 ; j<1000 ; j++)
                    m=m+1;
}
```

The demands on the machine from instructional coursework were identical for the two months being compared. The increase in performance can be seen across the board in all the statistics, with an average improvement factor of over 2. The increase in perceived machine performance is even more dramatic considering that the VAX 780 was running with 16 megabytes of memory in 1984, but only half that during the 1985 sampling period. There were no other configuration changes. The decrease in variance for all the figures demonstrates that a balanced system has the added feature of offering more predictable response times.

Table I
VAX 11/780 Performance Comparisons

ucbcory	**April 84 (w/o Maitre d')**	**April 85 (w/ Maitre d')**	**Improvement Factor**
# samples	2140	2134	-
mean users	20.01	19.31	-
median users	20	19.74	-
variance users	97	100	-
mean load	6.12	2.52	2.42
median load	4.6	1.8	-
variance load	23.95	6.07	-
mean editor (secs)	1.46	.82	1.78
median editor	1	1	-
variance editor	1.87	.362	-
mean compile (secs)	11.90	7.04	1.69
median compile	8	6	-
variance compile	94.6	20.42	-
mean execution (secs)	63	26.7	2.35
median execution	30	14	-
variance execution	5564	1142	-

The presence of lightly loaded machines, and *Maitre d's* ability to locate them also contributes to its success. Across even as few as six machines, the likelihood that one of them will be idle at any given time is fairly high. This is partially because of the type of workload imposed by instructional computing, i.e., a machine is *very* busy shortly before an assignment is due, and much less so at other times. This can be seen in appendix F, where the five minute load average (*Maitre d's* key value) is given for six machines over a one month period. These figures were taken in March 1984 on machines not running *Maitre d'*. After *Maitre d'* was in place, our busiest machine (a VAX 11/785) performed 66709 compiles, 38524 of them remotely, in just 96 days. The less powerful 11/750's would average between three and five thousand compiles per machine per week. An overloaded machine looked to find an idle remote processor without success on only 98 occasions. This indicates that, for the most part, at least one lightly loaded machine could always be found among the six possible servers.

One metric used to gauge the relative utilization of machines over a long period of time is the *mean* load average. This is simply the average value of a machine's load average when sampled at regular intervals. From this value we also found:

(a) that those machines whose mean load average before *Maitre d'* was less than *garcon's* acceptance threshold value tended to have their mean increase after load sharing was in place, and

(b) that those machines whose mean load average was above *maitrd*'s sendoff threshold saw a decrease in their mean load average, as well as a marked decrease in the variance of the load average.

This simply means that those machines most in need of assistance will benefit the most from load sharing, and machines which were idle will find themselves more busy. This is exactly what should be happening and is not surprising.

7. Conclusions

We have implemented an effective load-balancing system and demonstrated its utility. Further applications can be easily introduced to operate within the package's environment. Performance metrics indicate that this system is highly successful in creating a more responsive and pleasant working environment for users.

8. Acknowledgments

This work was made possible in part by a donation of equipment from Digital Equipment Corporation and the support of the State of California MICRO Program[3]. It was also supported in part by the National Science Foundation under grant DMC-8503575 to Professor Domenico Ferrari. The original version of *Maitre d'* was initially proposed and authored by undergraduate students Chris Williams and Chris Guthrie[8]. Professors Richard Fateman, Lou Katz and Domenico Ferrari should be credited for their patience in seeing this project through to the end. Finally, special thanks to Doug Cooper and Harry Rubin for their tireless advice on this paper's organization and presentation.

Sites interested in obtaining the *Maitre d'* load balancing software should contact the author in care of the Computer Systems Support Group (CSSG) at UC Berkeley.

Appendix A

Using RemoteRun

```
#include "defs.h"           /* for meaning of RemoteRun's return values */


main (argc, argv)
int argc;
char **argv;
{
	int r;
	int verbose = 0;
	int midway = 0;
	struct pipepiece PipeList[10];

	....

	MakePipeList(argv, PipeList);      /* automagically converts argv */
				     /* to pipe list */

	....

	r = RemoteRun(0, 1, PipeList);     /* inFD == stdin */
				     /* outFD == stdout */

	if (r) {
		if (r == -1)  {
			if (verbose)
				fprintf(stderr,"%s: Error in connecting to remote machine\n",argv[0]);
			do_local(argv);
		} else if (r == -X_ABORT) {
			fprintf(stderr,"%s: Remote server killed by signal\n.",argv[0]);
			midway++;
			verbose++;
			do_local(argv);
		} else if (r == -X_EXEC)    {
			fprintf(stderr,"%s: Remote failed to exec.\n",argv[0]);
			verbose++;
			do_local(argv);
		} else if (r == -X_NOFND) {
			if (verbose)
				fprintf(stderr,"%s: Remote doesn't know about this command.\n",argv[0]);
			do_local(argv);
		} else if (r == X_LOCAL)    {
			do_local(argv);
		} else if (r == X_BADENV)   {
			midway++;
			verbose++;
			do_local(argv);
		} else if (r == -X_MIDWAY)  {
			fprintf(stderr,"\n\007\007%s: Lost connection with remote host\n",argv[0]);
			midway++;
```

```
				verbose++;
				do_local(argv);
			}
		}
		exit(r);
}


do_local(argv)
char **argv;
{
		/*
		 * can only continue if input coming from file and not
		 * a pipe and the process had not yet started.
		 */
		if ( midway && !(isatty(0) && isattty(1)) )	{
			fprintf(stderr,"Broken pipe (remote)\n");
			exit(-1);		/* user outta luck */
		}

		if (verbose)
			fprintf(stderr,"Processing locally\n");

		execvp(*argv, argv);

		/* NOTREACHED */
		perror("matfront:");
		fprintf(stderr,"Could not execute %s\n", *argv);
		exit(-1);
}
```

Appendix B

Sample Configuration File

```
# UCBSIM
#
# Maitrd Load Balancing Configuration File
#
#             Instructions available to outside world
#                    I<command name command path proc uid>
#             Possible Client
#                    C<ucbxyzzy>
#             People Servers for me
#                    S<ucbxyzzy>          ... no entries means all
#             Entry both a client and a server
#                    B<ucbzyzzy>
#             Garcon configuration option
#                    G<option>     <value>
#             Clearing House Machine
#                    m<ucbxyzzy>
#
# These machines are allowed in:
Cucbear
Cucbeast
Cucbdali
Cucbarpa
# These machines should be willing to take jobs from me:
Sucbfranny
Sucbzooey
Sucbsim
Cucbbuddy
# This guy takes and receives:
Bucbernie
#
# Request from        (localhost == ourselves)
mlocalhost
#
##########################################################
#             Commands We Can Run
#
#
# Anything here can be run by people on the outside
# First column = name by request
# Second column = path of associated program
# Third column = user name for task
#
#       Icsh /bin/csh root           would be a bad entry
#
#
Iccom         /lib/ccom            nobody
Inroff        /usr/bin/nroff       nobody
Icat          /bin/cat             nobody
Idate         /bin/date            nobody
Iecho         /bin/echo            nobody
Iwhoami       /usr/ucb/whoami             nobody
```

Appendix B

```
Ihostname     /bin/hostname        nobody
ISpi          /usr/local/Spi       nobody         # pascal front end
IScc          /usr/local/Scc       nobody         # C front end
Itroff        /usr/bin/troff       nobody
Itroff_p      /usr/local/troff_p   nobody         # ditroff
Irwho         /usr/ucb/rwho        nobody
Ispice        /cad/bin/spice       nobody         # circuit simulation
#
#
####################################################################
#
#                     Server runtime options....
#
# Load threshold above which jobs are not accepted
Gload         3.0
# Number of non-idle users above which jobs are not accepted
Gusers        20
# Maximum number of clients allowed in
Gclients      15
# When the load exceeds threshold, raise priority (renice) of
# active jobs to argument.  In unix, this is analogous to preemption.
Gnice         4
```

Appendix C

Selected Manual Pages

Rpi -	Remote pascal interpreter
Rcc -	Remote C compiler
mdc -	*Maitre d'* dynamic control program

NAME

Rpi – local front end for using pi (pascal) with *maitrd*
Spi – server front end for using pi (pascal) with *garcon*

SYNOPSIS

Rpi [+gvd +o [file] -[pi options]]
Spi [+gv -[pi options]]

DESCRIPTION

Rpi is a "user- friendly" interface for using pi with the maitrd/garcon load balancing software. It finds an available machine for processing, transfers files needed by pi, and brings back any output generated by the remote pi. Before beginning the compile, *Rpi* copies any existing obj file to obj.bak.

In its simplest form, it is invoked just as pi is:

```
Rpi prog.p
```

Since pi requires as input a file ending in .p, Rpi does not call up pi immediately, but instead requests to run *Spi* on the server machine. *Rpi* and *Spi* transfer needed source files and new object files to one another. *Spi* creates a temporary work directory on the server end. It is here where the pi is actually performed. Pi error messages are routed back to the user through stderr. If the compilation was successful (an obj file exists in the remote work directory), the obj file is moved back to the current working directory. Only a single status message from the remote machine gives any indication that the program is not being compiled locally.

Spi can not be used directly, as it expects data to be headed with filestat information. In this manner, multiple files can be passed though a single pipe.

Pascal *include* files cause numerous headaches with remote processing. The source files must all be scanned for #include directives. As all work is done in a single directory on the remote end, including files from a directory other than the current one can cause "minor" problems. File pathnames are modified to replace all occurrences of '/' with '\' to maintain unique names in the flat name space on the remote machine. When the compilation completes, the executable's symbol table is examined for all references to '\'. These are changed back to '/'. Munging the filename in this way has two consequences. A filename is not allowed to include the special character '\', and all error messages generated by the compiler referencing the munged file will have '\'s replacing '/'s. So, an error in file /a/b/c/d/file.i will be reported as an error in \a\b\c\d\file.i.

OPTIONS

+g When the server has completed compiling, it will normally clean up whatever workspace it required. If the +g option is used, garbage generated by the compilation will be left behind, and the workspace path will be given upon program termination. This is only really useful for debugging.

+o Normally, *Rpi* output will go into obj. This option allows you to specify the output file.

+v This puts *Rpi* into verbose mode. It can sometimes be fun to watch if you like this sort of stuff.

+d This sets a debugging flag. It is not used for much. +v is better.

–[pi options]
Any switches preceeded by a – will be passed through untouched to the pi program.

ERRORS

If, for any detectable reason, the remote end can not perform the compilation (host dies, failed exec, etc...), the compilation will be performed locally.

SEE ALSO

maitrd(1), garcon(1), matfront(1), mdc(1), socket(2), pi(1)

BUGS

The translation between '/' and '\' is annoying.
The (g)arbage switch is really useless unless there is the capability to return the entire temporary directory back to the local machine.
Error messages from the compiler are sent to standard error. Any output from Rpi must be redirected with "|&" or ">&".

NAME

Rcc – Remote C Compiler
Scc – Server C Compiler

SYNOPSIS

Rcc [option] ... **file** ...

DESCRIPTION

Rcc is the UNIX remote C compiler. *Rcc* accepts several types of arguments:

Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with '.s' are taken to be assembly source programs and are assembled, producing a '.o' file.

Rcc (capital R) is not the same as rcc (lower case r). The latter fires up shells on remote machines and requires that the user have an account there. Rcc is written to work with the Maitrd remote server software and requires only that there be machines willing to accept job requests. When invoked, Rcc checks to see if the load is low enough to run the compile locally. If so, Rcc acts just like cc. If the local load is too high, Rcc will attempt to locate a foreign machine that is not too busy and execute much of the compilation at the remote host. If such a machine can not be found, or if the connection becomes 'flakey' or lost, Rcc will force the remote compilation to be done locally. Scc is used to perform the remote compilation and is called automatically from Rcc. Scc should not be run stand-alone.

C preprocessing (cpp) and loading (ld) are always done locally. Only compilation (ccom) and assembly (as) are done on the remote machine. If Rcc is invoked with the -S option, assembly on the remote end will be bypassed. If the input file is only an assembly source program (ends in '.s'), Rcc will not even attempt to assemble it remotely. It will be assembled locally.

The following options are interpreted by *Rcc* just like in *cc*. See *ld*(1) for load-time options.

–c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

–g Have the compiler produce additional symbol table information for *dbx*(1). Also pass the **–lg** flag to *ld*(1).

–go Have the compiler produce additional symbol table information for the obsolete debugger *sdb*(1). Also pass the **–lg** flag to *ld*(1).

–w Suppress warning diagnostics.

–p Arrange for the compiler to produce code which counts the number of times each routine is called. If loading takes place, replace the standard startup routine by one which automatically calls *monitor*(3) at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

–pg Causes the compiler to produce counting code in the manner of **–p**, but invokes a run-time recording mechanism that keeps more extensive statistics and produces a *gmon.out* file at normal termination. Also, a profiling library is searched, in lieu of the standard C library. An execution profile can then be generated by use of *gprof*(1).

–O Invoke an object-code improver.

–R Passed on to *as,* making initialized variables shared and read-only.

–S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.

–E Run only the macro preprocessor on the named C programs, and send the result to the standard output.

–C prevent the macro preprocessor from eliding comments.

–o *output*
Name the final output file *output*. If this option is used the file 'a.out' will be left undisturbed.

–D*name***=***def*
–D*name*
Define the *name* to the preprocessor, as if by '#define'. If no definition is given, the name is defined as "1".

–U*name*
Remove any initial definition of *name.*

–I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **–I** options, then in directories on a standard list.

–B*string*
Find substitute compiler passes in the files named *string* with the suffixes cpp, ccom and c2. If *string* is empty, use a standard backup version.

–t[p012]
Find only the designated compiler passes in the files whose names are constructed by a **–B** option. In the absence of a **–B** option, the *string* is taken to be '/usr/c/'.

–d Run in debugging mode showing the names of intermediate files as they are created.

–v Run in verbose mode, with the remote end indicating its actions as it goes along.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.**

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/cpp	preprocessor
/lib/ccom	compiler
/usr/c/occom	backup compiler
/usr/c/ocpp	backup preprocessor
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/usr/lib/gcrt0.o	startoff for gprof-profiling
/lib/libc.a	standard library, see *intro*(3)
/usr/lib/libc_p.a	profiling library, see *intro*(3)
/usr/include	standard directory for '#include' files
mon.out	file produced for analysis by *prof*(1)

gmon.out file produced for analysis by *gprof*(1)

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978
B. W. Kernighan, *Programming in C—a tutorial*
D. M. Ritchie, *C Reference Manual*
monitor(3), prof(1), gprof(1), adb(1), ld(1), dbx(1), as(1), maitrd(l), garcon(l)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Any problems occurring due to the migration of program source should result in the compilation being performed locally. Unless the -v option is set, this transition should be completely transparent.

BUGS

The compiler currently ignores advice to put **char**, **unsigned char**, **short** or **unsigned short** variables in registers. It previously produced poor, and in some cases incorrect, code for such declarations.
Each file on the argument line is compiled separately, so the final program may have been compiled on many different machines. This can be considered as either a bug or a feature.

NAME

mdc – maitrd control program

SYNOPSIS

/usr/local/mdc [command [argument ...]]

DESCRIPTION

Mdc is used by the system administrator to control the operation of the maitrd load balancing software. For any machine running the maitrd client daemon, *mdc* may be used to:

- set the dynamic load threshold at which jobs are exported,
- force the daemon to reread the configuration file, restarting all active connections, and attempting to reestablish dormant ones,
- display the current status of the maitrd daemon,
- kill the daemon without restarting it.

Without any arguments, *mdc* will prompt for commands from the standard input. If arguments are supplied, *mdc* interprets the first argument as a command and the remaining arguments as parameters to the command. The standard input may be redirected causing *mdc* to read commands from file. Commands to *mdc* may be sent to any machine running the maitrd software. If no machine is given explicitly on the command line, *mdc* directs the command to the last referenced machine. If no machine has yet been referenced, then the local host is assumed. Any number of hosts may be given on a command line. *Mdc* will send the command to each host. Commands may be abbreviated; the following is the list of recognized commands.

? [command ...]

help [command ...]
: Print a short description of each command specified in the argument list, or, if no arguments are given, a list of the recognized commands.

kill { (host)* }
: Terminate the active maitrd daemon at the host (or hosts) immediately. This command is restricted to the superuser.

load # { (host)* }
: Set the load threshold to the second argument at each of the indicated (or implied) hosts. This command is restricted to the superuser.

exit

quit
: Exit from mdc.

restart { (host)* }
: This will cause the maitrd daemon at the indicated hosts to reread the configuration file, close all existing connections and attempt to reestablish connections with each host given in the configuration file. All in-core statistics are zeroed. Jobs in the middle of processing are unaffected and will continue normally.

status { (host)* }
: This displays the status of the maitrd daemon at each indicated host. This is an unrestricted command.

machine { host }
: This sets the default host to its argument. With no arguments, it returns the current default host.

FILES

/usr/local/maitrd.conf maitrd configuration file

SEE ALSO

maitrd(l), garcon(l), Rpi(l), Rcc(l)

DIAGNOSTICS

?Ambiguous command	abbreviation matches more than one command
?Invalid command	no match was found
?Privileged command	command can be executed by root only

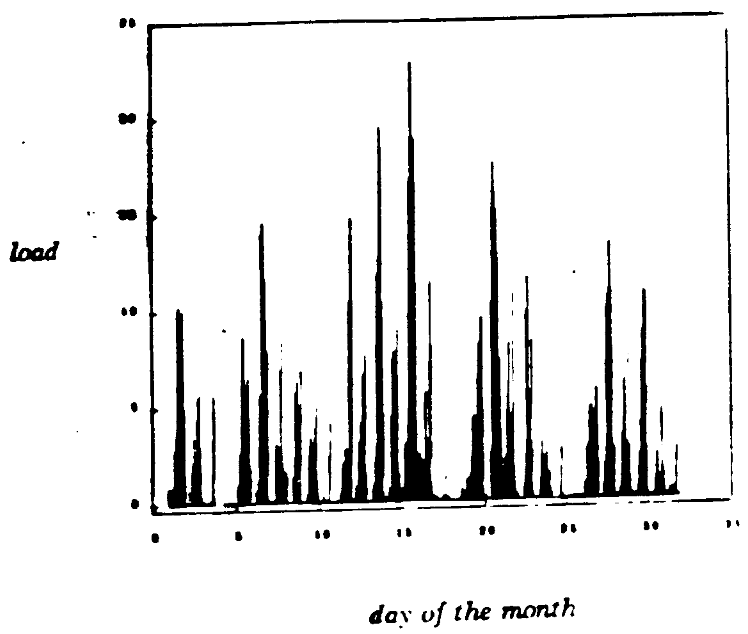
BUGS

The 'machine' command is probably silly.

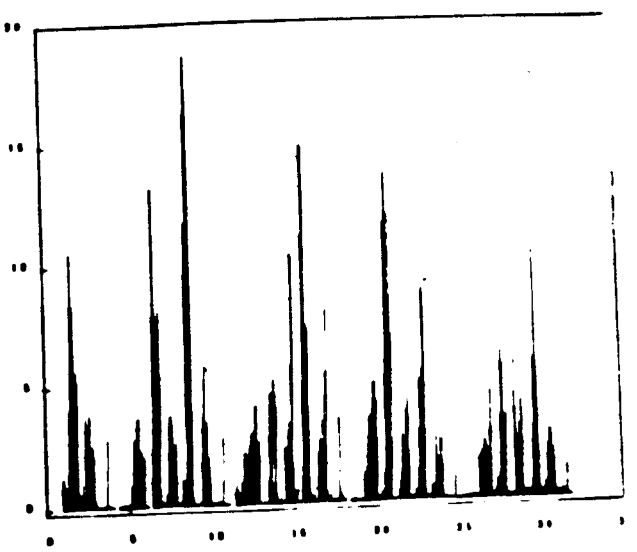
Appendix F

Load Averages For March 1984

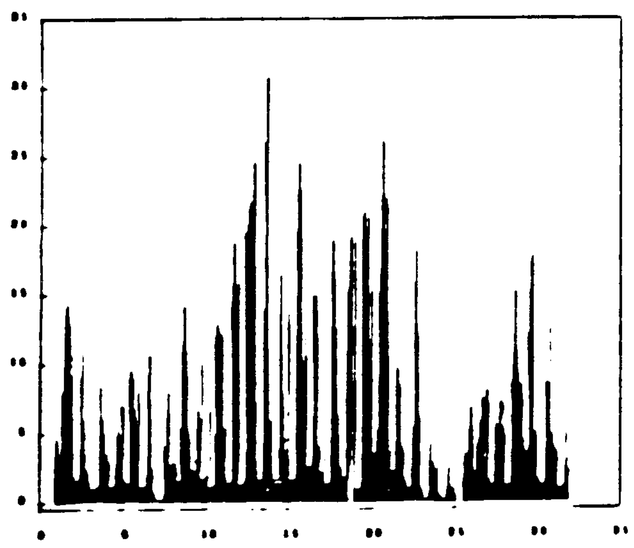
VAX 750



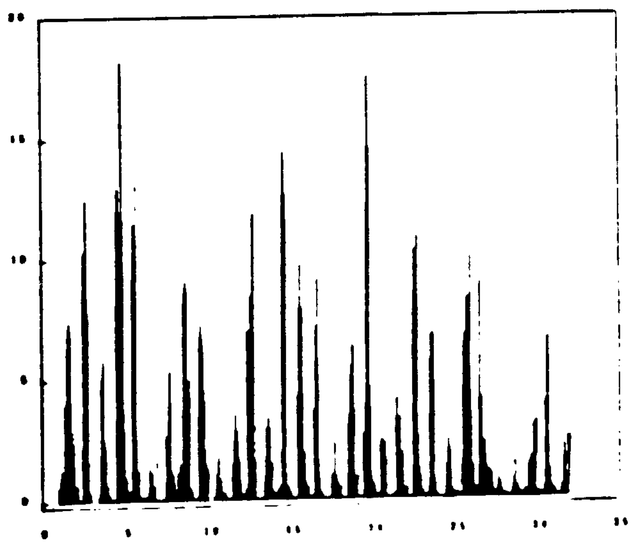
VAX 750



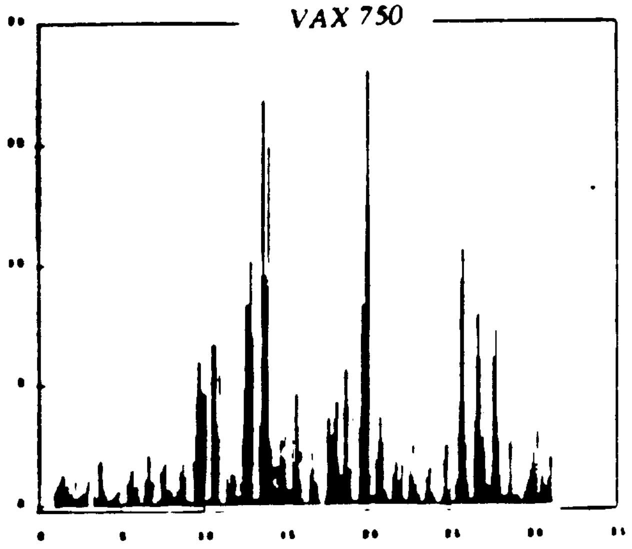
VAX 780



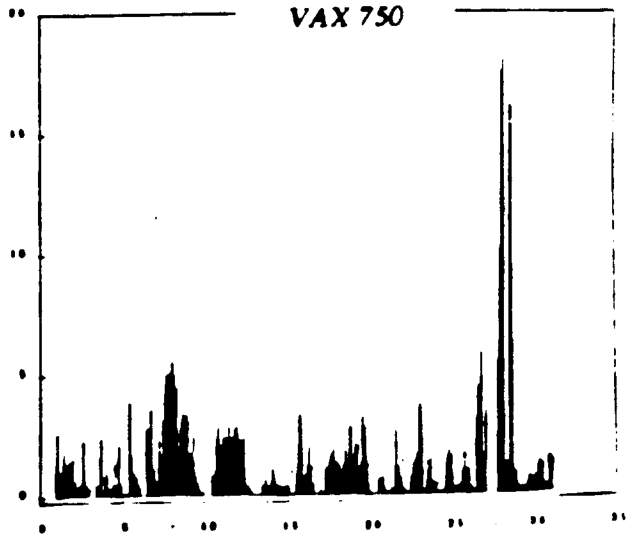
VAX 750



VAX 750



VAX 750



References

(1) G. Almes and E. Lazowska, *The Behavior of Ethernet-Like Computer Communications Networks*

(2) R. Alonso, *The Design Of Load Balancing Systems For Local Area Network Based Distributions*, U.C. Berkeley Publication, Fall 1983

(3) R. Fateman, *DEC MICRO Research Proposal*, May 16, 1984, Department of Electrical Enginering and Computer Science, University of California, Berkeley.

(4) W. Joy et al., *Berkeley Pascal Users Manual Version 3.0*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

(5) S.J. Leffler, R.S. Fabry & W.N. Joy, 1983, *A 4.2BSD Interprocess Communication Primer*, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

(6) R.M. Metcalfe and D.R. Boggs, *Ethernet: Distributed Packet For Local Computer Networks*, Comm ACM 19, 7, July 1976, 395-404.

(7) S. Sechrest, *Tutorial Examples of Interprocess Communication In Berkeley UNIX 4.2 BSD*, Computer Science Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

(8) C. Williams and C. Guthrie, *A Proposal To Study Remote Processing of CPU Bound Jobs On Idle Computers*, Computer Systems Support Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1984