A Multivariable Information Scheme to Balance the Load in a Distributed System

Stefano Zatti

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Executing an incoming job on the least loaded of the machines of a distributed system can considerably improve its response time. To make the right choice, it is useful to have some information both about the load on the machines of the system and about the job we are considering. The information about the load ought to indicate the resources a machine is able to give, whereas the information about the job must tell which resources the job is going to need. We developed a multivariable scheme to distribute load information and to match a machine's available resources with a job's specific requirements. The experiments we performed with a prototype implementation show that our tool is able to make the right choice on a set of test jobs between 55 and 88% of the times. Our purpose is to shed light on some controversial issues, in order to prune the intricate complexity of the problem and open the way to future more general implementations.

May 12, 1985

This work was jointly sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089, by the Ministry of Public Education of Italy and by the HUSPI project. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of any of the sponsoring Institutions.

A Multivariable Information Scheme to Balance the Load in a Distributed System

Stefano Zatti

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

1. INTRODUCTION

Suppose you are waiting in a long line at an ice-cream parlor, and you are concerned about having the ice cream in your hands as soon as possible. Suddenly, a messenger with wings on his heels appears and tells you that just around the block another similar ice-cream parlor is much less crowded. You think you could get faster service over there, but you are also concerned about the time it would take to walk, and the loss of a good position in line (if you had to come back, it would be a disaster!). Maybe you don't realize it, but you are concerned with a problem of *load balancing in a distributed system.*

Due to the parallel evolution of microprocessor and network communication technologies [Smith1984a], today's computer systems have tended towards small units for individual use, with their resources connected together through a communication medium. The power of each unit may be small, but they can join their efforts for the execution of particular tasks. Moreover, significant savings in installation costs can be accomplished by *sharing* resources like printers, tape drives, special hardware devices; each one of them may be connected only to one machine, but be accessible through the network to anybody else. In this case, a simple system map would be enough to locate the resource and ask the machine in charge of it for service.

In a further evolutionary step we may want to share even the basic resources any computer is equipped with, like the power of its cpu or its secondary storage area, with several different purposes: fault tolerance, concurrency control, data synchronization, performance improvement [Wang1985a].

Performance improvements, in particular, can be obtained with respect to different objective functions: the most intuitive criterion is to try to maintain the *same number of processes* on all the machines accessible to the users, with the assumption that the machines' performances will be similar. But the migration of programs over the network has its own cost, and could lead to unexpected delays, defeating the purposes of the effort. Other objective functions, more related to what the user sees and suffers, are the *maximization of the system throughput* and the *minimization of the system average response time.* In our case, we considered a subset of the latter: we designed a load balancing scheme with the goal of *minimizing the response time* of a particular set of jobs, as seen by the user of one particular machine at his terminal; by providing a better *individual* response time, we also hope to improve the *global* system responsiveness.

This work was jointly sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089, by the Ministry of Public Education of Italy and by the HUSPI project. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of any of the sponsoring Institutions.

Two main components can be identified in any load balancing scheme: the *control law*, that determines when and where the offloading can take place, and the *information policy*, that selects and spreads information on the load of each machine, in order to devise an optimal choice (fig. 1).

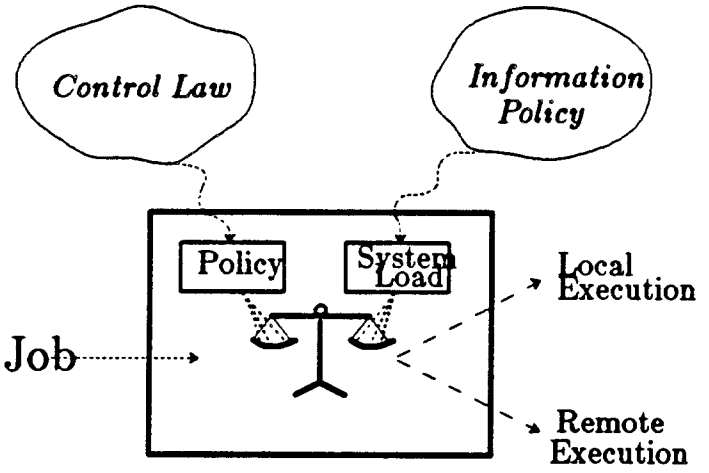


Figure 1: A Load Balancing Scheme

The design of the control law is the most crucial point of any load balancing scheme: if we send some load away, we may well increase the system's responsiveness by leveling the load, but we surely increase the utilizations of the remote machine and of the transmission medium.

1.1. Description of the Report

This report has the following structure: Chapter 2 describes some of the solutions devised by other authors in the past, and discusses some controversial points. Chapter 3 describes the scenario of our approach. Chapter 4 describes a set of experiments we performed in the first phase of the research in order to test the feasibility of the idea of sharing the load in a real environment, the Berkeley distributed system running under Berkeley UNIX [1] 4.2BSD. We were looking for some general load index from which we could get indications on the expected response time of every new job. We discovered instead several indices, each suitable for a different situation and a different type of job. This suggested the idea of a *multivariable* information scheme, where a separate index for each resource is maintained over the whole system. In the following three chapters the multivariable scheme is developed from the assumptions to the implementation: two alternative load schemes are introduced, one based on the real values of the indices selected to measure the load, and one on binary busy/idle indicators. In Chapter 9 the results of the experiments performed with a prototype implementation of the scheme are described: the multivariable information scheme and the binary choice are supported by the results.

1.2. Description of the Approach

When the goal is the minimization of the response time, we can think of a load balancing tool as a *job-level scheduler*[2] that decides, when a new job is submitted to a distributed system, whether or not the local load conditions would grant it a satisfactory response and, in case of a negative answer, chooses a remote machine and sends the job there for execution. The strategy is

---

[1] Unix is a trademark of AT&T Bell Laboratories

[2] In an interactive system, a user *job* usually corresponds to the unit that is executed as the result of the inputting of a command line. The execution of a job gives birth to one or more processes.

going to be successful provided that some unloaded machines do exist on the network. If all the machines are heavily loaded, the program should do as little as possible, to avoid performing useless work. We are not thinking, by now, of migrating processes already in execution, even if attractive schemes to do that have been proposed in the literature [Powell1983a, Danzig1984a]. These papers show clearly that the implementation of process migration is a major effort, that goes far beyond the scope of the present research.

The load balancing scheme we designed is characterized by the following attributes:

- *Dynamic*: the information about the system load is periodically collected and updated.
- *Deterministic*: the choice of the target machine is a unique function of the system load, with no randomness in the decision.
- *Nonpreemptive*: jobs that start running cannot be interrupted and moved to other machines (they are subject, of course, to the normal cpu scheduler of the machine they are running on).

We now define the general structure of our scheme, and introduce a terminology that will be consistently used in the sequel. For every job entering the system, the *submission site* is the machine connected to the terminal where the job is first submitted; the *execution site* is the machine where the job will be eventually executed. The job scheduler runs constantly, examines the jobs submitted to the system, selects some of them, and routes them to the current *best* execution site. If this site is a remote machine, the job must be transferred over the network.

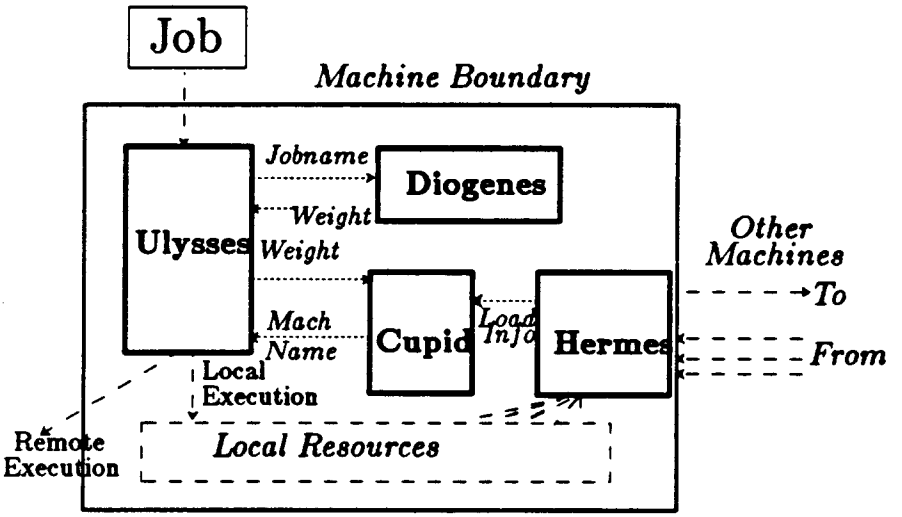


Figure 2: The Modules

The following four modules are running on each site (fig. 2):

1. *ULYSSES* (very clever and efficient), the job scheduler: picks the jobs and submits them to the best execution site.
2. *DIOGENES* (knows about the behavior of the programs), the job characterizer: maintains information about the requirements of the jobs.
3. *HERMES* (the messenger with wings on his heels), the system load information keeper: maintains current information about the load on the system, exchanging messages with the other messengers on the network.
4. *CUPID* (the couple-maker), the matching module: gets information from HERMES and DIOGENES, and reveals the best execution site to ULYSSES.

The modules will be described in detail in the following chapters. In the next, we briefly review some of the most relevant papers on load balancing that can be found in the literature.

2. RELATED WORK

The problem of sharing the load in a computer network has been extensively addressed in the literature. Very few of the available papers, though, describe actual implementations: they are mostly descriptions of theoretical algorithms or paper designs. Authors usually agree in presenting load balancing as a *job scheduling* problem, i.e., how to pair the jobs arriving at a distributed system and the machines in the system. Some of them even consider the possibility of stopping processes during their execution and moving them to other machines. There is less agreement on what the purposes of these schemes should be: some want to reduce the *average job response time*, some to increase the *global system throughput*, others to obtain a *uniform load* on all the processors.

The proposed schemes can be classified in many different ways (see e.g. [Zhou1983a, Wang1985a]). We will draw the first, coarse line between *static* and *dynamic* strategies. The static ones try to solve the problem once for all, by devising a *per node* routing rule to be applied to every incoming job. The rule can be deterministic (a job of type i will always be routed from machine A to machine B for execution), or probabilistic (a job of type i will be moved from machine A to machine B with probability p). Static policies typically ignore run time information about the current load of the system, and tend to devise a general criterion based on machine and job idiosyncrasies, or maybe on the past history. For example, [Chu1980a] solves a linear integer programming problem to determine the optimal allocation on the basis of some constraints depending on system parameters, while [Ma1982a] uses a *branch-and-bound* technique to solve an analogous problem, with the goal of obtaining balanced utilization of all the processors while minimizing the IPC cost. An alternative proposed by [Ni1981a] uses nonlinear programming techniques to build an optimal job scheduling matrix whose entries are probabilities for job class i to go to processor A. These schemes require nontrivial numerical computations to find the optimal solution[3], and do not ensure that the strategy will still be valid if any of the system parameters changes.

Other authors, concerned about the possible aging of the information decisions are based on, design *semi-dynamic* algorithms that periodically modify their strategies. These algorithms unfortunately can provide only sub-optimal solutions. [Ramakrishnan1983a] uses *time thresholding* to alternate execution among processors: after allocating a job to machine B, machine A refrains from sending more jobs to it for a certain *time threshold* t , whose optimal value is that which minimizes the response time in a queueing network model. [Stankovic1983a] and [Stankovic1985a] apply Bayesian decision theory to the problem of job allocation: all the possible states of the system are considered with their probabiltity distribution, and for each state the set of actions maximizing a utility function (in Stankovic's formulation of the problem, the expected response time) is computed. At run time, the allocation of a job is determined simply by a lookup of the table with the actions. Information about the load (which Stankovic defines as the number of jobs) is maintained on each processor. The overhead of remote execution is considered by introducing a *bias*, similar to our *delta threshold* (see *infra*): a remote machine's load is increased by the *bias* to keep into account the overhead of remote execution. The maximizing actions are recomputed at certain intervals (a simulation study suggested a value of 6 seconds for this interval), but we feel that such a frequent operation can be costly and influence appreciably the overall performance of the algorithm.

The dynamic schemes, many of which gave us inspiration for our design, try to make a decision based on the current status of the system, that is maintained and distributed at each node according to some information policy. The choice is generally deterministic, based uniquely on the current status. Dynamic schemes seem more feasible for real implementation. This is the

---

[3] *Branch-and-bound*, for example, is NP-Complete: if n tasks are allocated into m processors, the algorithm requires n^m complete allocations in the search tree.

case of UCLA's Locus Distributed System [Walker1983a]. Locus actually provides a user-initiated mechanism to execute a job on the least loaded machine, where the load is defined as the size of the run queue. Unfortunately, the designers did not develop any more general policy upon this mechanism. Also, Hwang *et al.* developed a Unix-based Local Computer Network, with a load balancing mechanism called *rxe*[Hwang1982a]. Only some of the commands can be fed to *rxe*, and the user must select them explicitly. They will be executed on the least loaded of the machines: the load index is a sort of "stretch factor", i.e., the ratio between the execution time of a standard job on a fixed machine with fixed load conditions (in their case, it was an empty PDP-70) and on the machine being considered. Neither one of these two papers mentions any results about the performance of their load balancing schemes. The possibility of balancing the load does not seem the main concern of the designers, but just a marginal issue.

A similar information policy is used by [Barak1984a] in the scheme developed at the Hebrew University of Jerusalem for the MOS operating system. The load index is the ratio between the number of time quanta requested by the processes and the number of quanta that the system was able to grant them during a certain time interval. This information is periodically refreshed by an algorithm that keeps information only about *some* of the machines, choosing them in a random way. A process runs on each processor, supervising the other processes and asking some of them to migrate out if they have been running for a while. The controller estimates each process's remaining execution time, and adds to it a *stability value* to avoid processor thrashing. The mechanism for process migration is not described in the paper.

A clever mechanism is the one devised by [Krueger1984a] to keep the load of all machines on the network as close as possible (i.e., within an acceptable load range) to a presumed *average load*. A machine that finds its load to be below the average load broadcasts a bid to accept incoming load, whereas a too loaded one broadcasts a request for bids. Bids and requests pair up and build a connection, and some processes are selected and moved through that connection. The criterion to choose the processes to suspend and move to other machines is suggested in a previous paper from the same group of researchers [Bryant1981a]: the authors prove that jobs that have been running for some time are likely to run as long in the future. Thus identified, *big* jobs are suspended and moved to less loaded machines.

A recent work in the field is [Eager1984a], which introduces and compares three different control laws: *random placement*, *threshold* (canvass a number of machines, and send the job for execution to the first found whose load is below a certain threshold), and *shortest* (maintain information about the load of a number of machines, and choose the least loaded one). *Load* is here the number of users in the run queue. The schemes are all evaluated by means of a queueing model, together with the worst case ($M/M/1$) and the ideal case ($M/M/K$). The results show that random placement gives a dramatic improvement in the response time with respect to $M/M/1$, whereas the other two schemes do a little better, but are not definitely worth the cost of getting the information they need. Chivalrously, the authors concede "the benefit of the doubt" to the policies that maintain and use load information.

Among the wide breadth papers which introduce several algorithms and compare them, both [Livny1982a] and [Wang1985a] deserve mention. In the former, the authors prove that there is a high probability that, if the arrival rates at the nodes of a computer network are the same, then there exist some nodes that are overloaded (queue length > 1) while some other nodes are very lightly loaded. This motivates the idea of moving the execution to remote machines. Three information policies are introduced: *State Broadcast* (the state of each machine is periodically updated and broadcast to the others), *Broadcast when Idle* (a node that becomes idle sends a bid to all the others) and *Poll when Idle* (a node that becomes idle asks some of the others for jobs). The three schemes are evaluated through simulation: their relative performances turn out to depend strongly on system parameters, and the authors do not express a definite opinion about any one of them. A very systematic approach inspires the paper by Wang and Morris, where 14 schemes are introduced, ordered by their needs of load information (from no knowledge to future knowledge), and subdivided in server- and source- initiative. The schemes are compared by means of a general performance index called *Q-factor*, that tells how close the system's behavior is to

that of a multiserver FCFS station ($M/M/K$), as seen by every job stream. The Q-factors are evaluated with several different methods (both queuing models and simulation): the main result is that *server initiative* generally outperforms *source initiative*, and that some less popular algorithms like *random service* or *cyclic service* can in some circumstances provide effective load sharing at low communication costs. Wang's taxonomy is not really comprehensive: all the load indices he considers concern only one resource, the cpu. No other resources that could be responsible for delays in the response time are considered in any of the 14 schemes described in the paper. A multivariable scheme like our own, for example, does not fit in this taxonomy.

3. PROCESSING SCENARIO

The computing environment to which our considerations apply includes:

- A set of multiple user hosts with:
  - Different CPU powers.
  - Different mass storage facilities (number and speed of disk drives).
  - Different memory sizes.
  - Connection to the same local area network, that provides broadcasts.
- The files containing the object code of the programs we want to execute are replicated everywhere. There is, therefore, no need to move them to the execution site. This assumption allows some heterogeneity to exist among the machines: a machine's executable code resides on it, without any need for a translation. If the program needs one or more input files, though, these files have to be moved and maybe even translated into the target machine's data representation code. The assignment of a job to a host is accomplished simply by shipping the command line, and possibly the input files.

The network might also contain (note that this would require an extension to the design):

- Single user workstations.
- Machines with special-purpose hardware, like array processors, FFT processors, floating point processors, or special printers. These resources may be present in multiple quantities on the network, thus requiring the choice of a particular one of them.
- Machines dedicated to specific services, like file servers, name servers, and mail servers.

We call the set of machines that could possibly receive the job we want to offload *the set of eligible machines*. This set is constantly changing on the basis of the load information (see *infra*).

Besides the load, however, other external factors like ownership and autonomy could determine a reshaping of the set: the owner may want to restrict the access to her facilities. For example:

- The owner is willing to allow only N% of her resources to be used for foreign jobs, and, when this limit is reached, she claims the right to refuse further load. An automatic mechanism must be provided to guarantee that this will be possible.
- A workstation's owner wants the machine *all* for herself when she is working on it, but is willing to yield it when she is not using it.
- The owner wants to be able to "close the faucet" by denying access to her system, or maybe only to a single type of resource. In this case no job at all can make use of that resource or that machine, until she changes explicitly her mind.

These considerations require a definite choice about the "politeness" of our policy: due to the particular timing of messages over the network, a machine could still get incoming jobs after its availability has been denied. Where should the authority to decide the execution site reside? In transient situations, the transmission of a high load index value to state unavailability could not be enough to avoid an unpleasant convergence of jobs into a machine that used to be unloaded.

For example, a machine could be quickly jammed by incoming jobs, before the change in the value of its load index can reflect the new situation and induce the load balancer to stop the flow. In this case, the target machine's Ulysses could resend the extra jobs out, incrementing a hop-counter to avoid an endless bouncing of the same job from machine to machine. Alternatively, the chosen machine could be impolitely compelled to execute everything it receives, with no possibility of refusal. This choice should be made dependent on the environment we are considering, the ownership of the machines, and the existence of a general authority that can impose decisions.

We will henceforth assume that our environment is made up of machines totally willing to cooperate: if the copy of Ulysses running on one of them still sees jobs arriving after access was denied, it will either process them or send them out again under its own responsibility. The copy of Ulysses running at the submission site will soon receive the new value of the corresponding load index, and refrain from sending there jobs to execute.
Possibly, an owner could even refuse access to someone else's jobs, or to some particular job classes. I will not consider this here: our load balancing should not enforce any class discrimination. Finally, being the environment so collaborative, no malicious misuse of the mechanisms will be considered.

4. THE FIRST PHASE OF THE EXPERIMENTS

4.1. The Feasibility Study

When first approaching the problem, in the Spring of 1984, we did not have any experience with remote execution, that could convince us that distributing the load was a good idea: the documented implementations we knew about [Hwang1982a, Walker1983a] did not provide very conclusive data. We designed and ran a first set of experiments to test the *feasibility* of remote execution, and to analyze some load indices on which a decision about remote allocation could rely. We started by addressing the following issues:

- The nature of a system's load, and its characterization by means of a *load index*. Several metrics had been proposed [Alonso1983a, Cabrera1985a], but they had not been related with measurements to a program's response time.
- The amount of the improvements in job response time we can get by remote execution, and the price we have to pay for network communication ([Cabrera1984a] evaluates in detail the TCP/IP network protocols) and control of the process on the remote machine.
- The characterization of the *job classes* that would benefit from remote execution and the ones that could always be executed locally.

A rather primitive tool, called *dsh*, was available under Berkeley Unix [Presotto1983a]. *dsh* (distributed shell) allows remote execution of a manually selected job on the currently *least loaded* machine, using Unix's *load average* as the load index. Load average is actually the length of the queue of runnable processes waiting for the cpu, exponentially smoothed over an interval of one minute; clearly, it is a metric purely related to the cpu load.

For our experiments we selected some machines and generated artificial *background loads* on them by means of shell scripts. Then we ran repeatedly some standard *probe jobs*, both with and without *dsh*. The response times can be easily measured with the C-shell *time* command[CSHa]. Two typical results of the experiments are shown in figure 3 and figure 4. They are described in [Zatti1984a]. The dotted lines have been added only for pictorial clarity.

Figure 3
Response Time of an I/O Bound Job

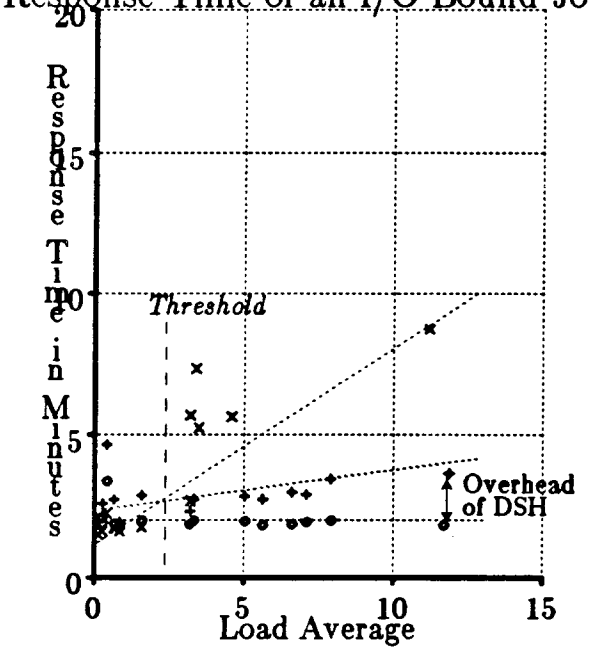
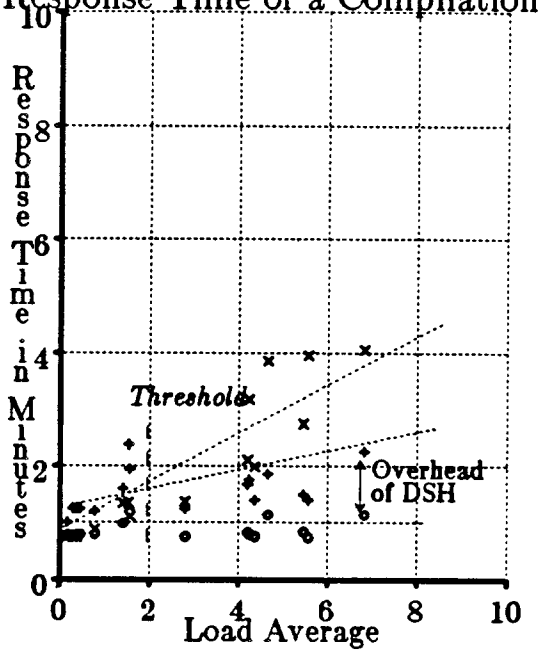


Figure 4
Response Time of a Compilation



Legend

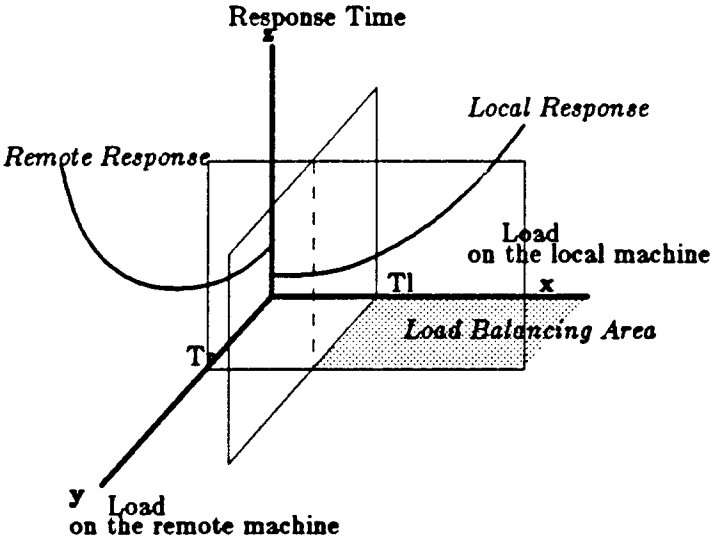
× local response time
\+ response time with DSH
o remote execution time

We observe that the *local* response time is better than the remote one, as long as the load remains below a certain *local load threshold*. But as soon as the load passes this threshold, we get noticeable improvements if we can find, somewhere in the network, a *sufficiently unloaded* machine. Note that, to consider a remote machine sufficiently unloaded, the value of its load average must be situated below a second load threshold, the *remote load threshold*, that is smaller than the local one. Due to the overhead of *dsh* itself, in fact, a remote execution is penalized, and can improve the response only under very particular load conditions.

The overhead of *dsh* could be evaluated by measuring the response time *on the remote site* and subtracting this value from the total response time: we found the value of this overhead to be very high (16 seconds on the average). In fact, *dsh* collects the loads of the remote machines by creating a TCP/IP connection [Postel1981a] in turn with every one of them. TCP/IP provides reliable communication by retransmitting for up to 10 times a message that has not been acknowledged. The round-trip time of a message is severely affected by the reliable transmission mechanism, and by the load of the sending and receiving machines, as shown by [Cabrera1984a, Cabrera1985b]. The delay to gather the information about the remote machines is entirely suffered by the user, waiting in front of his terminal for the response. A faster, nonreliable remote procedure call mechanism based on UDP/IP [Postel1980a] would be much more suited to this probing strategy, but still any delay incurred by the mechanism would be suffered by the user. Moreover, the probing strategy requires messages to be sent every time a new job comes under consideration: this does not scale well with the number of jobs coming into the system. We prefer the idea of maintaining information about the load with an asynchronous mechanism (that we called *Hermes*), and getting the data with a fast *local* inquiry every time it is needed.

However, the main conceptual result of the experiments was the definition within the load space of a *load balancing area* where we are likely to get some benefit by remote execution (fig. 5).

Figure 5
The Two Load Thresholds



The area is delimited (the picture is restricted to a two-dimensional local-remote load space for clarity) by two thresholds: The *Local Threshold* , beyond which we can expect better results from a remote execution, and the *Remote Threshold*, lower than the local one, that limits the benefits achievable on a remote machine (there is one of each for each machine, of course). We call *Delta Threshold* the difference between these two thresholds: this is an important parameter for the binary information scheme we will define in Chapter 5.

4.2. The Load Metric

In the second set of experiments we investigated the relationship between the probe job's response time and several load indices we selected to express the system's load. Our goal was to identify *one* load index suitable to *all* jobs and *all* load conditions: the performance of load average in the first experiment was not very convincing. Of the indices we tried, instead, no one showed the *generality* we were looking for: we decided to look at different resource-related indices depending on the type of job that we are considering.

In these experiments, on a given machine a background load was generated by running several instances of the same two kinds of very heavy programs, a cpu bound and an i/o bound one, every time with different mixtures of the two. A probe was run once for each different load value, and its response time was plotted versus different resource-oriented metrics. Some of the results are shown in figures 6-10: every point refers to a single observation.

Figure 6
Cpu Bound Load-Cpu Bound Probe

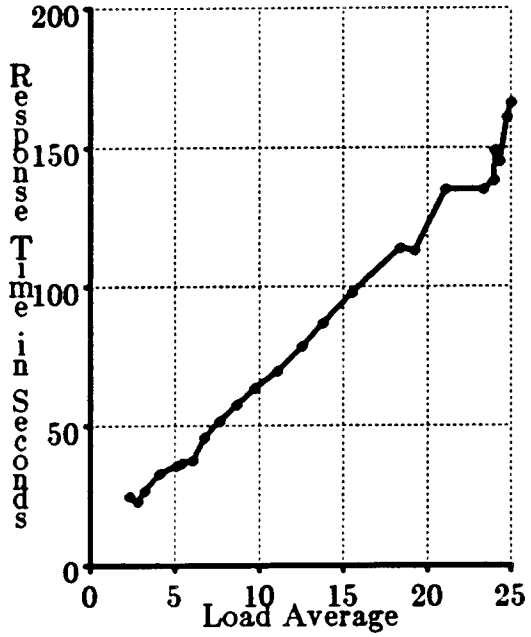
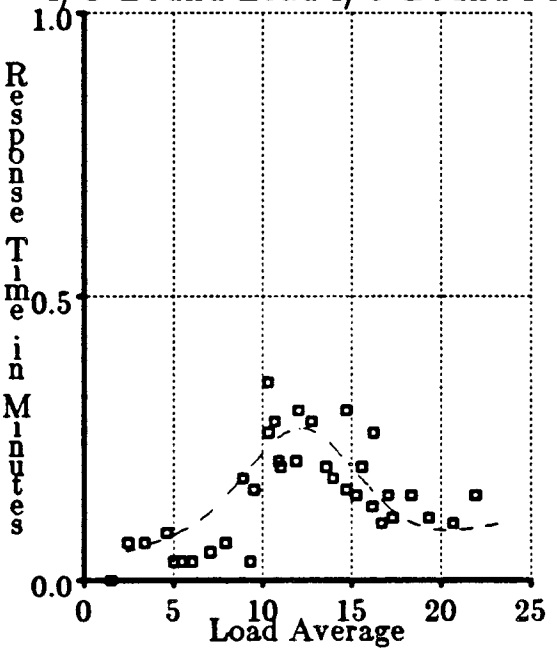


Figure 7
I/O Bound Load-I/O Bound Probe



As intuitively expected, if we run a cpu bound job with a pure cpu bound background load the response time increases roughly linearly (see fig. 6); the same is true for i/o bound load and i/o bound probes, even if we use the load average as our index (not shown). More intricate cases show stranger behaviors: The cpu bound job's response time plotted in figure 7 does not increase monotonically with the load average, but starts decreasing beyond a certain value: the scheduler reduces the background processes' priorities after they have been running for a while, making it easier for the probes to get executed quickly. According to the requirements expressed in [Ferrari1985a], we want a curve of a job's response time that is *single valued, monotonically increasing*, and *linear*. The load average is not a good metric in the case of figure 7.

If we have an i/o-cpu balanced background load, we still have an almost linear curve for the cpu bound probe (see fig. 8), but an irregular one for the i/o bound, that flattens (i.e., does *not* get worse) after a while.

Figure 8
Mixed I/O-Cpu bound load

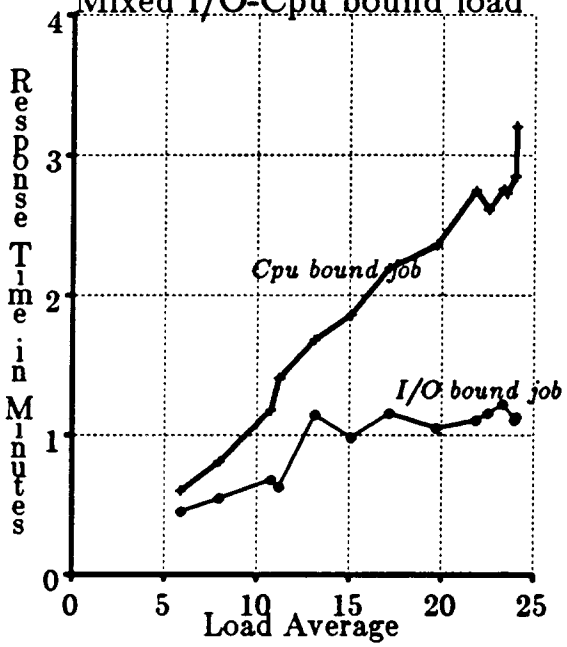
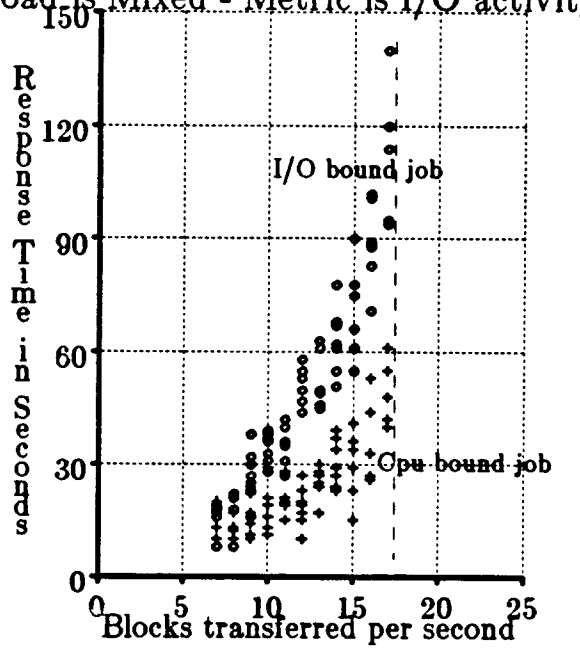


Figure 9
Load is Mixed - Metric is I/O activity



One possible reason for this behavior is that an i/o bound job makes use of the cpu anyway, but keeps cycling between it and the i/o without wholly using the time slice the scheduler grants it; therefore it does not see its priority decreased by the multi-level feedback-like mechanism used by the scheduler [Peterson1983a].

The crucial observation here, though, is that the load average does not show a good correlation, under general load conditions, with the response time of typically i/o bound jobs.

Consequently, we tried some other metrics, related to other resources, and we got results we found interesting. Figure 9 shows the response time of two typical i/o (symbol: "o") and cpu bound (symbol: "+") jobs as a function of the system's i/o traffic. The machine was a VAX 750 with 2 disks, and the background load was balanced. The i/o system has a maximum throughput, that we can observe to be about 18 blocks/s (on both the disk drivers altogether, assuming the traffic to them is evenly split). When the system is under full load, the response time for the i/o bound job grows very fast, whereas the cpu bound job's response, increases more slowly. An i/o bound job is *extremely sensitive* to the load of a machine's i/o system.

Figure 12 shows another result about the i/o index: the i/o traffic is plotted versus the load average, and no correlation can be seen between the two. This experiment was performed on another machine, with more disks and therefore a larger total disk i/o bandwidth: the i/o system works at full rate even with a small number of processes in the cpu queue, almost independently of their number.

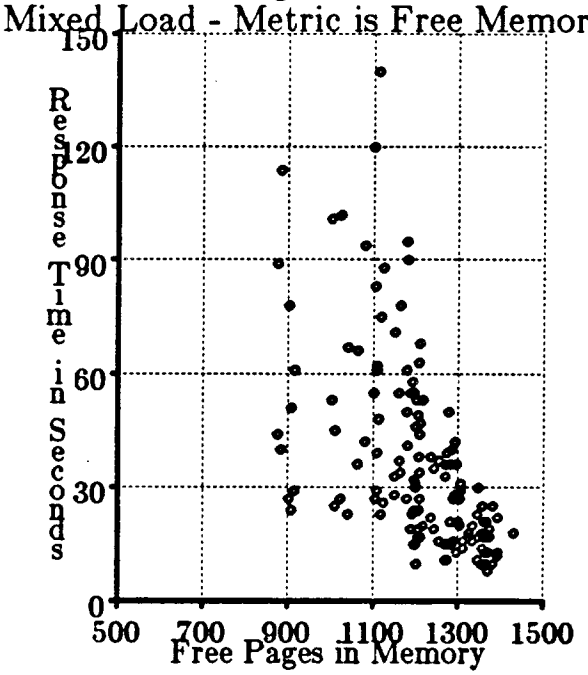
Hence, we can relate the behavior of an i/o bound job to the i/o traffic, but we must make sure that the i/o traffic is less than the maximum throughput that the machine's i/o system can provide, if we do not want the response time to degrade too noticeably. The maximum throughput is an *i/o threshold*, analogous to the cpu threshold we defined earlier. This consideration led us to the definition of the binary load scheme that will be introduced in Chapter 5, Section 2.

Exploring the behavior of the virtual memory, we obtained a good correlation [4], respectively, between the free list size (fig. 10) and the active list size (analogous, but growing

[4] The correlation coefficients are, respectively, -0.6 and 0.7; they are lowered by the high variance of the response time for low (high) values in the free list size (active list size).

with the size of the list), and the response time. First, a higher response time corresponds to smaller free memory size: with little free memory it is more difficult for the virtual memory manager to allocate the job's working set. Second, and somehow complementary, more memory occupied causes delays to further allocations. Notice that the graph does not show any pathological situations, such as when the system is thrashing: this would happen for a free list size smaller than 500, but such a phenomenon was not observed during our experiments (the swapping rate was monitored, too, and was found quite consistently to be around a value of 15).

Figure 10
Mixed Load - Metric is Free Memory



The main phenomenon we can observe here is the clustering of most of the responses, respectively, in the big-free-list and small-active-list areas. This confirms the well-known conclusion that the response time is generally good if there is enough memory available.

To bind this result to the number of processes, we ran some tests trying to correlate the size of the active list to the load average. Figure 11 shows the results; as expected, the number of active pages in memory is correlated (with correlation coefficient 0.74) to the number of processes in the cpu queue.

Figure 11
La/Used Memory Correlation

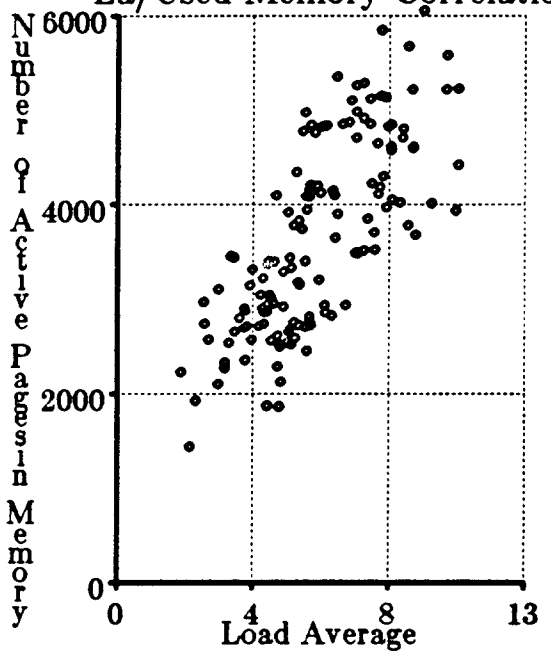
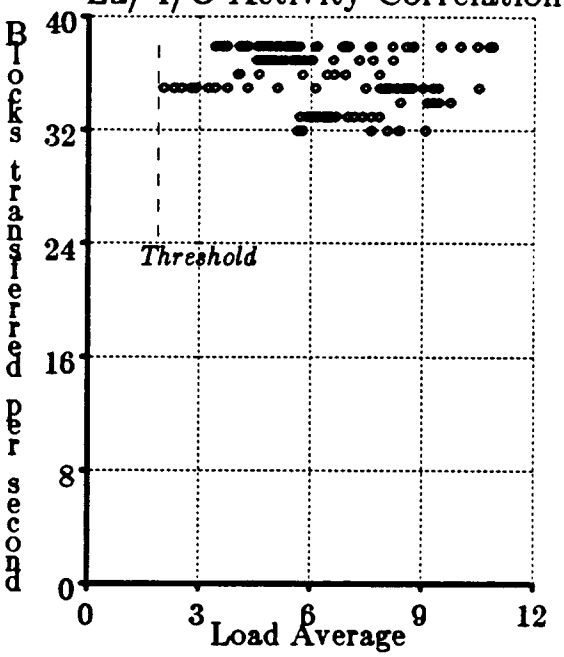


Figure 12
La/ I/O Activity Correlation



In conclusion, no one of our experiments revealed a load index suitable for all circumstances: as far as we can tell from this set of experiments, that analyzed, however, only a few of the possible load indices, the best load index depends rather on the *type of job* we are considering. We were led to a general information scheme: we characterize each resource with a scalar value, and build a *load vector* with these values; the load vector is the index that Hermeses on all the machines exchange with each other to make their situation known. Another parallel project at Berkeley [Ferrari1985a] has suggested resource-oriented load indices to be used alternatively in different occasions, depending on the type of job being considered. Analogously, our approach keeps *all* the load information in the vector, but matches it with the job's requirements in a uniform way, independent of the job type, using *all* the fields available.

The experiments suggested also that some of the resources behave in a binary way: they provide a generally good service as long as they are not saturated. We will characterize these resources with a binary value (resource saturated/non saturated) and compare the performance of this method with that of the real-valued one (i.e., that using the actual value of the load index).

5. HERMES: MAINTAINING SYSTEM LOAD INFORMATION

A computer system is a composite device, made up of separately available resources. Some of these, like cpu, memory, i/o system, are common to all computers, while others (tapes, printers, or floating point processors) are peculiar only to some of them. The *requirement* for a certain amount of a specific resource is different for every runnable job. The *availability* of a specific resource at a specific site must be advertised, in order to allow a redistribution of the processes as tailored as possible to the needs of each process. We propose a *multivariable* scheme to distribute information about resource availability.

The load on a system is expressed by a vector of n fields, each of which refers to the availability of a particular resource. We will refer to it as the *a.vector*.

The values of each field of this vector can be either binary or real. Sometimes the nature of the field requires a binary value, as in the case of a special device, about which we just need to know whether it is there or not. Other times, an integer value is necessary, as in the case of a hop count. In yet other cases, the value of the field can be either binary or real. We consider both the alternatives for the fields related to cpu and i/o activities, trying to determine which is better.

The mechanism for updating the information, common to both approaches, is described in the next section. The nature of the information, the data structures used to store it, and the use that the program makes of it by matching it with the requirements of the incoming processes, are different; they are dealt with in the following sections.

5.1. Updating the Information

The information about the resources we selected should be kept constantly up to date, but this requires frequent measurements that generate load on the system. That information must also be transmitted to the other machines, and this generates further load, both on the system and on the network. Therefore, the fresher the information is, the higher the load on the transmission medium: this tradeoff is commonly referred to as the *transmission dilemma* [Livny1984a]. We try to solve the dilemma by refreshing the information only at periodical intervals, of such length that the information should not change appreciably during them.

A Hermes process runs constantly on each machine, performing the following operations:

- It reads the load indices from the kernel with interval t_1 (the *update interval*), and smoothes the values with an exponential combination of the old one, to avoid sudden jumps.
- It broadcasts this information to all the other Hermeses only when some of the fields "change." The meaning of "change" depends on the approach (see *infra*)[5].
- It receives from the other machines' Hermeses information about the remote loads, and builds a structure to keep this information.

This sequence of messages allows:

- The dynamic reconfiguration of the set of the eligible machines (a new one comes up, another one denies access,...).
- Periodic checkpoints for correctness of current information.
- The detection of crashes, or even of very stable conditions: Hermes can determine whether during a certain interval a machine did not send any message.

In both the real and binary vectors, we have integer (and not binary) values for the hop count and for memory availability, which is expressed by number of pages in the free list. A binary characterization of memory (thrashing/not thrashing) seems too rigid and is not motivated by any of our previous results.

5.2. The Binary Metric

The a.vector has binary components, one for each resource, whose meaning is:

0 = Saturated

1 = Not saturated.

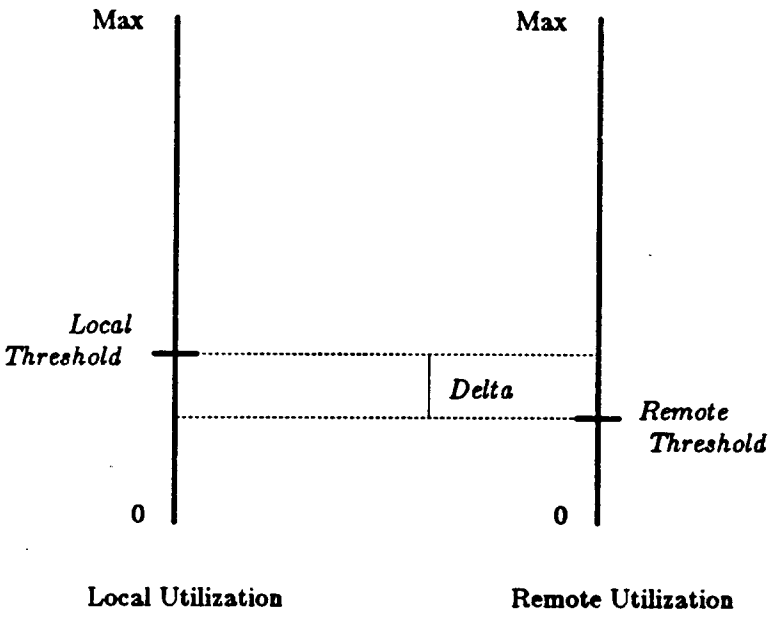
The availability is here defined with respect to a *load threshold*: each time the load threshold is crossed (what we referred to as "change" in the previous section), the bit toggles, and the a.vector is sent out to the other machines with its updated value. The load threshold is determined experimentally on the basis of the following factors:

- The *utilization* of the resource: when the threshold is passed, the resource is so busy that it cannot ensure a satisfactory service to new clients.
- The *relative power* of the resource, with respect to the corresponding ones in the network (i.e. cpu power (MIPS), speed and number of the disk drives).
- The *distance* of the resource: sending a job through the network to be executed remotely costs in terms of response time. The remote load threshold (the non-availability limit) must

[5] The information broadcast could be just the *difference* from the previous one: this could avoid transmission of redundant information, but would make the system more vulnerable to message losses: if one update is lost, the information about that machine will be wrong, and, what is worse, will remain wrong until a brand-new a.vector will arrive. We will not follow this alternative method.

be smaller than the local load threshold (the limit above which balancing is performed). The difference between the two thresholds (*Delta Threshold*) represents the cost of remote execution (see figure 13): the job would take approximately the same time to execute on the local machine with a load equal to the local threshold as on the remote machine with load equal to the remote threshold.

Figure 13: The Delta Threshold



The following pseudocode shows how the information algorithm works:

```
while(running) do {
     for each field in a.vector {
          read the load index;
          if (value > threshold) bit = 0;
          else bit = 1;
     }
     if (a.vector != a.previous) {
          broadcast new a.vector;
          a.previous = a.vector;
     }
     wait for a time t1;
}
```

Obviously, this approach is the only reasonable one when the field refers to the availability *tout court* of a specific hardware resource, like a laser printer, on a given machine, but in this case we do not need to refresh the information unless the resource is physically removed from the network.

5.3. The Real Valued Metric

The a.vector is made up of real components, one for each resource. Each one of these values is *normalized* into a scale that is the same for all processors, so that it allows us direct comparison of values from different machines. In order to perform the normalization, we could run the same probe job, that uses heavily the resource we are considering, on two different machines with the same load conditions (ideally: no load) and we compute the ratio between the execution times. We use this value as the normalization factor for the resource and for the two machines. For the real value metric, the degree of availability is simply stated by the value of the

index for the resource that is being considered.

As in the previous case, the information about a machine is sent out with an interval t_1. But, since it is represented by a real number, this value is very likely to be too sensitive to any small changes. The new value just obtained from the system is therefore smoothed by the previous value with parameter *alpha*, with an algorithm similar to that used by the load average command. Also, if we want to save on the side of the network traffic we can define a per-machine *sensitivity threshold* for a resource, as the range within which we consider a value "unchanged" since the previous time it was observed. The algorithm would be:

```
while(running) do {
     changed = 0;
     for each field in a.vector{
          measure resource's metric value into current;
          if |current - previous| > sensitivity threshold{
               previous = current;
               changed = 1;
          }
     }
     if (changed = 1) broadcast new info;
     wait for a time t1;
}
```

The binary and real metrics are sometimes alternative for specific resources, especially for the cpu. Other resources could be well characterized by a binary value. Recall the experiment described in Chapter 4 (see fig. 9 and 12): as soon as the bandwidth of the disks is saturated, and this may happen even if the cpu load is low, the i/o system causes long waiting times to those jobs that make some use of it. A simple saturated/not saturated bit is enough to keep the typical victims, i.e., the i/o intensive jobs, off that machine. As a general rule, we want the job to stay away from a resource that is too busy to provide it a satisfactory service (i.e., we want to avoid bottlenecks).

5.4. The Fields of the A.Vector

The choice of the fields in the a.vector was inspired by the experiments we described in Chapter 4: we observed there that, in different situations, different resources have critical importance in the allocation decision. We orient our choice to those resources: cpu, i/o and memory availabilities are declared in every machine's a.vector. Special hardware devices need a field too, to indicate their presence. A field is allocated to the hop count: after a certain number of hops, the job will be executed locally. In the case of a workstation, the cpu field would act as a busy/idle indicator: a workstation, we said, is available only if nobody is working on it.

A resource-oriented metric seems more suitable than a functional or logical one because it is a physical resource that might turn out to be a bottleneck, while other resources in the same machine could be idling. Relieving the physical bottleneck of some of its load could produce a considerable increase in the overall performance of the system. The resources, obviously, are not independent from each other, but do work separately; on this basis we consider their load values separately.

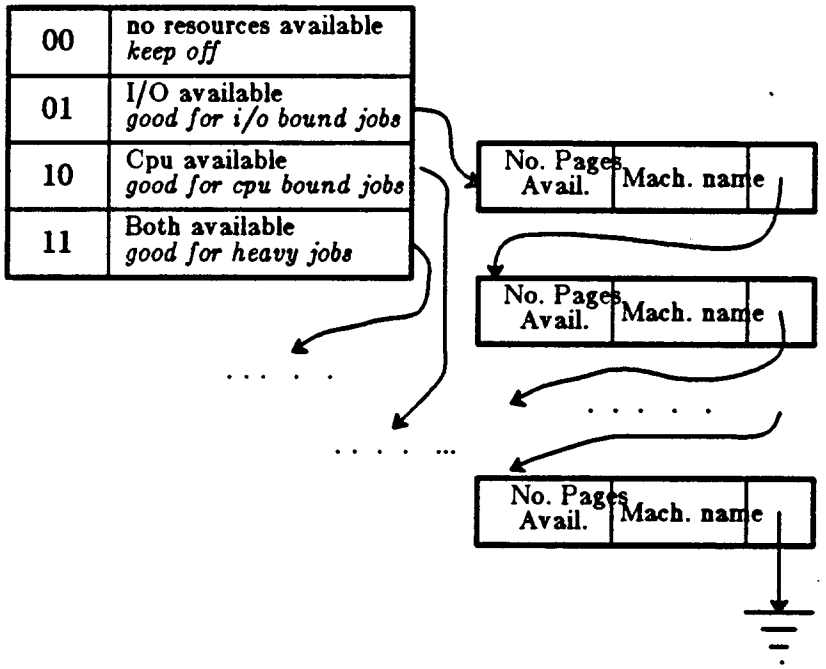
The choice of the best execution site for a newly submitted program is performed by Cupid (described in Chapter 7), which checks the load information maintained on the submission site by Hermes, and the information (if any) that Diogenes can collect about the requirements of the job itself. Cupid must be extremely quick in his decision, since the time it takes will be entirely suffered by the user waiting for response at the terminal (recall that our goal is giving the user a better response time). The price for an efficient decision has to be paid by Hermes, which must organize its data to optimize the access time. We call this data maintained by Hermes the *a.matrix*, composed by a.vectors.

In this context the *data structure* we use for the a.matrix is crucial: we would like to find the best execution site with a minimum number of searches, and also to gain access only to that subset of data we are going to need. Cupid uses some of the information about the load itself as an access key to the a.matrix.

5.4.1. Binary Information

We organize the a.matrix in a *hashing table* with 4 slots 0, 1, 2, 3, accessed through a function of the cpu and i/o bits, considered as an integer key. If we have other bits in an a.vector, representing other resources' availabilities, they could be embedded in the hashing function as well. The memory field and all the other various fields of the a.vector are stored in a linked list sorted in ascending order of the memory fields. The table is shown in the Figure 14.

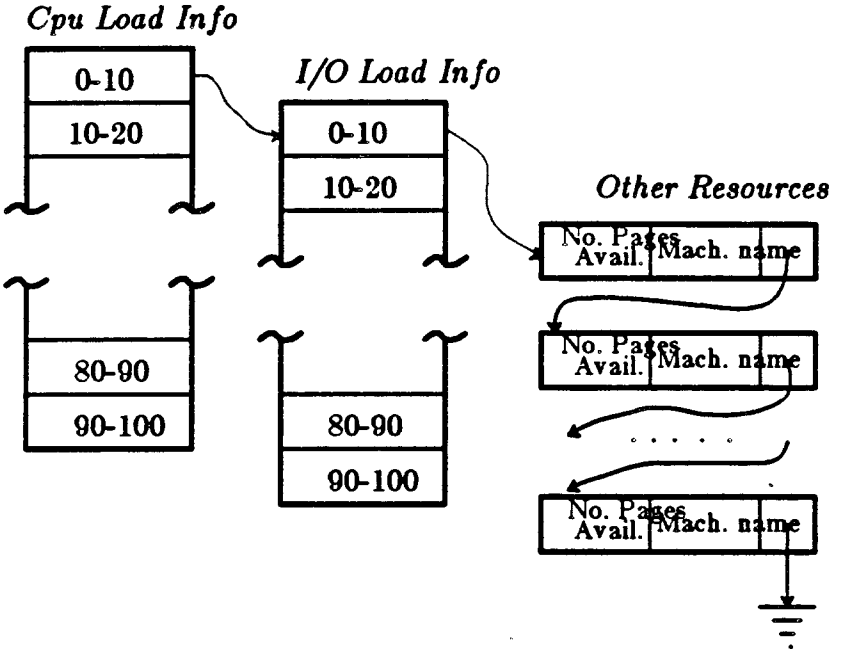
Figure 14: The Binary Information



5.4.2. Real Information

The real information is much more difficult to organize, since we would like to gain access to it with multiple keys (e.g., i/o field first for i/o bound jobs, and so on). We would need a database-like structure, but we do not want to suffer any query processing delay. In analogy with the binary scheme, therefore, we divide the domain of the cpu load metric into n finite intervals, thus creating an n-slot ordered table. For example, if the range is 0-100 we divide it into 10 intervals, each of which corresponds to a field of a table with 10 entries. The values of the i/o field are divided into intervals in ascending order as well, constituting analogous tables pointed to by each slot of the cpu table. We get a multiple-level index structure ordered in a somehow "lexicographic" way with respect to each one of the fields. When an a.vector arrives, Hermes breaks it and fills the slots in the tables corresponding to the values of the indices. The memory field and the machine name are then inserted in a linked list appended to each i/o table slot, according to the ascending order of the memory field, as suggested by figure 15.

Figure 15: The Real Information



This way of organizing the a.matrix allows almost random access, that is faster than a sequential search, to the cpu and i/o fields. Cupid has eventually to search the linked list until a complete match is found, but, given the sparseness of the a.matrix, we can expect this list to be very short.

6. DIOGENES: MAINTAINING PROCESS LOAD INFORMATION

Once we have all the information about the system load, we want to be able to use it for the choice, at the submission site, of the execution site for a new job. There are some commands (probably the majority) that are so light that they do not deserve any effort: they should be simply executed locally. The first selection should identify the class of jobs that could benefit from remote execution, and make their names known to Ulysses. To achieve a better allocation, it would be helpful to collect some more information about these jobs and their resource requirements. A complete workload characterization is a major effort, and requires:

- A thorough analysis of the jobs submitted daily to a system.
- The selection of those heavy enough to deserve consideration.
- The grouping of these jobs into classes, if necessary to reduce the size of their population.
- The definition of a single *requirement vector* for each class (henceforth to be called the *r.vector*).
- The characterization of each selected job with its class' r.vector.

A new job should be assigned an r.vector on the basis of its behavior (how much of each resource it used in its first run) or on the basis of hints contained in the command line. Only some of the job classes would benefit (or suffer...) from the load balancing policies. Diogenes tells Ulysses which jobs should be considered for remote execution, and which ones should always be executed locally, by looking at their r.vectors. The r.vector belongs naturally in a job's executable file(s), as a piece of information that each of them should carry with itself.

In this study we will not perform a full scale workload characterization that, however interesting, is beyond our present scope. However, we need to characterize some test jobs for our

next experiments: rather arbitrarily, we assign an r.vector to a job according to the following rules:

- In the real vector case, we insert in the r.vector the cpu utilization of the job, obtained dividing the cpu time actually used by the elapsed time, or the number of processes in the run queue, and the i/o rate, that we can obtain by dividing the total number of i/o operations performed by the elapsed time.
- In the binary vector case, we put a 1 in the cpu-field and a 0 in the i/o field (i.e., access key 10: 2) if the job is considered cpu bound, a 1 in the i/o-field and a 0 in the cpu field (i.e., access key 01: 1) for an i/o-bound job, a 1 in both fields (i.e., access key 11: 3) if the job acts heavily on both.
- The memory field contains in all cases the real memory (resident set) size of the process in 1024 byte units, since a binary characterization of the memory needs does not seem to make much sense in this context. In Unix, this information can be obtained by reading the RSS field of the output of the *ps* command.

7. CUPID: THE MATCHING MODULE

We have an a.matrix, with information about system-wide availabilities, and an r.vector. We must now design a vectorial function from the *(systemload X processload)* space into the system space, that returns to Ulysses the name of the machine expected to yield an optimal execution time to the command under consideration:

$$f(a.vector, r.vector) = machine\ name$$

Cupid must be extremely fast, since its response time is totally perceived by the user. Its action depends on the kind of vector (binary vs. real) we are using. Separate functions, therefore, have to be designed in relation with the approach we want to test. The *distance* to a possible execution site is crucial: whenever feasible, local execution is the best choice, and then the execution on machines on the same network. We could even move out to other networks, if we had for example a pool of workstations with their own independent connection, but we will not go that far in the present implementation. Cupid will check first of all if the local load is good enough to allow a satisfactory local execution. In case we allow the remote machine to ship the job further, the hop count is kept in the r.vector and checked to determine whether or not the job can still be sent away. After this preliminary exam, Cupid looks into the data structure maintained by Hermes to find the optimal matching. The operations differ depending on the type of vector (binary vs. real).

7.1. Binary Matching

In the case of the binary approach, we want to match a 1 in the r.vector (expressing need for a resource) with a 1 in the a.vector (expressing availability of the resource for that machine). The matching function implemented by Cupid uses the bits in the r.vector as an access key to the load table, then scans the linked list according to the free memory field to find the first fit (note that in this case it is also a best fit), and returns the name of the machine. (We do not consider here other possible fields for special devices, that would have to be checked as well, by embedding them in the bit pattern of the access function.) Then Ulysses sends the job to the named machine for execution, waiting for an acknowledgement to make sure the machine has not crashed in the meantime. The copy of Ulysses running on the target machine will reconsider the job, deciding whether to execute it (as presumably will happen), or ship it out again (if the situation of the load changed in the meantime). The hop count is increased at each hop, and, when a maximum value is reached, the execution takes place anyway (at that point the user will be in a rage anyhow). After Ulysses has made a decision about the process, the local Hermes changes *its view* of the a.vector of the chosen machine, by subtracting the fields of the job's r.vector from the corresponding fields of the a.vector, thus avoiding any further allocation to that machine until a new a.value is received from it. (In this case, Hermes simply resets the bits of the binary fields, and diminishes the free memory size by the allocated process's working set size.) This mechanism

mimics the *time thresholding* scheme mentioned above [Ramakrishnan1983a], with the interval between two consecutive load index upgrades as the value of the time threshold. Note once again that the availability of the local machine will be always checked first, to try local execution. Local execution is inevitable either in case the hop count in the process' r.vector has reached the maximum allowed, or in case no available machine has been found on the network (*null* returned by Cupid).

7.2. Real Matching

In the real-indices case, the load table is accessed using the cpu field as a key. The i/o field is then directly accessed in the table corresponding to the cpu field we already matched, and then the first fitting memory field is checked. The first complete match will point to a machine name returned by Cupid as the "optimal" execution site. We may need to backtrack while scanning the tables, but we are always sure that *all the following slots* of a table we accessed will be suitable for that job as well, since they are ordered. As for the binary case, after a choice has been made, Hermes updates its own view of the chosen execution site: the values of each field of the a.vector are adjusted in correspondence to the values of the fields in the r.vector. For example, if we are using the size of the run queue as the index for the cpu, we will increase by one unit (the process we just allocated) the size of the run queue in the remote machine, and if the i/o traffic is our i/o metric, we will increase it by the submitted job's i/o rate. The true values will be restored after t_1 time units by the new a.vector coming directly from the machine. Note that the local machine must still be checked first to see whether a local execution is reasonable or whether the maximum hop count has been reached.

8. EVALUATION CRITERIA

We now list some important criteria for evaluating a load balancing scheme, and use each one of them to assess our own scheme.

- *Scalability*: How does the scheme react to an increase in the number of the machines?
  We saw that every Hermes broadcasts its own machine's load information on the network. Too many processes exchanging messages can load considerably both the network and the machines that have to process and organize the incoming information. The scheme is not scalable to very large networks. However, it has been demonstrated [Livny1982a] that there is a good chance that, in a *sufficiently large* network, some machines are found idle, or very lightly loaded, while some others are too busy. For example, in a network with 10 machines, if the average system utilization is between 0.4 and 0.8, the probability that one node is idle when another one is overloaded is higher than 0.9. It might be sufficient, therefore, to keep load information only about *a fraction* of machines, chosen according to some selection rule as in [Barak1984a] , to ensure a reasonably high probability of finding a satisfactory execution site. This allows us to set an upper bound to the growth of the a.matrix. The network traffic generated by the distribution of the information, though, will be growing with the number of the machines, causing increasing traffic problems that could be solved by piggybacking the load information on other packets that have to be sent anyway.
- *Flexibility*: Does the scheme work for different machines and configurations?
  The threshold scheme allows the maximum flexibility, with some risks of instability. The real metric has to be scaled into a unique range of values. This operation can be hard with very different architectures, requiring heuristic solutions to be devised case by case.
- *Tunability*: How easily can we modify the scheme, to adapt it to the characteristics of the load?
  This is the best advantage of the binary vector metric: we can use the thresholds as knobs, to tune the scheme up according to the real system's behavior. The real vector metric is harder to manipulate since it is based on the *real* values of the load indices.
- *Resource Cost*: How much does the scheme cost, in terms of resource consumptions?
  When the local load is small, the choice is going to be a local execution. It might be better just to turn the scheme off. But the load could grow and the load balancing process be still

sleeping. We believe it is necessary to provide a mechanism simple enough to work continuously without imposing a major overhead on the system. For example, we could monitor regularly only the local load, and start collecting remote load information only when the local load goes beyond the local threshold. The cost of the service would be the sum of the following components:

- The cost of maintaining fresh information on the load (always present).
- The cost of choosing the optimal execution site (always present, even if the local is the optimal).
- The cost of the remote execution, that is the sum of:
  * The cost of transmitting the command, possibly with input files.
  * The cost of execution (present also in the local case, of course).
  * The cost of resending results back to the terminal where the user is waiting.

9. THE EXPERIMENTS

9.1. The Setup

We built an experimental version of Ulysses to execute jobs locally and remotely while measuring their response times and the current system load. Using this tool we ran a series of experiments on a selected set of *probe jobs*, to assess both the improvements in response time Ulysses is able to provide and the relative performance of the real and binary metrics. We wanted to find out whether the choices based on the binary metric are as effective as the ones based on the real values.

The experiments were performed within the range of a full month, at various times of the day and the night, on machines running with their real users and their real loads. The setup included:

1) A *local* machine on which two processes were running, together with the normal load:
   - The first process encompassed the functions of Ulysses, Diogenes, and Cupid, executing and timing any jobs we submitted to it both locally and remotely. For convenience, remote execution was performed by *rsh*[RSHa], that creates two TCP/IP connections [Postel1981a] between the two machines, one for standard input and output, and one for the error messages and the transmission of signals, with a high setup cost. We measured the overhead of *rsh* by executing remotely several times a very light command (*date*), so that we could assume that the response given by *time* accounts only for the setup operations. When both the machines were unloaded, *rsh* averaged 4.9 seconds of response time, whereas when the local machine was loaded (with a load average of about 6.5) and the remote was unloaded, the response time was 5.6 seconds on the average. To give an upper bound, we mention also that *rsh* takes on the average 6.4 seconds to execute *date* between two equally loaded machines (load average about 6.5), even if a remote execution would never be chosen by Ulysses in these conditions. The major part of *rsh*'s delay is due to the layers of software to be crossed by the messages (connection setup, buffer management, checksums, input and output queues); the delay due to the transmission medium itself is probably much smaller: Cabrera *et al.* measured TCP/IP transmission delays of the order of a few milliseconds, with a maximum of a few tenths of a second in case of high load [Cabrera1984a]. However, the minimum delay we measured is suffered by any remote execution performed by means of *rsh*.
   - The second process realizes the functions of Hermes, reading from *kmem* (in Unix, a file that contains up-to-date information about the virtual memory) the values of cpu utilization, i/o traffic, free memory size and run queue size, and building an a.vector with them. Hermes reads new values every 30 seconds, and smoothes them with the old corresponding ones using an exponential smoothing algorithm with α=0.5 (i.e., the

previous value of the average is considered as important as the new value of the index). The resource consumption is extremely low (order of 0.1 % of cpu usage), since the process is almost always inactive, waiting either for the timeout to elapse or for a request of information from Ulysses, and when it is active it simply seeks the parameters in the *kmem* file. A request of information from Ulysses to Hermes takes an average round-trip time of 27 milliseconds.

2) A *remote* machine with its own (generally light) load, ready to receive the remote executions. A copy of Hermes runs on this machine too, and exchanges with its peer the information it collects using the UDP/IP message exchange protocol provided by the 4.2BSD inter-process communication facilities.

9.2. The Test Workload

For this set of experiments, we chose four test jobs to be used as probes, and we analyzed them through a series of runs, in order to compute their r.vectors. Diogenes, just a piece of the balancing process, maintained the r.vectors in a table.

The selected four jobs had the following characteristics:

- CC: The compilation of a 6Kbyte C program (the source code of Hermes), that exercises alternatively both the cpu and the i/o subsystem.
- IO: An i/o bound program (10 copyings of a 60K file), that exercises heavily only the i/o subsystem.
- CPU: A cpu bound program, that performs only arithmetic operations. Please notice that the cpu% (cpu utilization wrt real time) of CPU is slightly smaller than CC's and NROFF's: since CPU never releases spontaneously the cpu, it gets delayed by the scheduler.
- NROFF: The formatting of a 6Kbyte text, a chapter of this report: cpu bound and with a big memory resident set.

We assigned an r.vector to a process according to the following rules:

- In the real case, we inserted in the r.vector the cpu utilization of the job, obtained by dividing the cpu time used by the elapsed time, and the i/o rate, obtained by dividing the total number of i/o operations performed by the elapsed time. The data was obtained by running each job 75 times on an unloaded machine, measuring its response times with the *time* command, and averaging the values. The number of repetitions was chosen big enough to provide a standard deviation substantially smaller than the average value (no more than the 20% of this value). Table 1 shows the complete characterization of the jobs used in our experiments with the real-valued metric. The elapsed time is given in seconds, the i/o rate in number of blocks transferred to/from disk per second; the "memory" column shows the number of pages in the free list.

Table 1.
The Real-Metric Characterization of the Test Jobs

type of job	cpu%	elapsed	i/o rate	memory
CC	56.98	27.52	5.46	7
IO	20.75	17.28	27.70	7
CPU	51.10	13.09	0.0	5
NROFF	51.59	46.83	0.4	73

- In the binary case, we put a 1 in the cpu field and a 0 in the i/o field (i.e., access key 10: 2) for the jobs we considered cpu bound (CPU, NROFF), a 1 in the i/o-field and a 0 in the cpu field (i.e., access key 01: 1) for the i/o-bound job (IO), a 1 in both fields (i.e., access key 11: 3) for CC, that is a heavy consumer of both resources. Table 2 shows the characterization of the jobs in this case.

Table 2. The Binary-Metric Characterization of the Test Jobs			
type of job	cpu	i/o	memory
CC	1	1	7
IO	0	1	7
CPU	1	0	5
NROFF	1	0	73

- The memory field contained the real memory (resident set) size of the process in 1024 byte units, as given by the RSS field in the output of the *ps* command.

9.3. The Execution

We ran a separate session for each one of the probe jobs. For each session we executed the probe many times, recording each time the local and remote response times and the corresponding a.matrix. In this way we were able to construct in the load hyperspace two related clusters of values: those of the local and of the remote response time. Note that Ulysses executed a job *both* locally and remotely at every step, and simply *printed out* the load and the execution times, to give us a way to determine the choice it would have made in the real and binary case; we did this by computing the matching function by hand, and adjusting the thresholds to their optimal value. If Ulysses' choice agreed with the best execution site (i.e., the one where the job executed faster), it scored a *win*, otherwise it lost. The total count of the wins was divided by the total number of the executions to yield the *win rate*. Two win rates were computed for each job type: w_{bin}, based on the binary matching mechanism, and w_{real}, based on the matching of the real values.

We plotted the response time versus all the load indices that constitute the a.vector, obtaining many projections of the response time set of points onto the component planes. The projections which turned out to be of interest are shown in the following description of the results.

9.4. The Results

9.4.1. CPU: Cpu Bound Probe Job

The win rates of Ulysses's choice for the cpu bound job, that is based only on the cpu_{idle} field, since CPU does not perform any i/o operation, are
for the binary:

$$w_{bin} = \frac{158}{289} = 0.54$$

and for the real:

$$w_{real} = \frac{161}{289} = 0.55$$

The cpu was considered busy if its idle time was less than 10% (local load threshold). Of the 158 right choices of the binary scheme, 69 were local and 89 were remote, whereas of the 161 of the real scheme, 90 were local and 71 were remote. The real scheme tends to favor very slightly the local choice, but without a really dramatic benefit. It is clear that this decision strategy is not the best we can do: we simulated random placement on the same data and we got a win rate of

$$w_{rand} = \frac{142}{289} = 49.135$$

not too different from either one of them. If we observe carefully the behavior of the cpu_{idle} field, though, we notice that it is 0 the great majority of the times. The few times when $cpu_{idle} \neq 0$, local execution is better, but even when $cpu_{idle} = 0$ the response time can be excellent. Many times Ulysses chose the remote machine on the basis of this criterion, and the choice turned out to be wrong. There is a clear negative correlation between cpu utilization and response time, as figure

16 shows, but a much greater variance corresponds to small values of the cpu_{idle} (see how scattered are the values for cpu_{idle}=0). The response time, in the next as well as in all the following figures, is given in seconds.

Figure 16
Local Response of CPU

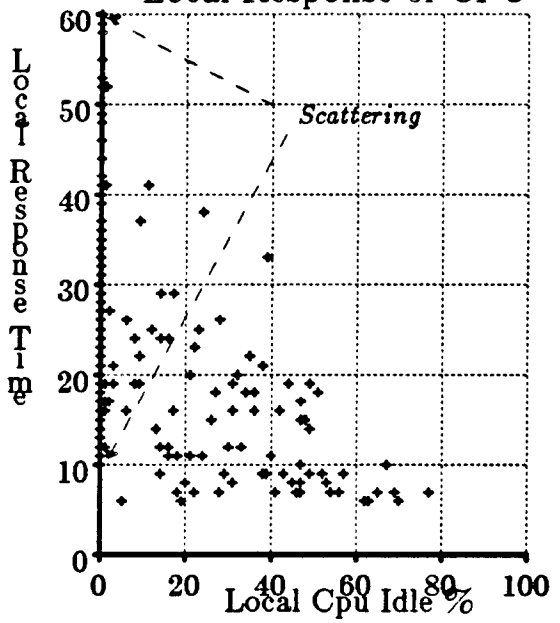
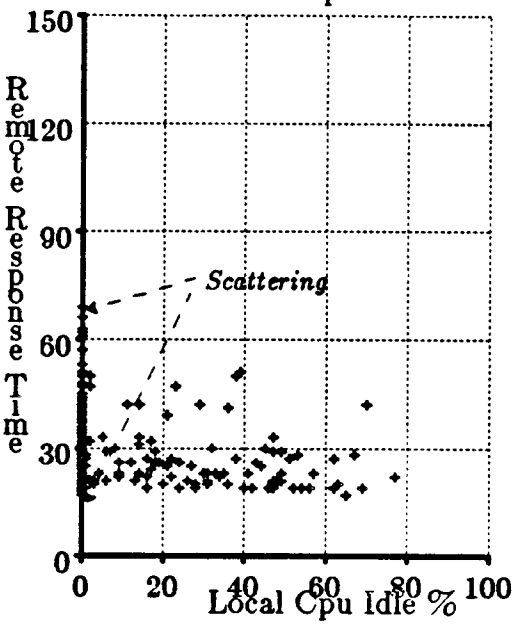


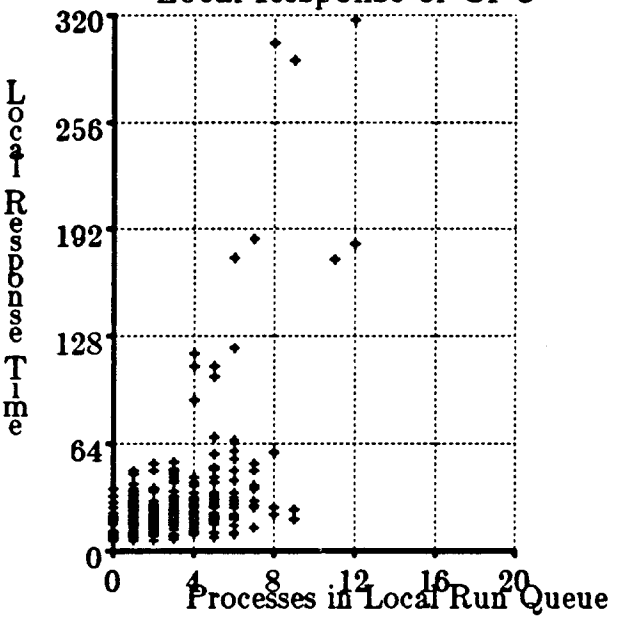
Figure 17
Remote Response of CPU



The variance of the remote response time, however, is always limited, since we always choose for our remote execution the *right* machine, that is an unloaded one: figure 17 shows that the remote execution time can still be bigger than the local one, but has a smaller variance.

Even though it makes matching very easy, the use of cpu utilization as an exclusive load index is to be considered a bad idea: it does not really tell how *fast* the service provided by the cpu will be at a given moment. A few jobs in the cpu queue are enough to bring the idle time down to zero, but this does not necessarily mean that the response time will be bad.

Figure 18
Local Response of CPU



A possible alternative index to consider is *la*, the number of processes in the running queue, that shows a good correlation with the response time of CPU (figure 18) and a good negative correlation with cpu_{idle}. We can observe that the variance is limited when the number of processes in the run queue is small, but increases dramatically when this number grows (please compare this figure with figure 6 in Chapter 4). We consider the machine busy (setting the cpu bit to one) when the local run queue is bigger than a certain local threshold. We computed the binary win rate when the choice was based on this index. We used a local load threshold of 4, as suggested by the figure, and a delta threshold of 3, i.e., Ulysses did not send the process out unless the cpu queue of the target machine was smaller that the local's by more than 3 processes. We got a considerable improvement in the win rate:

$$w_{la} = \frac{206}{289} = 0.7128$$

This suggests that the number of processes in the run queue is a better load index for cpu bound jobs with a binary matching scheme.

9.4.2. IO: I/O Bound Probe Job

The idea of matching the per job i/o rate with the i/o traffic does not turn out to be very effective: if we try to perform the matching in this way, the local machine is always chosen in despair, since the high i/o bandwidth (27.7 blocks/sec) required by IO is hardly available in mass storage systems with a maximum bandwidth of 40 blocks/sec; the win rate corresponds actually to the win rate of the local execution (about 0.5), i.e., we win only when we execute locally. As we observed in Chapter 4, if we know that a job does *some* i/o, it is sufficient to keep it out of a machine where the i/o traffic is close to the saturation point. The relationship between i/o traffic and response time for IO is represented in figure 19. There is a clear positive correlation between the two quantities, while the variance increases with the traffic.

Figure 19
Local Response of IO

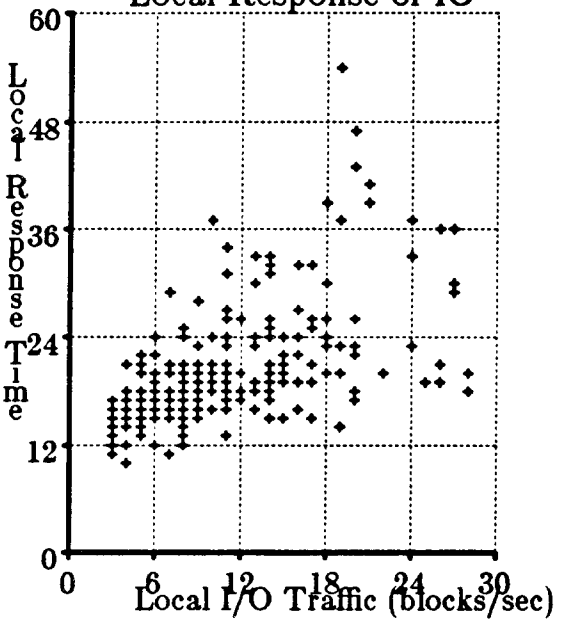


Figure 20
Remote Response of IO

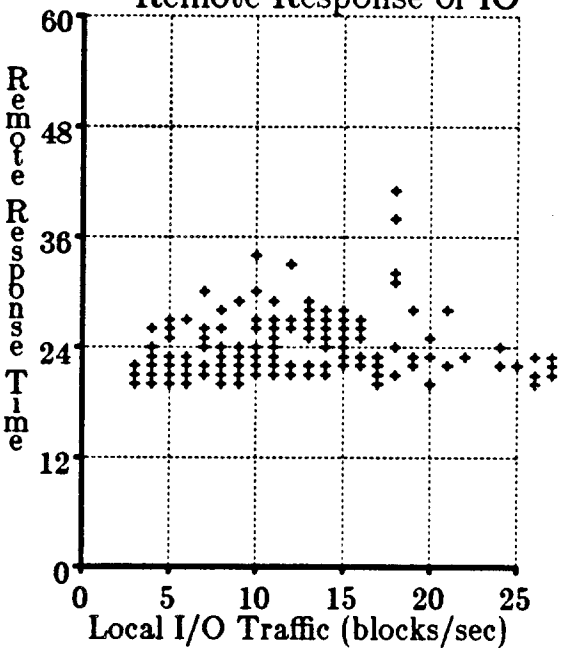


Figure 20 shows that, on the contrary, the response time is quite constant on the remote machine, and is better than the local when the local traffic has passed a certain threshold. We originally set this threshold to 10 blocks/sec, but we realized that Ulysses was sending the i/o bound job out too easily, with unsatisfactory results. By increasing the threshold to 15 blocks/sec, we got a remarkable win rate:

$$w_{bin}=\frac{252}{284}=0.88$$

We can consider the i/o traffic a definitely good load index for an i/o bound job, whereas we feel that the i/o rate is too detailed to characterize properly a job's i/o requirement: a simple *i/o need* bit is probably sufficient.

9.4.3. Compilation

The compilation job, balanced in its cpu and i/o requirements, offers probably the most difficult choice among our test jobs. The binary metric provided an unsatisfactory:

$$w_{bin}=\frac{121}{261}=0.4635$$

whereas the real metric gave:

$$w_{real}=\frac{164}{261}=0.628$$

The inaccurate indication Cupid gets from the cpu_{idle} field is responsible for this not very exciting result. A compilation has a *bursty* behavior: it alternates cpu and i/o service requests, and we observed in Chapter 4 that bursty processes are favored by the scheduler. Cupid always chooses the remote machine when it bases its choice on the fact that $cpu_{idle}=0$, and many times this is wrong because the job could still obtain a fast local service, thanks to the scheduler. The real metric has a higher win rate because sometimes Cupid, not finding sufficient cpu power on the remote machine, comes back to the local in despair. However, the correlation between i/o traffic and response time is good: the graphs are similar to the i/o bound job's (figures 19 and 20): all the times that the i/o field determines the choice, Cupid wins. As for CPU, we recomputed the win rate for the binary metric using the number of processes in run queue as the cpu load index. We used the same thresholds, 4 for local and 3 for delta. We got:

$$w_{la}=\frac{163}{261}=0.625$$

that represents an improvement with respect to using cpu_{idle}, and encourages us to choose this index for the load of the cpu. Out of curiosity, we tried with a different threshold value for the local, 3: We got 5 more wins, not a very substantial improvement. The win rate that the number of running processes yielded turned out to be very close to the real metric's one. We must notice, though, that the local choices of the latter are due to desperation (neither of the execution places was considered good), while the former's are based on the actual value of the metric; we favor the binary approach because it is more likely to work well in other, more general situations.

9.4.4. Text Formatting

The win rate for the text formatting job, a typically cpu bound one, was quite good (local threshold: 4, delta threshold: 3):

$$w_{bin}=\frac{107}{133}=0.8$$

and:

$$w_{real}=\frac{103}{133}=0.77$$

Looking at the results in detail, we notice that remote execution gave better responses most of the times: the text formatting job is twice as heavy as the pure cpu bound job, and since it tends to use up all its cpu time slices, it sees its priority reduced by the scheduler very soon. If the local machine is not empty (i.e., if $cpu_{idle}=0$), the remote choice pays off.

We here observed that two jobs with *the same* resource-relative characteristics (both cpu bound) can perform very differently when Ulysses uses the same information policy. Once more, this result highlights the importance of an accurate workload characterization and suggests that any control law should keep into account both the behavior of the scheduler and the size of the input file.

Finally, we must mention that in all of our experiments the memory field never determined a wrong choice: there was always enough memory for a job to be executed, either locally or remotely. This is probably due to the peculiarity of our work environment, where the jobs are generally small, the text segment of the images is shared, and there is plenty of memory available. The jobs we selected are also typical of our work environment. There might be other environments where memory is a critical issue, e.g., where machines have relatively small main memories. Since maintaining information costs very little, we think we should keep memory usage anyway, to provide the scheme with more flexibility and generality.

10. CONCLUSION AND FUTURE WORK

The aim of our research was not to offer a full and comprehensive solution to the problem of balancing the load, but just to shed light on some issues in order to clear the way for future work.

There are two flaws in the design we are perfectly aware of (plus probably many others we are not):

- The information policy based on periodic broadcasts leads to large network traffic and higher processor load when many machines are connected to the network. We can get an idea of the impact of such a mechanism by looking at an analogous one already implemented in Unix, the *rwho daemon*, a distributed service that maintains information on the status of the network by means of periodic broadcasts. The *rwho daemon* is *always* among the 5 heaviest processes running on every machine, even if it runs only once every 5 minutes!
- The *local* response time improvements we observed do not necessarily imply that the *global* response will be optimal as well. If our own probe jobs got better response times, the happy users of the unloaded target machines must have paid something. We did not consider the

price of our intrusions.

There are also some crucial issues that we considered only marginally in our experiments:

- A general characterization of the workload is important. Even if we do not need a description of the requirements at the quantitative level, we must divide the jobs into classes and devise a separate strategy for each one of them.
- The influence of the size of the input file must be investigated with an experimental setup somehow *orthogonal* to ours: while maintaining constant load conditions, the same command should be fed with different-sized input files, and the response time monitored. The behavior of the scheduler can change the response time in a critical way, depending on the type of the command as well as on the size of the input file.
- As we already pointed out at the beginning of Chapter 9, the use of *rsh*, i.e., of the TCP/IP protocol, for remote execution does not yield optimal performances. We think there is space in this area for research on faster and more efficient mechanisms for remote execution.

Nevertheless, we feel we learned something from our experience. Both the binary metric and the multivariable information policy have turned out to be good instruments for a correct location policy: maintaining distributed information about the load is easy with not too many machines, and, when the load index is properly chosen, we can base effective choices on it.

We had hoped that cpu utilization would be a good load index because it is easy to match with a single job's cpu requirements. It turned out that cpu_{idle} does not produce good estimates of a job's possible response time, because it tends to be zero too frequently. The number of processes in the run queue gives a better indication of the load, and it too is fortunately very easy to match with a job's requirements through a binary scheme based on load thresholds. These thresholds have to be tuned manually, and their determination is an important factor of the success of the scheme.

The i/o traffic is a good load index for a specific class of jobs: the ones that make a large use of i/o; in this case, a binary, threshold-based selection is enough to determine choices that frequently provide a better response. We do not need too much information about the job, but only an indication of its requirements.

ACKNOWLEDGEMENTS

My work has received invaluable contributions from all the members of the Load Balancing sub-group of the Progres group: Domenico Ferrari, Luis Felipe Cabrera, Riccardo Gusella and Harry Rubin. To all of them a heartfelt thanks, in the hope that they liked working with me as I liked working with them. My best wishes to Riccardo Gusella for the fruitful and successful development of his career, and to Harry Rubin for the continuation of this research. My special expression of love to Berkeley and California, for having been a wonderful life environment for the two most productive and intense (so far) years of my life.

References

Alonso1983a.

R. Alonso, "The Design of Load Balancing Strategies for Local Area Network Based Distributed Systems," *PROGRES Group Internal Report*, CS Division, UC Berkeley, October 1983.

Barak1984a.

A. Barak and A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer," *Internal Report*, The Hebrew University of Jerusalem, Jerusalem, 1984.

Bryant1981a.

R.M. Bryant and R.A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. Int. Conf. Distributed Computer Systems*, pp. 314-323, 1981.

CSHa.CSH, *UNIX 4.2BSD Manual Page for csh (C Shell).*

Cabrera1984a.

L.F. Cabrera, E. Hunter, M. Karels, and D. Mosher, "A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD," *Report No. UCB/CSD 84/217*, UC Berkeley, December 1984.

Cabrera1985a.

L. F. Cabrera and G. Rodriguez-Galant, "Predicting Performance in UNIX Systems from Portable Workload Estimators Based on the Terminal Probe Method.," *IEEE Transactions on Software Engineering (to appear)*, 1985.

Cabrera1985b.

L.F. Cabrera, M. Karels, and D. Mosher, "The Impact of Buffer Management on Networking Software Performance in Berkeley UNIX 4.2BSD: A Case Study," *Procs. of the Summer USENIX Conference (to appear)*, Portland, Oregon, June 1985.

Chu1980a.

W. Chu, L. Holloway, M. Lan, and K. Efe, "Task Allocation in Distributed Processing," *Computer*, pp. 57-70, IEEE, November 1980.

Danzig1984a.

P. Danzig and N. Gal, "EXODUS: Process Migration in a Network of Personal Workstations," *CS 262 Final Project*, UC Berkeley, December 1984.

Eager1984a.

D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Dynamic Load Sharing in Homogeneous Distributed Systems," *Internal Report*, University of Washington, Seattle, October 1984.

Ferrari1985a.

D. Ferrari, "Dynamic Balancing of Distributed Systems Loads," *Presentation at the 7th ILP Conference*, Berkeley, March 14th, 1985.

Hwang1982a.

K. Hwang, W. Croft, G. Goble, B. Wah, F. Briggs, W. Simmons, and C. Coates, "A UNIX-Based Local Computer Network with Load Balancing," *Computer*, pp. 55-66, April 1982.

Krueger1984a.

P. Krueger and R. Finkel, "An Adaptive Load Balancing Algorithm for a Multicomputer," *Computer Science Tech Rep. #539*, University of Wisconsin - Madison, Madison, Wisconsin, 1984.

Livny1982a.

M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Comp. Networks Perf. Symposium Proceedings*, pp. 47-55, 1982.

Livny1984a.

M. Livny, "The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems," *Computer Sciences Technical Report #570*, University of Wisconsin, Madison, WISC, December 1984.

Ma1982a.

P.R Ma, E.Y Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Transactions on Computers*, vol. C-31, pp. 41-47, January 1982.

Ni1981a.

L.M. Ni and K Hwang, "Optimal Load Balancing Strategies for a Multiple Processor System," *Procs. 10th Intl. Conf. Parallel Processing*, pp. 352-357, August 1981.

Peterson1983a.

J. Peterson and A. Silberschatz, *Operating Systems Concepts*, pp. 115-117, Addison-Wesley, 1983.

Postel1980a.

J. Postel, "User Datagram Protocol," *RFC 768*, USC Information Sciences Institute, August

1980.

Postel1981a.
J. Postel, "Transmission Control Protocol," *RFC 793*, USC Information Sciences Institute, September 1981.

Powell1983a.
M. Powell and B. Miller, "Process Migration in DEMOS/MP," *Procs. of the 9th ACM-SIGOPS Symposium on Operating Systems Principles*, pp. 110-119, October 1983.

Presotto1983a.
D. Presotto, *UNIX 4.2BSD Manual Page for DSH*, 1983.

RSHa.RSH, *UNIX 4.2BSD Manual Page for rsh (Remote Shell).*

Ramakrishnan1983a.
K.K. Ramakrishnan and A.K. Agrawala, "A Resource Allocation Policy using Time Thresholding," *Procs. of PERFORMANCE '83*, pp. 395-413, North-Holland, 1983.

Smith1984a.
A. J. Smith, "Trends and Prospects in Computer System Design," *Report No. UCB/CSD 84/219*, Computer Science Division (EECS), UC Berkeley, December 1984.

Stankovic1983a.
J. Stankovic, "A Heuristic for Cooperation Among Decentralized Controllers," *INFOCOM Proceedings*, pp. 331-339, 1983.

Stankovic1985a.
J.A. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," *IEEE Transactions on Computers*, vol. C-34 n.2, pp. 117-130, February 1985.

Walker1983a.
B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Procs. of the 9th. ACM-SIGOPS Symposium on Operating Systems Principles*, pp. 49-70, October 1983.

Wang1985a.
Y. Wang and R. Morris, "Load Sharing in Distributed Systems," *IEEE Transactions on Computers*, vol. C-34 n.3, pp. 204-217, March 1985.

Zatti1984a.
S. Zatti, "A Load Balancing Feasibility Study Using DSH," *CS 266 Final Project*, UC Berkeley, May 1984.

Zhou1983a.
S. Zhou, "A Survey on Load Balancing in Multiprocessor and Distributed Systems," *PROGRES Group Internal Report*, Fall 1983.