AN EXPERIMENTAL EVALUATION OF LOAD BALANCING STRATEGIES

Rafael Alonso

CS-TR-112-87

September 1987

AN EXPERIMENTAL EVALUATION OF LOAD BALANCING STRATEGIES

Rafael Alonso

Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-3869

ABSTRACT

In this paper we report on an experimental study of load balancing strategies for a network of workstations. We have implemented a load balancing mechanism on which a variety of strategies have been tested. Our current implementation runs on a local area network composed of a variety of Sun workstations. Some of the difficulties involved in developing a practical load balancing mechanism are described, as well as our suggested solutions to those problems. Among the issues addressed are how to avoid instabilities in the decision policy (especially in the case where the load balancing mechanism must deal with incomplete load information), and how to implement load sharing in an environment where the individual processors are owned by separate users (clearly, on such networks, policies that evenly spread the load throughout the system are not appropriate).

AN EXPERIMENTAL EVALUATION OF LOAD BALANCING STRATEGIES

Rafael Alonso

Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-3869

1. Introduction

In many of today's computing environments, it is not uncommon to see a mix of idle and overloaded machines on the same network. This is specially true in local area networks of workstations, where users may use their machines only sporadically. It is also the situation in distributed systems where the workload requirements have a large variance throughout the day. This imbalance leads to a needless degradation in system throughput and to a large increase in mean response time. Although users may realize that there are cycles available elsewhere and individually execute their jobs remotely, we feel that there is a need for mechanisms that automatically perform this task.

In the past, most work in the area of load balancing has been carried out within a theoretical framework. For example, Stone [1] approached this problem by modeling task allocation as a graph partitioning problem, while Chu [2] chose instead to answer load balancing questions by translating them into 0-1 integer programming problems. The trouble with such approaches is that, while they shed a certain amount of light on the subject at hand, they also abstract away much of the complexity which must be addressed in an implementation.

Our work has consisted of actually implementing a load balancing mechanism, in order to study load sharing policies with as large a degree of realism as possible. Our system runs on an Ethernet [3] local area network of SUN workstations of various models. The software consists of a set of cooperating daemons that periodically transmit load information, and local shell (i.e., command interpreter) programs that use that information to decide on the appropriate execution site for user jobs. As shown in Figure 1, users present their commands to a local shell (lsh) which in turns queries a local daemon (rld) which has been exchanging load information with rld daemons at other machines on the network. Based on the information provided by the local rld, the lsh software decides to run the incoming job locally or to run it remotely (which it does using the help of a remote slave, lshd). We refer the interested reader to [4] for more details of our implementation.

Although we have developed only a very simple load balancing scheme, our measurements show that, even under conditions of relatively small load imbalance, sizable performance gains can be achieved. For example, as shown in [4], C compilations running on a loaded machine can be made to run almost 4 times faster by using our implementation. Furthermore, the overhead involved in running our system is very small (for both users and non-users of our mechanism). That is, users can expect our implementation to provide them with as fast a response time as they could achieve by themselves if they were fully aware of load conditions in the entire network. And those network users who do not wish to utilize the load balancing functionality find that

their performance is not significantly degraded by the overhead of running our system.

In the process of developing our implementation, a number of problems have arisen. In the remainder of this paper, we will describe some of those problems and our current thoughts on solving them. The first of the four topics we will concentrate on is the issue of dealing with incomplete load information. The second is the problem of deciding upon a reasonable definition of load. The third involves the accuracy of statistics gathered by operating systems. The last one addresses the special requirements that must be met by load balancing implementations in networks of independently-owned workstations. Finally, we present some conclusions about the general usefulness of load balancing schemes.

2. Incomplete Information

In any load balancing implementation, load information must be communicated among all the participating nodes. The most appropriate manner for accomplishing this on an Ethernet is to have each machine broadcast its load at periodic intervals. The main difficulty with this approach is that, if the broadcast interval is not frequent enough, the load information data may be stale (i.e., out of date) by the time the decisions have to be made. On the other hand, the frequency of broadcasts cannot be increased arbitrarily since there is a cost involved in obtaining, sending and receiving data. Clearly, the more often load information is communicated, the higher the overhead of the balancing mechanism, and the more network bandwidth is wasted. Most importantly, processors must be interrupted in order to receive network messages, and may not be able to carry out any useful work if the number of such messages becomes too frequent. For example, our measurements show that an otherwise idle processor that has to deal with approximately 20 incoming load messages per second will have a CPU utilization of about 25%, while if there are about 45 incoming messages per second the CPU utilization for this task is more than 50%.

However, it is also dangerous to send information updates too infrequently. This is best made clear by the following example. Consider the case in which one machine on the network is is completely idle. In that case, all the other processors may decide (based on the last load information message sent by that machine) to send their jobs to the idle processor. Very soon, that ‘‘victim’’ machine is overloaded, and (if the broadcast interval is sufficiently large), other machines will not find out about the change in load. Then, when everyone finds out that the victim processor is overloaded, all the machines may pick the same ‘‘least loaded’’ site and the situation just described is repeated once again. In such a scenario, the entire network runs at the speed of a single (and overloaded!) processor.

In Figure 2 we show the results of an experiment where the above problem becomes clear. Here, a simple-minded load balancing scheme is used, that is, one where jobs are sent to the least loaded processor. Load information is broadcast by each processor at 20 second intervals. In this experiment five users were active on a network of four processors. (Each user is modeled by a script that correspond to a typical user work pattern of editing a program, compiling it, running the resulting object code followed by some thinktime.) Each of the points along the X-axis represents the initial placement of users on the network (i.e., 2,1,1,1 means that two users were at the first machine and one user was logged on to each of the other three processors). The Y-axis marks the average response time of the user scripts. The experiment was repeated multiple times to achieve statistical accuracy.

The line marked ''no lsh'' is the result of executing all the jobs without the benefit of load balancing. It represents the performance that could be expected in most distributed systems. The line labeled ''lsh'' marks the response time obtained under our simple load balancing scheme. As can be seen, load balancing only helps when the system is already extremely imbalanced (i.e., all the users at the same site). The victim problem described above accounts for the poor performance of the load balancing software.

We have explored a variety of techniques to minimize problems with stale data. For example, to try to eliminate the victim problem, we have tried a ''required difference'' policy. That is, a machine will only move a job to a remote processor if that processor's load is less than the local one by a certain amount. The intuition behind this is that if the difference is small, chances are that it has already disappeared by the time that the local processor is ready to send the job but that disappearance has not yet been reflected in the broadcast loads. The results of this policy are also shown in Figure 2, for the case of a required difference parameter of 1.0 (that is, a job is executed remotely if the remote site has a load average smaller than the local one by 1.0 or more) by the line marked ''lsh (delta=1.0)''. While the performance improves somewhat from the first case, it is still not satisfactory for the cases where the initial placement is relatively balanced.

A second technique that also can be tried to alleviate the stale data problems is to try to reduce the uncertainty in the load information in the following fashion. Whenever a job is sent to a remote processor, the sending processor can update its own value of the load at the other site, for example by increasing by 1.0 the load value it received from the remote processor the last time the latter broadcast its load. The results of this strategy are also shown in Figure 2, in the line marked ''lsh (gamma=1.0)''. As can be seen, this new technique still does not help much in the balanced network case, but obtains a greater improvement than before in the unbalanced network cases.

Finally in Figure 3, we show the results of combining both strategies. Now we can see that our load balancing scheme shows a performance improvement over the no load balancing case for all cases. Clearly, the combined technique provides us with the means for dealing effectively with the problems inherent in making scheduling decisions based on old data. We refer the reader interested in more details of these policies to [5].

3. Load Definitions

One of the first problems that arise in the context of designing a load balancing mechanism is that of deciding on an appropriate definition of ''load''. Clearly, since we are developing a system for balancing the system load, a clear definition of it is required before work can proceed. Furthermore, since we will have to decide at some point whether a machine is ''more loaded'' than another, the load definition employed must be a single-dimensional quantity.

At first glance this issue might seem a simple matter of defining load as a function of the load statistics available from the operating system. However, it is not clear how much weight to give to the individual statistics of each of the different resources in a computer system. That is, if for a given machine we are told the amount of free memory, CPU utilization, and the number of disk requests per second, it is not obvious how much emphasis to give to each of these items. In addition, the load function itself is bound to be quite complex. For example, if there is enough memory to contain the working set of the currently executing jobs plus that of the job being scheduled, any memory statistics will be mostly unimportant. However, if the arriving job's memory requirements are so large that they may push the system into thrashing, those

same memory statistics become of paramount importance.

Finally, it is not even clear that a single definition of load will suffice for most jobs. Consider the following example. To a CPU-bound job, the load at a site with mostly I/O-bound processes is very slight. This is so because the processing-intensive job will have few competitors there for the CPU. On the other hand, to an incoming I/O-bound job, the load at that machine will seem quite high. Thus, load may not only be a quite complex function, but it may also depend on the job currently being load balanced.

Fortunately for us, the situation is much simpler in the type of environments that we are interested in, i.e., in local area networks of workstations. In such networks, the current technological trends are towards the use of diskless nodes, that is, machines that store and retrieve data from a file server across the network. Since in this case the disk traffic is the same for all processors, the I/O load component of load need not be taken into account. A second technological development is that of the declining price of memory. Thus, it now becomes relatively inexpensive to provide a workstation with more than enough memory to run its local jobs. Hence, for our purposes, focusing on the CPU load seems to be sufficient for defining load.

Although we have carefully considered the design of load metrics [6], in practice we have found that using the UNIX load average (i.e., the exponentially averaged number of processes in the run queue over the last minute), is adequate for our needs. This load metric is simple to obtain (it is already computed by the operating system) and our load balancing implementation has achieved good performance with it.

4. System Statistics

As was mentioned in the previous section, in principle a definition of load may be based on a variety of performance indices provided by the operating system. But this assumes that the required information will be made available by the operating system. Furthermore, it implies that those statistics will be correctly gathered and compiled at a frequent enough rate. These assumptions may or may not be met by the available operating systems.

For example, in Berkeley UNIX [7], which is the operating system used for our experiments, it has been noted that many of the system statistics are computed in such a way as to be almost useless (see [8] for a description of some of the problems empirically observed). Additionally, in that system, many of the statistics gathered by the software are not computed all the time, but rather at 5 second intervals, which may be too infrequent for some applications. Finally, in the case of the load average (already described in the previous section) which is the performance index we are particularly interested in, it is computed as an exponentially-smoothed average over a number of time intervals (1, 5, and 15 minutes). Even if the figure computed for the shortest interval is used, it is clear that the load average will require about a minute to ''react'' to large changes in processor load.

Up to now, we have not addressed the question of the correctness of the statistics gathered, or fully investigated the effect of the sampling rate on system statistics. However, we have carried out a brief experiment on the effect of the averaging interval used in computing the load average. We modified the operating system kernel to compute the load average at much shorter intervals (down to 15 seconds). Our measurements show that while the performance of the load balancing mechanism improves somewhat (in particular, it is more resilient to the stale data problems described in a previous section), the improvement is not significant enough to warrant a permanent modification of the Berkeley UNIX kernel.

5. Independently-Owned Machines

This area of research is motivated by the following observation. It seems to us that there are really two environments in which load balancing may be of use. One is that of a network where there are a number of machines owned by a single entity; there, it is desirable that load should be evenly balanced across all the machines. A second one is that of a network of workstations, where individual users own their machines; while those owners may not mind that someone else is ''borrowing'' a few cycles while they are not fully utilizing their processors, they certainly are not willing to see their own response suffer in order to help the overall system response time. While most of the work in load balancing relates to the first kind of environment, we feel that the second one also merits careful study, and presents an entirely new set of design requirements. We are currently developing policies for networks of independently owned workstations, as well as exploring techniques that can be used in systems that consist of a mix of the two environments.

Although this aspect of our research is not yet over, our ideas on this topic have centered on the following approach. In order to provide users with a well-defined method for specifying how much they are willing to help others, our scheme consists of allowing each processor to set both high and a low watermark. If the load of a machine is below the low watermark, that processor will not consider itself overloaded and thus will be willing to execute processes sent to it by other machines. If the load is above the low watermark, that processor will deny all remote requests for job execution. The high watermark comes into play to determine whether a processor will look for help from other machines. If a machine's load is below the high watermark, that processor will consider itself underloaded and thus will never try to execute any of its jobs remotely. Conversely, if the load is above the high watermark, a network node will begin to look for another machine that will accept the remote execution of the node's jobs.

There are a number of details in this scheme that we will not describe here; please see [9] for the relevant information. However, it is clear from the descriptions given above that the watermarks divide the load space into three regions. If the system load is below the low watermark, the processor will execute all jobs locally and accept remote jobs for execution. In this case the machine can be said to be underutilized. If the load is between the low and high watermarks, the processor will cease accepting remote jobs but will continue to process locally all its jobs. We consider this the normal case for efficiently used machines. Finally, if the load rises above the high watermark, the processor will try to offload its jobs on some remote site (and continue to deny remote requests for execution).

A final comment on this scheme is that, if the low watermark is set above the high one, when the load is in the region between the two, the processor will be trying to offload its jobs while continuing to accept foreign ones. this will clearly result in a performance degradation as jobs will have to pay the extra communication overhead without any savings in processing time. At the present time we are integrating this scheme with the load balancing mechanism described in Section 1.

6. Conclusions

In this paper we have described our load balancing implementation, as well as detailing some of the problems we have encountered in designing a practical system. Four issues were discussed. The first was the difficulty of dealing with out of date information in load balancing systems in which participants broadcast their load at periodic intervals. A number of strategies were described which we feel are sufficient for dealing with this problem. Two other

discussions concerned the various difficulties encountered in defining a load metric and in obtaining the appropriate operating system statistics to compute those load metrics. Finally, some of the ideas involved in our current work on networks of independently owned workstations were described. For those networks, it is clear that load sharing is a more appropriate goal than load balancing. Although there are still many questions to be answered, the experimental results we have obtained so far in this work has convinced us that it is possible to construct a simple load balancing mechanism that can still achieve significant performance improvements in realistic situations.

Acknowledgements

I would like to thank Luis Cova, Phil Goldman, Peter Potrebic, and Ann Takata for the role they have played in this research.

References

1. H. S. Stone, ''Multiprocessor Scheduling with the Aid of Network Flow Algorithms,'' *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 83-93, January 1977.
2. W. W. Chu, L. J. Holloway, M. Lan, and K. Efe, ''Task Allocation in Distributed Data Processing,'' *IEEE Computer*, November 1980.
3. R. M. Metcalfe and D. R. Boggs, ''Ethernet: Distributed Packet Switching for Local Computer Networks,'' *CACM*, vol. 19,7, pp. 395-404, July 1976.
4. Rafael Alonso, Phillip Goldman, and Peter Potrebic, ''A Load Balancing Implementation for a Local Area Network of Workstations,'' *Proceedings of the IEEE Workstation Technology and Systems Conference*, March 18-20, 1986.
5. Rafael Alonso, Phillip Goldman, and Peter Potrebic, ''Methods for Coping with Stale Data in a Load Balancing Environment,'' Technical Report, Department of Computer Science, Princeton University.
6. Rafael Alonso, ''The Design of Load Balancing Strategies for Distributed Systems,'' *Proceedings of the U.S. Army Research Office Future Directions in Computer Architecture and Software Workshop*, May 5-7, 1986.
7. S. Leffler, W. Joy, and K. McKusick, *UNIX Programmers's Manual - 4.2 Berkeley Software Distribution,* U.C Berkeley, August 1983.
8. Robert B. Hagmann, ''Performance Analysis of Several Backend Architectures,'' Ph.D. Dissertation, U.C. Berkeley, p. 48, June 1983. Also U.C.B. Report No. 83/124, August 1983 (To appear in Transactions on Database Systems).
9. Rafael Alonso and Luis Cova, ''Sharing Jobs Among Independently Owned Processors,'' Technical Report, Department of Computer Science, Princeton University.

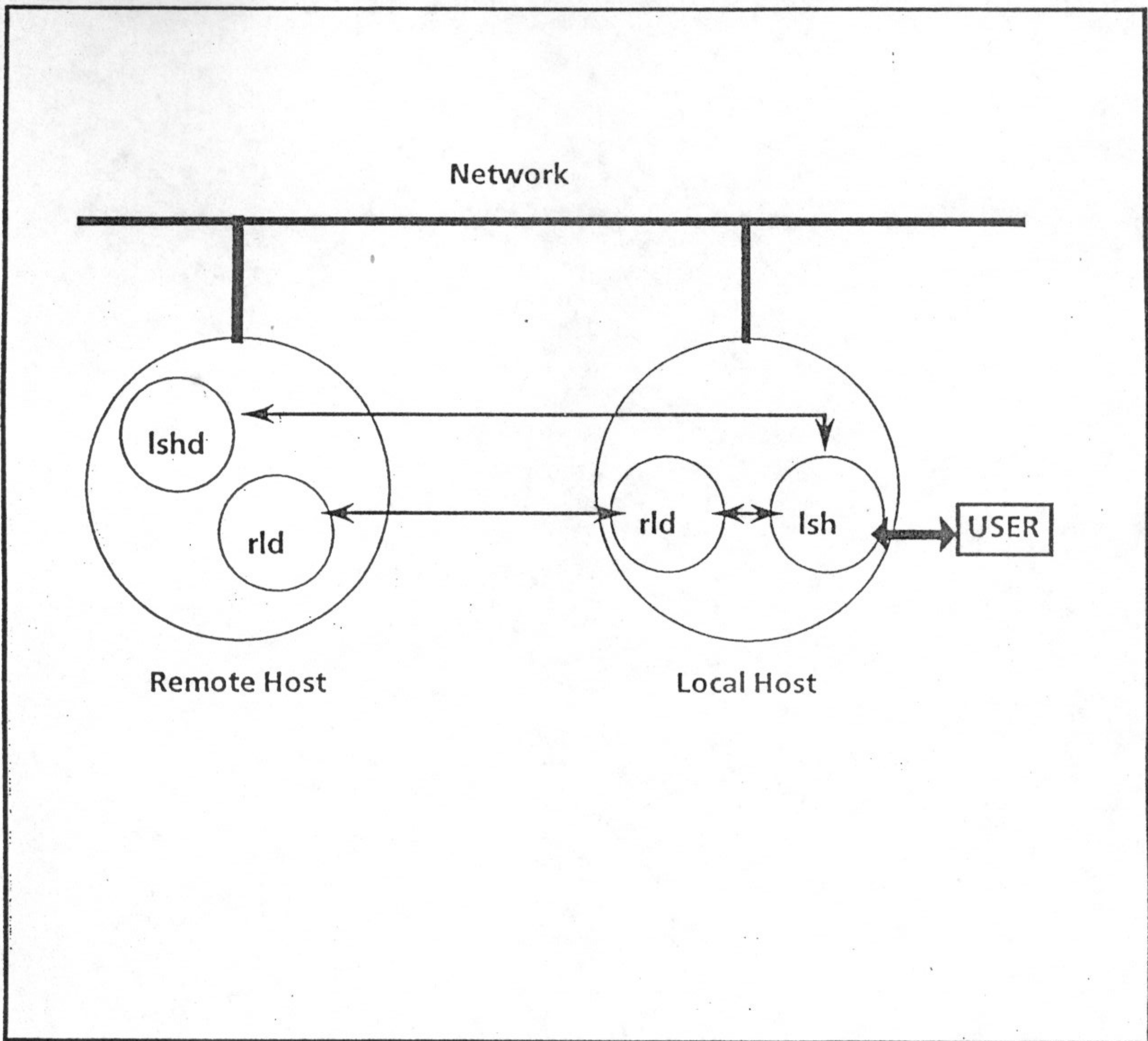

Figure 1

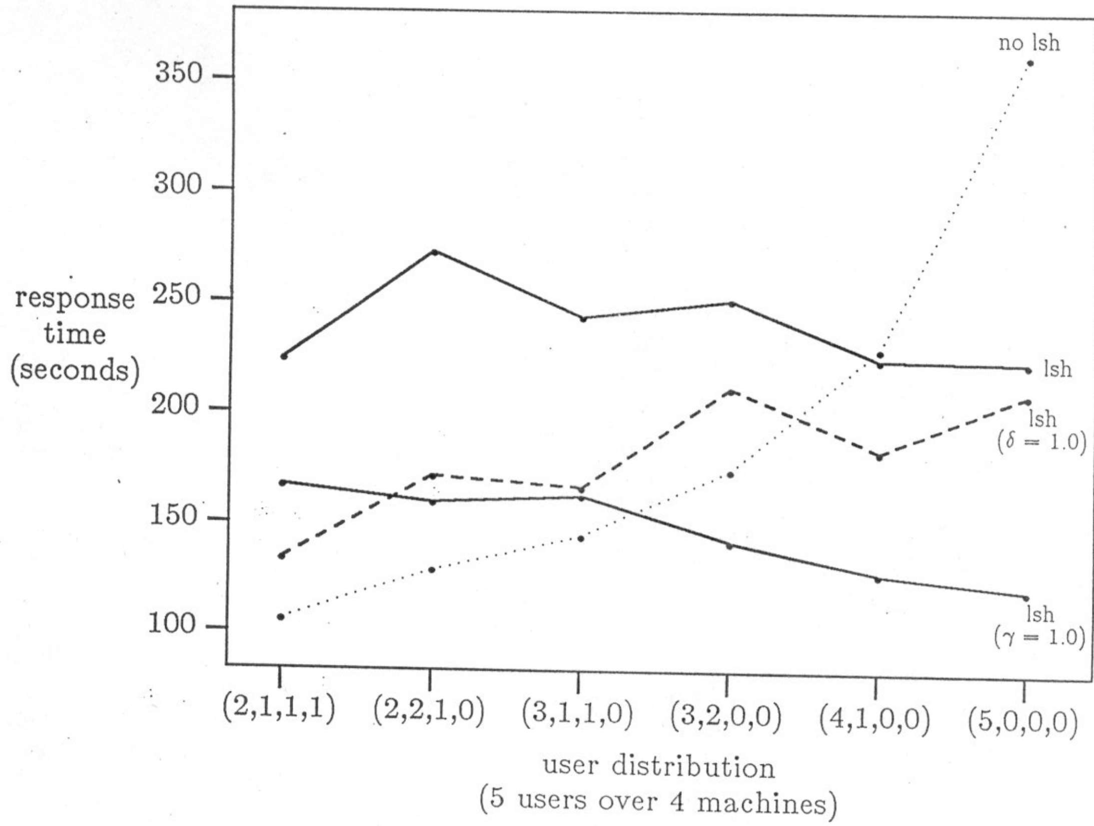

Figure 2

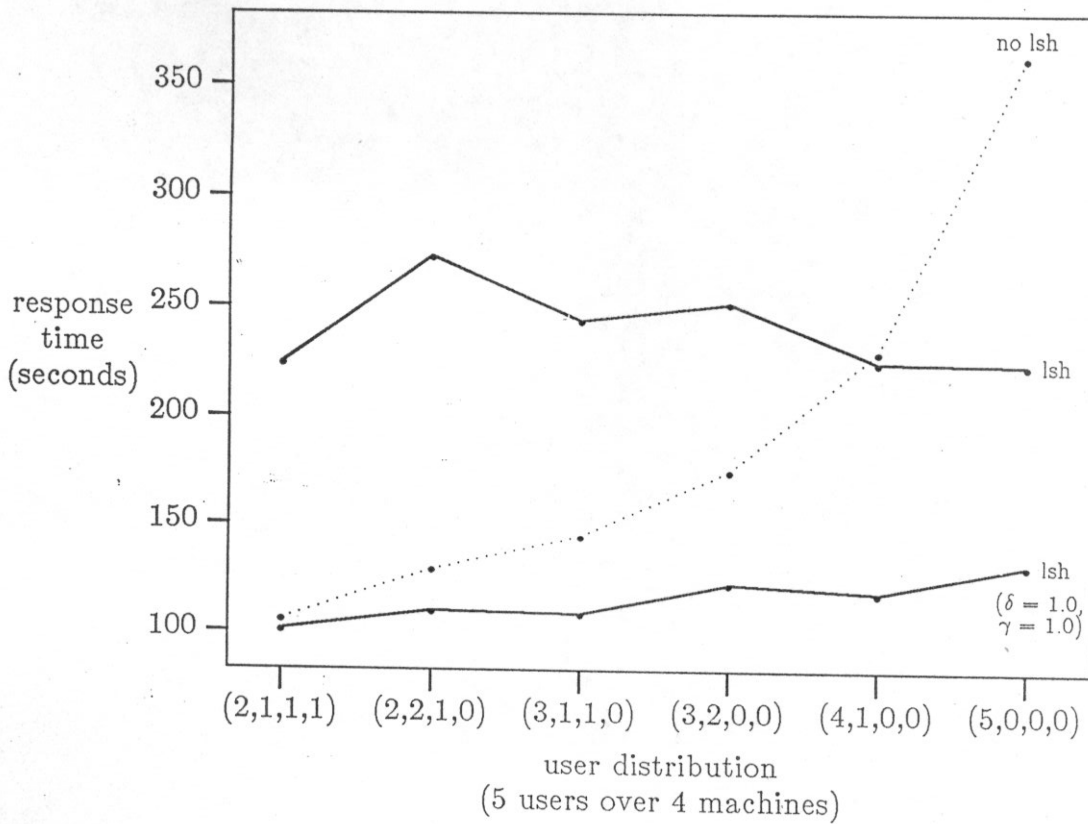

Figure 3