Stability of Load Sharing in a Distributed Computer System

by

Mahboob Ashraf

M. Sc., University of Dacca, 1974

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
September 1987

Approval

Name: Mahboob Ashraf

Degree: Master of Science

Title of Thesis: Stability of Load Sharing in a Distributed Computer System

Examining Committee:

Dr. Louis J. Hafer
Chairman

Dr. Tsunehiko Kameda
Senior Supervisor

Dr. Wo Shun Luk

Dr. Stella Atkins
External Examiner

Sept. 8, 1987
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

STABILITY OF LOAD SHARING IN A DISTRIBUTED COMPUTER SYSTEM

Author: _____
(signature)

MAHBOOB ASHRAF
(name)

September 9, 1987
(date)

Abstract

Load sharing attempts to improve the performance of a distributed system by making global scheduling decisions in a decentralized, adaptive way. Results from analytical and simulation studies indicate that load sharing policies of modest complexities are capable of achieving significant performance improvements. Very few of these studies, however, treat stability issues for distributed load sharing policies. In this thesis, the stability issues of three representative simple policies are studied. The performance of these policies under heavy and fluctuating load is evaluated with respect to specific stability issues and analyzed in terms of the response time. The dependence of the performance on the environment in which the policies are implemented and on the nature of interprocess communication is discussed.

Acknowledgements

It is my pleasure to acknowledge the contribution of my senior supervisor Dr. Tiko Kameda. His support and encouragement were always there when I needed them. I would like to thank Dr. Stella Atkins and Dr. Wo Shun Luk for reading the thesis carefully and making thoughtful suggestions. My discussions with Tony Speakman, David Mauro and Brian Terry were very helpful in clarifying my thinking. I also thank the staff of the Computer Science Instructional Laboratory for making the network available to me. Finally, special thanks to my wife, Isobel, for her support, encouragement and editorial help.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

A distributed computer system to be studied in this thesis consists of a network of more or less autonomous nodes. One of the advantages of such a system is its capability to deliver high performance through parallelism inherent in the system. In designing a distributed system, one needs to make crucial decisions whose effects on the system performance are not intuitively obvious. One of these design issues deals with the strategy for distributed scheduling of tasks[1]. Distributed scheduling involves two coupled subproblems: the *local scheduling* of processes within one node, and the *global scheduling* of tasks within the entire system. A major function of global scheduling is *load sharing*, which is the problem of allocation and sharing of system processing resources among the tasks. In this thesis we present a performance study of an important aspect of load sharing policies, namely, their *stability*. Our goal is to implement representative policies for load sharing in a LAN (local area network) based distributed computer system and study their performance with respect to specific stability issues. The purpose is to understand how the performance of these policies depends on the system state information, on how this information is collected and on the nature of communication. In this chapter we introduce the notion of load sharing and point out some of its desirable attributes. We will describe what we mean by stability of load sharing, discuss the characteristics of a stable load sharing strategy, and explain why it is important to study it.

---

[1]Task is the initial articulation of the work to be done, that is, the run-time description given to the operating system to invoke the execution of a particular piece of code. We shall also use two other terms: *process* which is the program object code plus the operating system run-time process control data structures which maintain and execute the object code; and *job* which is the entire work cycle, from task invocation through process execution.

1.1. Load Sharing

In a distributed computer system, it may happen that a task waits for service at one node, while at the same time another node which is capable of servicing the task is idle. One can prevent a system from reaching such a state by transferring part of the workload of a congested node to less congested ones for processing. This redistribution of system workload is called load sharing.

Load sharing distributes the workload of a system among the nodes according to some load sharing policy which is designed to achieve a system wide improvement in a specific performance metric such as response time, throughput, etc. In our model, the load sharing policy is executed by all the nodes in the system. The nodes communicate with one another without any centralized control and gather information about the changing load status of the system. On arrival of a new task, a node uses this information to make a scheduling decision according to the load sharing policy. Load sharing, thus, is the problem of making global scheduling decisions in a decentralized, adaptive way.

It is important to note the differences between load sharing and another aspect of distributed scheduling called *task assignment*. Task assignment deals with distributed programs, each of which is partitioned into communicating modules for which communication and service time costs are known. Its responsibility is to assign the modules optimally to processors so that the cost for a fixed task load is minimized. Task assignment is static in nature; the nature of task load, the cost of running each module on each processor, and the cost of network communications are all assumed to be known beforehand. Load sharing, on the other hand, is dynamic. It does not have *a priori* knowledge of the size of task load and it does not make any assumption about service-time and communication costs. A load sharing policy collects information on the fly and makes scheduling decisions dynamically. Also, it treats only logically independent tasks; tasks are the unit of assignment and not subdivided.

In attempting to fully exploit system processing power, a load sharing policy makes decisions based on small variations in load at different nodes. Also, the information about the system state on which these decisions are based is usually barely up to date. The requirement to react to small variations in loads at various parts of a system means that the inherent inaccuracy and the changing nature of system state information may cause a policy to behave in an unstable manner under certain input conditions (i.e., task arrival pattern), resulting in poor performance. To ensure good performance a load sharing policy should contain explicit or implicit damping mechanisms enabling it to make sound decisions under a broad range of system loads.

To be useful a load sharing policy should possess a certain set of attributes. First of all, it should be *effective* - it should not lead to an increase in average response time or decrease in throughput from what the system would have without load sharing. It should be *efficient* by not increasing the cost-to-benefit ratio. It should also be capable of either retaining or collecting a reasonable amount of knowledge about system state and make decisions on the basis of that knowledge adaptively. Last but not the least, a load sharing policy should be *stable*, that is, it should be able to handle any abrupt, sharp fluctuations in system load.

In the last few years load sharing has received a great deal of attention and a large number of load sharing policies have been proposed. Most of these policies have been evaluated either analytically or by simulation studies, and they are found to be effective and efficient under the conditions tested. These evaluations, however, make many assumptions about the averages and distributions of various parameters, and very few of them treat stability issues explicitly.

The analytical models assume known distributions of quantities such as the task arrivals, the service requirements of the tasks, etc. Furthermore, they assume the system to be in equilibrium and free from any irregular fluctuation in these parameters. Because of this, the stability issues of these policies are not at all addressed.

Simulation studies can in general deal more easily with changing network and system conditions, and they test a wider range of conditions than analytical models. Even then, these studies make many assumptions about averages and distributions. For example, most of them assume the arrival rate of tasks to be constant in time and the interarrival time to be distributed exponentially or hyperexponentially. Also, the simulation studies arrive at the values of various parameters suitable for a given system by making many runs and adjusting the values during each run. By determining the functional dependencies of the parameters on one another and adjusting the values accordingly, these simulations obtain good performance compared to analytical bounds and ensure that they do not perform too poorly under any reasonable input, and thus they address the stability questions implicitly. But, the functional dependencies are not constant. Also it is difficult to determine (or even identify) all possible functional dependencies that exist, for it is very difficult to define what a good performance is under all conditions. Thus, the question naturally arises: "are these load sharing policies stable?" Before we try to answer this question, we need to elaborate on what we mean by a stable load sharing policy and identify the characteristics of such a policy.

1.2. Characteristics of a Stable Load Sharing Policy

Informally, a load sharing policy is defined as stable if its response to any reasonable input does not exceed some bound on a given performance metric such as response time, throughput, etc [Stankovic 85]. Reasonable inputs include a wide spectrum of arrival rates with rapidly changing patterns and bursts of arrivals. The rate of arrivals may increase very rapidly during certain time of the day or some nodes in the system may periodically receive tasks that arrive at an enhanced rate. Occasionally, a system may experience a sudden burst of arrivals at all nodes at the same time. A load sharing policy should be capable of handling these inputs without causing serious degradation in performance. Also, it should be robust enough to deal with certain transients (inputs that may occur for a finite amount of time) such as a condition in which the

arrival rate of all tasks in the network is faster than the sum of the service rates of the network, and a sudden surge of arrivals following recovery from a network failure. It is natural that the performance will degrade under such severe input conditions. The degradation, however, should not be so drastic that it is felt long after the fluctuations have ceased to exist and the system in the long run would be better off without load sharing. A stable load sharing policy would not have such drastic performance degradation and should not take an unduly long time to relax back to an equilibrium state.

There are several reasons for the degradation of performance when load sharing is subjected to the kinds of inputs mentioned above. One reason is the over-movement of a task - a task being moved repeatedly from one node to another without being processed by any one. A number of policies attempt to prevent this problem by fixing an upper limit on the number of times a task can be moved. This, however, does not take into account the fact that the performance is usually load-dependent. Also, the costs of transferring tasks and gathering system state information may become prohibitive under heavy loads, and any movement of tasks under such conditions should be minimized. In general, there should be very little movement under very light and heavy loads, and a proper amount of movement for moderate loads. A stable policy would obtain good performance by ensuring "optimal" movement of tasks among the processors.

The performance can also degrade if one or more nodes are very lightly loaded and other processors experience a sudden increase in task arrival rate. The latter processors may react to this by transferring a substantial number of tasks to the lightly loaded ones. It is entirely possible that the lightly loaded processors quickly become overloaded and subsequently reallocate the newly arrived tasks. A load sharing policy must be able to identify these conditions for any number and type of burst arrivals and not over- or under-react.

1.3. Why Study Stability of Load Sharing?

Determination of stability of load sharing policies is an important issue. For a distributed computer system to be useful under inputs which contain short-term fluctuations, its overall performance should be better than the performance of a system without load sharing. For, it is in precisely these situations that one needs an improved performance from a system. A load sharing policy that fails to perform well under such input conditions is of little use. It is very important to determine how capable a load sharing policy is of adapting itself to a rapidly changing input environment.

Although the literature on load sharing policies is quite extensive, very few of the published works address the issue of stability directly. The two notable exceptions are those by Bryant and Finkel [Bryant & Finkel 81] and Stankovic [Stankovic 85]. The policy proposed by Bryant and Finkel estimates the remaining service time of a task that is already under processing and uses this time to decide which task to transfer from a congested node. This policy, thus, is concerned less with load sharing than with process migration which is much more complex in nature than load sharing. Stankovic [Stankovic 85] evaluated the performance of two specific policies under various types of burst arrivals and found that the techniques employed specifically to deal with the problem of instability did not always perform well.

Results from several recent analytical and simulation studies [Eager, Lazowska & Zahorjan 86; Speakman 86], on the other hand, indicate that load sharing policies of modest complexities are capable of achieving significant performance improvement. These studies, however, evaluated the policies under input conditions which are not specifically designed to bring out any inherent instability. Since these results are very encouraging, it is desirable to evaluate the performance of these policies under heavy and fluctuating loads with respect to specific stability issues.

There are other reasons for studying stability of load sharing. The quality of scheduling decisions depends crucially on the accuracy of system state information maintained by each node. But maintenance of up to date information about the system is difficult under heavy and fluctuating load. Consequently, the performance of load sharing policies becomes very sensitive to fluctuations in input conditions. As mentioned earlier, the performance of the policies proposed in the literature has been evaluated under ideal input conditions in which the rate of task arrival is assumed to be constant. In a less idealized situation the arrival rate will definitely be time-dependent, reflecting time-of-the-day effects and bursts of arrivals. It is important to study the load sharing policies under such input variations in order to understand their limitations, their dependence on the environment in which they are implemented, and above all to determine if they are stable in the sense defined earlier.

1.4. Organization of the Thesis

In the preceding sections we have introduced the problem of load sharing, discussed the meaning of stability and cited several motivations for undertaking a performance study of the stability of load sharing policies. The remainder of the thesis is organized as follows. In the next chapter we shall present a survey of various load sharing policies proposed in the literature. We shall describe some of these policies and summerize the results of various performance studies. Chapter 3 describes the design of our experiment and the details of the implementation of the load sharing policies we have chosen to study. We present the results of our experiment in chapter 4, and discuss these results and draw some general conclusions in chapter 5.

CHAPTER 2

LOAD SHARING POLICIES

The broad range of issues involved in load sharing and the various attributes a load sharing policy may possess have resulted in a large number of different policies. These policies differ from one another in the assumptions they make, the environments in which they are intended to be used and the specific aspect of system performance they are designed to improve. In this chapter we present a brief survey of the load sharing policies proposed in the literature. This survey is not meant to be an exhaustive one. We cite a few typical examples of load sharing policies and describe in greater detail only those which are similar in nature to the policies studied in the present work. A more comprehensive survey is given in [Wang & Morris 85].

2.1. Types of Load Sharing Policies

The goal of a load sharing policy is to transfer tasks from one node to another in order to maximize the overall performance of a distributed computer system. A load sharing policy thus needs to make two decisions: when to transfer a task and where to transfer it. Thus, every policy is built of two major components: a *transfer policy* which determines whether a task arriving at a node should be processed locally or transferred to another less busy node, and a *destination policy* that determines to which node a task selected for transfer should be sent.

Load sharing decisions can be made either by source nodes via which tasks enter the system or by the server nodes which eventually process these tasks. Depending on who initiates the load sharing actions, all distributed load sharing policies fall into one of two basic categories: *source-initiated* policies in which congested nodes search for lightly loaded nodes to which tasks

may be transferred, and *server-initiated* policies in which lightly loaded servers seek out congested nodes from which tasks may be received. Besides the difference in the types of nodes making scheduling decisions, the source-initiated and the server-initiated policies differ in a more fundamental way. Server-initiated policies typically require the transfer of already executing tasks, since tasks are not usually made to wait without being executed. Such process migration incurs substantial costs in most systems [Powell & Miller 83]. Source-initiated policies avoid such costly transfers by scheduling a task upon arrival, before it begins execution.

Load sharing policies can be further classified into *static* and *dynamic* policies, depending on the amount of system state information needed by a node to implement a policy. Static policies use only information about the predetermined average behaviour of the system; transfer decisions are based on prespecified system parameters independently of the actual current system state. Static policies may be either deterministic or probabilistic. In a deterministic policy the sources and the servers are assigned to one another in a fixed partitioning (e.g., send tasks selected for transfer by node X to server Y). A probabilistic policy by contrast transfers tasks from one node to others with a prespecified probability (e.g., send tasks selected for transfer by X to servers Y and Z with probabilities 0.8 and 0.2, respectively).

The principal advantage of static policies is their simplicity; there is no need to maintain and process system state information. The lack of this information, however, makes them incapable of responding and adapting to short-term fluctuations in the task load. The dynamic policies attempt to rectify this by maintaining up to date state information and base scheduling decisions on the knowledge of system state. This makes a dynamic policy more complex, but at the same time allows it to achieve significantly better performance than static policies.

2.2. Examples of Load Sharing Policies

In this section we present a short survey of various load sharing policies proposed in literature and describe them very briefly. The static policies that allocate servers permanently or

semi-permanently to arriving task streams are considered in [Stone & Bokhari 78; Chu, *et al.* 80; Chou & Abraham 82]. All these works formulate a load sharing problem as a mathematical programming or network flow problem and make scheduling decisions by performing optimizations against some chosen performance metric. The policy of Li and Abani [Li & Abani 81] distributes task randomly among all servers according to a given probability distribution. The policies which distribute tasks among the servers according to some cyclic schedule are considered in [Yum 81; Agarwala & Tripathi 81].

More recently, Eager, *et al.* [Eager, Lazowska & Zahorjan 86], have proposed three policies which use only local state information and do not exchange any information among the nodes in deciding whether to process a task locally or remotely. A task is transferred from a node if its arrival causes the node's load to exceed a predetermined threshold.

The three destination policies which Eager, *et al.*, have used in their policies are referred to as *random* , *threshold* and *shortest.* The random policy selects a server at random and transfers a task to that node for processing. The receiving node treats it just as a task originating at the node. If the local load is below threshold the task is accepted for processing: otherwise it is transferred to some other node selected at random. The system keeps track of the number of times a task has been transferred, and once this number exceeds a static *transfer limit,* the task is processed by the node that has received it last. This stabilizes the policy against a sudden increase in task load.

In the threshold policy, a source node selects a potential server at random and probes that server to determine its load. If that load is less than a prespecified threshold, the task is transferred; otherwise a different server is chosen for probing. This procedure is repeated until the task is transferred or the number of servers probed exceeds a predetermined *probe limit.* If the task cannot be transferred, it is processed by the source node where it originally arrived. When a server receives a task that is transferred from a different node, it always processes the task irrespective of any change in its load status during the interval between probing and arrival

of the task.

The shortest policy is a simple variation of the threshold policy. Instead of choosing one node, a source node selects at random L distinct nodes all of which are probed for their load status. The task is transferred to the node with the smallest load that is also less than the threshold.

The transfer and destination policies of the static policies described above use only local state information and a few predetermined criteria in making load sharing decisions. The one exception is the set of policies proposed by Eager, *et al.* The destination policies of these policies are based on the information about the load status of a small subset of nodes. The dynamic policies we describe in the following, by contrast, exchange considerably more information among all the nodes in a system and use it to decide both when and where to transfer a task.

Ni and Abani [Ni & Abani 81] have considered a *join-the-shortest-queue* policy in which each source independently assigns a new task to a server with the least number of tasks waiting in its queue. Livney and Melman [Liveny & Melman 82] have proposed two dynamic policies in which the nodes base their decisions on queue lengths. The interesting aspect of Livney and Melman's work is their model of the distributed system: a collection of homogeneous processes connected by a network which supports broadcast. In these policies, one of which is source-initiated and the other is server initiated, the nodes collect information about system load via a broadcast mechanism.

The source-initiated policy of Livney and Melman, called the *state-broadcast* or STB policy, is a version of join-the-shortest-queue and contains a strong notion of balance. In this policy the load status of a node is characterized by the number of tasks executing and waiting to be processed at that node. Each node maintains a system state vector which contains the load status of all the nodes in the system. Whenever a node receives a task or completes execution of a task, it broadcasts a message describing its new load status. On receipt of a broadcast message, every

node *i* updates its system state vector and transfers a task to a node *j* if the following conditions are satisfied:

1) if the number of tasks at node *i*, *n(i)*, exceeds *n(j)* by a given percentage of *n(j)*,
2) if node *i* has more tasks than every other node, and
3) if node *j* has fewer tasks than every other node.

This policy attempts to balance the system load by reducing load disparity between nodes when the disparity exceeds a certain threshold.

The server-initiated counterpart of the above policy is the *broadcast-when-idle* or IDB policy. This policy employs a transfer policy that requires less use of the network by broadcasting messages only when necessary. A node broadcasts a message only if it enters an idle state. On receipt of this message, each node invokes the following algorithm. If the number of tasks *s(i)* at a node *i* is greater than one, the node *i* delays for an interval inversely proportional to *s(i)*. If no other node has already broadcast a message reserving the idle server during the delay, node *i* broadcasts a reservation message. If the server is still idle and *s(i)* is is still greater than one, a task is transferred to the server. Since the node with the longest queue delays the least and thus is the one to transfer a task to the idle processor, this policy behaves like a *serve-the-longest-queue* policy. The interesting feature of this policy is that it tries to minimize the network utilization by sending messages and transferring tasks only when it is absolutely necessary. Also, it tries to share the load by not allowing a processor to remain idle (as opposed to the STB policy which attempts to keep the system load balanced).

There are several other policies which use mechanisms similar to those proposed by Livney and Melman. These policies base their transfer decisions on the minimum and maximum values of some load metric. If the node's load is below minimum, it broadcasts its willingness to accept tasks. If the load exceeds the maximum it attempts to transfer incoming tasks to other nodes. A description of these policies can be found in [Wang & Morris 85].

The policies mentioned so far are all designed to reduce response time or increase throughput. But none of them is capable of dealing with situations where tasks have to meet some sort of deadline. Consequently, these policies are not applicable to real-time distributed systems. Recently, Ramaritham and Stankovic [Ramaritham & Stankovic 84] have proposed a scheduling policy based on contract bidding algorithm [Smith 80] applicable to real time systems. In this policy, when a new task arrives at a node, an attempt is made to schedule it at that node. If this is not possible, the scheduler on that node interacts with those on other nodes, using a bidding policy and determines to which node the task can be transferred. The bidder part of the scheduler at each node maintains information about surplus processing capacities of other nodes. This information is derived from information passed between nodes in bids or is explicitly requested from other nodes. When a node needs to transfer a task, it utilizes this information to request bids from nodes with surplus processing power. On receiving a request for a bid, a server may submit a bid for work depending on some criteria, namely, the response deadline (a deadline before which the biddings should arrive at the requesting node), work queue length, availability of its resources, etc. The requesting node queues all bids until the response deadline. Once the deadline passes, it computes the task's estimated arrival time at each bidder's node and for each bidder the surplus (i.e., the time from this estimated arrival time to the task's execution deadline). The task is sent to the bidder with the largest surplus, and the rest of the bidders are notified that their bids are not accepted.

2.3. Performance of Load Sharing Policies

To test the performance of a load sharing policy it is necessary to begin with a model that reflects the design decisions inherent in the policy. One way to do this is to use analytical models such as queueing theory models. Such a model usually consists of N identical $M/M/1$ queueing systems corresponding to N processors. The relevant parameters (e.g., arrival rate, mean service time, queue length thresholds, network delays, etc.) are fixed and the solution is obtained by exact

analysis. Analytical models are useful in testing performance of simple static policies in which the system parameters are predetermined and can be modeled easily. For more complicated policies which involve parameters that are time-dependent and not easily amenable to modeling, a simulation written in some high level simulation language (e.g., SIMSCRIPT, GPSS, etc.) is used.

Once the model is chosen, we need to specify the performance metrics against which a given policy is to be tested. The choice of a performance metric depends on the specific attributes of the policy and on the nature of applications for which the system is designed. The metrics most frequently used in the literature include the *turnaround time*, the *mean response time*, the *network utilization* and the *throughput*. Wang and Morris [Wang & Morris 85] uses a very different metric called the *quality of load sharing factor* or the *Q-factor*. The Q-factor of a load sharing policy is the ratio of the mean response time over all tasks in a system under a first-come-first-serve (FCFS) policy to the maximum mean response time over all nodes in the system under the given policy. It is designed to measure how close the system comes to a multiserver FCFS system.

Besides being tested against one or more performance metrics, a load sharing policy should be evaluated against some general requirements. It is generally accepted that the more information a node has about the system load, the better capable it will be of making the best scheduling decisions. But, the ability to collect large amount of system state information also adds to the complexity of a load sharing policy. This added complexity and the maintenance of completely up to date information, however, may contribute significantly to the cost as well as to the communication overhead. The question that arises naturally is whether this added cost is worthwhile.

A partial answer may be derived from the performance evaluation of Livney and Melman's STB and IDB policies. Livney and Melman [Livney & Melman 82] model their system as a queueing network with FIFO (first-in-first-out) discipline and uses SIMSCRIPT simulation to

compare the turnaround time for these policies for varying number of nodes. Both policies result in turnaround times that are less than those obtained without any load sharing. But as more nodes are added to the system, the turnaround time in these policies decreases till a critical number of nodes is reached (the values of this critical number are different for different policies with the STB having the lowest). Any further addition of nodes results in an increase in the time because of excessive utilization of the communication channel. Higher channel utilization causes an increase in message queueing delays and thus slows down the load sharing process. Although the STB policy performs better than the others when the number of nodes is less than the critical value, the STB policy reaches its critical value much faster than the other two policies, making it inferior for a larger number of nodes.

The STB policy, it should be noted, is capable of making better scheduling decisions by virtue of more state information made available to it. But it breaks down due to excessive communication overhead as the number of nodes in the system increases beyond a certain level. This type of breakdowns, in general, are more common for policies that rely on broadcast because of their large information dependency and the accompanying communication overhead.

The question of appropriate complexity is also addressed by Eager, *et al.* [Eager, Lazowska & Zahorjan 86]. In evaluating the performance of their *random, threshold* and *shortest* policies, Eager, *et al.*, found that the threshold policy which uses little state information in its destination policy performs significantly better than the random policy which uses no information in deciding on the destination for a task. The threshold policy, however, performs marginally worse than the shortest policy, even though it utilizes less information than is used by the shortest policy. However, a complex policy may not always be more advantageous than a simple one because the former may become inefficient, unstable, and may lack robustness.

Next, we consider the performance of source-initiated and server-initiated policies. The relative performance of these two types of policies depends crucially on the cost of task transfers.

A source-initiated policy usually schedules (and transfers if necessary) a task upon its arrival. The server-initiated policies, on the other hand, typically require the transfer of already executing tasks.

In a comparative study of these two types of policies, Eager, *et al.* [Eager, Lazowska & Zahorjan 85], found that both source-initiated and server-initiated policies offer performance advantages over the situation with no load sharing. If the cost of task transfer under the two policies are comparable, server-initiated policies are preferable at high system loads and source-initiated policies are preferable at light to moderate system loads. But, if the cost of task transfer under server-initiated policies is significantly greater than that under the source-initiated policies, the source-initiated policies provide uniformly better performance under all system loads. Modifying server-initiated policies to transfer only newly arrived tasks yields unsatisfactory performance, even though it avoids the cost of transferring

The above discussion gave a general idea of the nature of performance we may expect from different types of load sharing policies. But the lack of uniformity of assumptions made in various studies and the idiosyncratic features of different policies make it very difficult to ascertain the performance gain that is potentially achievable from a particular policy. To rectify this situation, Wang and Morris [Wang & Morris 85] have proposed a set of canonical policies and compared their performance using a variety of queueing theory models and solution techniques under a uniform set of assumptions.

The canonical set of Wang and Morris includes both static and dynamic policies which use a first-come-first-served (FCFS) local scheduling policy. In the deterministic static policies either the sources are partitioned into groups and each group is served by one server or the servers are partitioned into groups and each group of servers serves one source. In the latter policy, a FCFS queue is formed at each source and tasks are removed and served one at a time by one of the available designated servers. In their probabilistic counterparts, each source distributes the tasks

randomly and uniformly to each server (*random splitting*) or each server visits a source at random and serves a single task (*random service*). The canonical set also contains a few cyclic policies. In the source-initiated *cyclic splitting* policy each source assigns its ith task to i(mod K)th server, K being the total number of servers. In *cyclic service* policy, the server-initiated dual of cyclic splitting, a server visits the sources in a cyclic manner and removes one or more tasks from the queue for processing.

The dynamic policies in the set include a *join-the-shortest-queue* policy, its server-initiated counterpart, the *serve-the-longest-queue* policy, and a multiserver FCFS policy that selects the least recently arrived task for an idle server.

If the overhead for interprocess communication (IPC) is considered negligible, the server-initiated policies like the cyclic service, the random service, and the serve-the-longest-queue policies all produce the same mean response time averaged over all tasks as the first-come-first-served (FCFS) policy. This is because these policies simply result in an interchange of the order of processing from FCFS and thus, by Little's law, do not change the mean delay averaged over all jobs. The join-the-shortest-queue (JSQ) policy, however, performs poorly compared to FCFS, indicating the possibility of a job waiting at a server while another server is idle. This situation is more pronounced in source-initiated random and cyclic splitting policies which have no knowledge of server states.

In case of negligible IPC overhead the load sharing capabilities (as measured by the Q-factor) of server-initiated policies are found to be superior to those of source-initiated policies using the same level of information. The server-initiated policies, however, are much more sensitive to the load distribution (degrading rapidly when the load is unbalanced) than their source-initiated counterparts. Also, as the number of servers are increased, the source-initiated policies suffer degradation of performance, while the server-initiated ones, such as the cyclic service policy approach the ideal load balancing. The most interesting aspect of these results is that a simple

policy like the cyclic service that uses very little state information is as capable of good performance as more sophisticated policies.

One, however, needs to be cautious in drawing any general conclusion from the above results which are valid only if the IPC overhead is negligible. The overhead introduced by IPC is highly implementation dependent. Consequently, a performance analysis in which IPC overhead is taken into account is difficult to perform and may not produce very conclusive results. Wang and Morris found that as IPC overhead increases the cyclic splitting policy may become more advantageous than the cyclic service because the cyclic splitting has no polling latency and may have smaller overhead per job. But as overhead increases further, the cyclic service eventually becomes preferable if it processes more than one task whenever it visits a source.

None of the studies we have mentioned so far addressed the stability issues directly. The only exception is a simulation study by Stankovic [Stankovic 85] of the bidding algorithm [Ramaritham & Stankovic 84]. Stankovic evaluated the performance of this algorithm under various types burst arrivals. Recall that in this policy the scheduling decisions are based on the information about surplus processing capacities of the nodes in a system. This surplus is estimated by using a variation of moving average forecasting technique. It is based on the assumption that the surplus processing capacity of a node in the future will be about the same as that in the recent past. The simulation results indicate that estimating the surplus in this way results in a sinusoidal pattern in the surplus for lightly loaded nodes. This causes an underloaded node to win all the bids during a certain time period, and then have all the tasks corresponding to those bids arrive in the next time period, resulting in a very busy node.

CHAPTER 3

EVALUATION OF STABILITY

In the previous chapters we have introduced the subject of load sharing in a distributed computer system and described some of the known results related to the performance of a few load sharing policies. These results were obtained by using analytical models and by simulation studies, which did not address the stability issues directly. The evaluation of stability (in its full generality) of a load sharing policy is not easy, because it is difficult to capture the intuitive notion of stability in a a more formal and quantitative definition than we have given in section 1.3. Such a definition needs to include not only the issues discussed in chapter 1, but also the numerous special cases that are particular to the environment in which a policy is implemented, to the policy itself, and to the designer's notion of what constitutes good performance.

It may, however, be less difficult to evaluate the stability of a load sharing policy if we implement it in an actual computer system and evaluate its performance, focusing our attention only on a small subset of stability issues instead of considering the entire set. This thesis is an experimental exploration of this alternative. We have implemented a set of three load sharing policies in a computer system running under UNIX, and studied their performance under inputs designed to put severe strain on their capabilities. Our objectives are to determine whether or not these policies improve the overall performance of a system under such inputs, and if they do not , to determine the reasons for their failure and to suggest possible remedies. The goal is to identify the characteristics of these load sharing policies, understanding of which should help the design of stable policies.

In this chapter we describe the design and the implementation of our experiment. We will first give a brief description of the three policies chosen for study and discuss the motivations behind our choice. We will describe the system in which these policies are implemented, and outline the various components of the software that runs this experiment.

3.1. Choice of Policies

The three policies we have chosen to study are: the threshold policy of Eager, *et al.* [Eager, Lazowska & Zahorjan 86], a broadcast based policy, and a policy with centralized control (called the centralized policy). These are all source-initiated policies, in which tasks are scheduled upon their arrival at the system, before they begin execution. All three policies use the same transfer policy that uses the load status of only the local host and does not use any state information of the nodes. A task is selected for transfer if its arrival causes the load status of the source node to exceed a prespecified threshold. These load sharing policies also use the same destination policy: a potential server is selected and if this server's load is below the threshold a task is transferred; otherwise the task is processed at the source node. They, however, differ from one another in the way a potential server is selected, in the amount of state information collected and maintained by the nodes and in the way this information is collected.

These policies can be characterized as simple in the sense that they use relatively less system state information than other more complex policies proposed in the literature. The reasons for our choosing simple policies over more complex ones are as follows. It is generally accepted that under a complex policy a node has more system state information and thus, is better capable of making the best scheduling decision than a node under a simple policy. A complex policy, however, may not always perform significantly better than a simple one to justify the added cost of collecting the extra state information. In fact complex policies are found to perform marginally better than the simple ones [Eager, Lazowska & Zahorjan 86]. It is more important to study the stability of less complex load sharing policies, for these policies tend to react more slowly to

changes in the system state and thus, are inherently less susceptible to variations causing instability.

The choice of the broadcast policy and the centralized policy can be motivated as follows. In the broadcast policy the nodes try to maintain up to date system information by broadcasting individual load statuses. Consequently, the broadcast policy is susceptible to breakdown under heavy load due to excessive communication overheads. A load sharing policy with centralized control is potentially capable of delivering a good performance. The central node, however, can also become a bottleneck in a system under heavy load, leading to poor performance.

3.1.1. Threshold Policy

In this policy, the only state information a node maintains is its own load status which is characterized by the number of jobs being processed at the node. A newly arrived task is selected for transfer to a remote node if the local load is greater than or equal to a prespecified threshold.

To find the destination for a task selected for transfer, the policy chooses a potential remote server at random and probes that server to determine its load status. If the load of the remote node is less than the threshold, the task is transferred; otherwise a different node is selected for further probing. The source node repeats this until it finds a destination node or the number of servers probed exceeds a probe limit. If a task cannot be transferred, it is processed by the source node.

3.1.2. Broadcast Policy

In this policy, each node maintains a system state vector containing the load statuses of all the nodes in the system. Whenever the load status of a node changes with respect to a threshold (that is if it goes above or below threshold), that node broadcasts a message containing its new load status. On receipt of a broadcast message, every node updates its system state vector. Thus, the system state vector effectively contains the busy/idle statuses of the various nodes in the

system.

To find the destination for a task selected for transfer, a node searches its system state vector and selects a server whose load is less than the threshold. If it fails to find a server, the task is processed at the source.

It is important to note the difference between this policy and the state-broadcast (STB) policy of Livney and Melman [Livney & Melman 82]. In the STB policy a node broadcasts a change of load status whenever it receives a task or completes execution of a task. Under a heavy load this generates a large volume of network traffic. In the present policy, a node broadcasts a message only if its load goes above or below the threshold, and thus, on the average generates considerably fewer messages. It also does not try to balance the load. The major difference between these two broadcast policies, however, lies in the status of a task just prior to a transfer. Since the present policy attempts to transfer a task on its arrival, it always transfers a task before the task begins execution. The STB policy attempts to transfer a task when a node updates its system state vector. In an operating system in which tasks are queued before being dispatched for execution, the STB policy is able to transfer a task that has not begun execution. In a multiprogramming system like UNIX, in which the degree of multiprogramming is unbounded, the STB policy inevitably results in a transfer of a task which has already started execution (process migration).

3.1.3. Centralized Policy

This policy is very similar to the broadcast policy, except that it employs a centralized control. Instead of having a separate load sharer at each node, the entire system has only one load sharer. The central load sharer maintains a system state vector which contains the load statuses of all the nodes. A node communicates any change in its load status with respect to a prespecified threshold to the central load sharer who then updates its system state vector.

If a newly arrived task is selected for transfer, the source node requests the central load sharer to find a server for the task. The load sharer searches its system state vector and selects a server with a load less than the threshold. If there is no such server, the task is processed at the source.

3.2. Implementation

We have implemented these three load sharing policies in a network based multicomputer system and studied their performance by using a controlled workload. The workload itself is generated artificially. That is, the arrival of tasks at the system and the service requirements of these tasks are generated by using appropriate statistical distributions. The reason for using an artificial workload is straightforward. We would like to have total control over the input, so that we can change its nature at will and study load sharing under different types of inputs.

The communication aspects of the policies, however, are not simulated. We have used an actual network and implemented the parts of the policies related to communication almost the same way as they would be in a working system. The advantages of using a real network instead of a simulated one are that it frees us from dealing with a very large number of parameters, and also, allows us to draw conclusions about the performance with respect to a particular network environment. Our system model is, thus, a hybrid one; parts of it are simulated and the rest are the real physical system.

In this section we outline the design and implementation of our experiment in detail. We will describe the generation of workload, explain the hardware and software components of the implementation, and discuss several design decisions.

3.2.1. Generation of Workload

The purpose of our experiment is to study load sharing policies under inputs which reflect time of the day effect, sharp fluctuations in task arrival rate and burst of arrivals. We can simulate the effects of these non-stationary characteristics by using inputs in which the rate of task arrival has sharp peaks of widths that are small compared to the duration of a trial.

This type of arrival process can easily be modeled by a nonhomogeneous Poisson process. A nonhomogeneous Poisson process is essentially the same as the more familiar Poisson process, except that its rate function, instead of being a constant, is time-dependent.

A nonhomogeneous Poisson process has the characteristic properties that the number of events in any finite set of non-overlapping intervals are independent random variables, and that the number of events occurring in any interval has a Poisson distribution [Cinlar 75; Cox & Lewis 66]. The most general nonhomogeneous Poisson process can be defined in terms of a monotone nondecreasing right-continuous function $\Lambda(t)$ which is bounded in any finite interval. Then the number of events in any finite interval, for example $(0, t_o]$, has a Poisson distribution with parameter $\mu_o = \Lambda(t_o) - \Lambda(0)$. $\Lambda(t)$ is called the integrated rate function and has the interpretation that, for any $t \geq 0$,

$$\Lambda(t) - \Lambda(0) = E[N(t)],$$

where $E[N(t)]$ is the expected value of $N(t)$, the total number of events in $(0, t]$. Its right derivative

$$\lambda(t) = \frac{d\Lambda(t)}{dt}$$

is called the rate function of the process. In contrast to the more familiar homogeneous Poisson process (in which $\lambda(t)$ is constant, usually denoted by λ), the intervals between the events in a nonhomogeneous Poisson process are not identically distributed.

A nonhomogeneous Poisson process with a general rate function $\lambda(t)$ can be easily simulated by a method based on the following theorem [Lewis & Shedler 79].

THEOREM: *Consider a non-homogeneous Poisson process* $\{N^*: t \geq 0\}$ *with rate function* $\lambda^*(t)$, *so that the number of events,* $N^*(t_o)$, *in a fixed interval* $(0, t_o]$ *has a Poisson distribution with parameter* $\mu_o^* = \Lambda^*(t_o) - \Lambda^*(0)$. *Let* $T_1^*, T_2^*, ..., T_{N^*(t_o)}^*$ *be the events of the process in the interval* $(0,t_o]$. *Suppose that for* $0 \leq t \leq t_o$, $\lambda(t) \leq \lambda^*(t)$ *holds. For* $i = 1, 2, ..., n$, *delete the event* T_i^* *with probability* $1 - \lambda(T_i^*)/\lambda^*(T_i^*)$; *then the remaining events form a nonhomogeneous Poisson process* $\{N(t): t \geq 0\}$ *with rate function* $\lambda(t)$ *in the interval* $(0, t_o]$.

In the present work, we generate nonhomogeneous Poisson arrivals by using the following Algorithm.

ALGORITHM:

1. Generate points in the nonhomogeneous Poisson process $\{N^*: (t) \geq 0\}$ with rate function $\lambda^*(t)$ in the fixed interval $(0, t_o]$. If the number of points generated, n^*, is zero, then exit; there are no points in the process $\{N(t): t \geq 0\}$.
2. Denote the (ordered) points by $T_1^*, T_2^*, ... T_n^*$.. Set $i = 1$ and $k = 0$.
3. Generate U_i, uniformly distributed between 0 and 1. If $U_i \leq \lambda(T_i^*)/\lambda^*(T_i^*)$, set k equal to $k + 1$ and $T_k = T_i^*$.
4. Set i equal to $i + 1$. If $i \leq n^*$, go to step 3.
5. Return $n = k$ and $T_1, T_2, ... T_n$.

The method of thinning based on the above theorem can be simplified considerably if we use $\lambda^*(t) = \lambda^* = \max_{0 \leq t \leq t_o} \lambda(t)$. Then $\{N^*(t): t \geq 0\}$ becomes a homogeneous Poisson process with a constant rate function λ^*. In an actual implementation, we supply the rate function $\lambda(t)$ as a function of time t, and compute $\lambda(t)$ for any t in the interval $(0, t_o]$ by a piecewise cubic spline interpolation. We also take $\lambda^*(t)$ to be a constant λ^* equal to the maximum of $\lambda(t)$ in the interval

$(0, t_o]$. The interarrival time, $t_{i+1} - t_i$, between a pair of events i and $i+1$, is obtained by generating and accumulating exponential (λ^*) random numbers $E^*_{1,i}, E^*_{2,i}, ...$, until the following is satisfied for the first time :

$$u_{j,i} \leq \lambda(t_i + E^*_{1,i} + ... + E^*_{j,i})/\lambda^*,$$

where $u_{j,i}$ are independent uniform random numbers between 0 and 1. This process is continued until the specified number of events is generated or until the time t_o is exceeded.

In the present implementation, a task consumes the cpu resources by executing a loop. Hence, we have represented the service requirements of the tasks in terms of the number of times a task executes this loop. We assume that this number is distributed exponentially, and generate it by using an inverse integral transformation. One could object that the choice of exponential distribution for service requirements may not be very representative of the actual workload in a computer system. Under a different distribution a load sharing policy may perform very differently. In fact, the assumption of hyperexponentially distributed service demands increases the absolute response times achieved by certain load sharing policies [Chou & Abraham 86]. It is unlikely, however, that the choice of a different distribution would affect the relative performances of load sharing policies in which we are interested.

3.2.2. Hardware

The system in which the investigation is conducted consists of eight SUN-2 workstations (each with 2 megabytes of memory and running SUN UNIX 3.2) connected by a 10 Mb Ethernet. We have used a ninth machine to run the central load sharer in our implementation of the centralized scheduler. The load sharing policies are implemented as a collection of user (or application) processes. The processes involved in scheduling decisions are run at a priority level higher than that of the tasks.

Each processor runs its own copies of the processes that implement a policy. These processes communicate through the Ethernet among themselves for making global scheduling decisions. The file service is provided by three separate SUN-2 workstations using SUN's Network File Servers. The central features of the system are:

1. it is a homogeneous system,
2. the interconnect is a broadcast network with CSMA/CD access discipline, and
3. the network is completely isolated and used exclusively for this investigation.

3.2.3. Software Architecture

We have mentioned earlier that the load sharing policies are implemented as collections of user processes which communicate through an Ethernet among themselves to make load sharing decisions. Figure 3.1 presents a schematic diagram of the software architecture, and illustrates the interprocess communication. In this figure each circle represents a process as labeled. A box groups the processes which run together in a single processor.

An arrow between two circles denotes a communication link between the respective processes. The arrowhead indicates the direction of the message transfer and the label specifies the protocol used by the communication link. The solid arrows represent communication on the *loopback* between two processes on the same processor. The loopback is a software mechanism provided by the network layer to allow processes on the same processor to communicate without using the network medium. The communication link between two processes residing in different processors, by contrast, is represented by a broken arrow indicating communication over the Ethernet. All the TCP connections represented by solid arrows are established before we start making measurements, and the costs of creating these links are not included in the service-time cost. The TCP connections to a remote machine (represented by broken lines), on the other hand, are established and dissolved dynamically during the trial, and the associated overhead is included in the cost of task transfer.

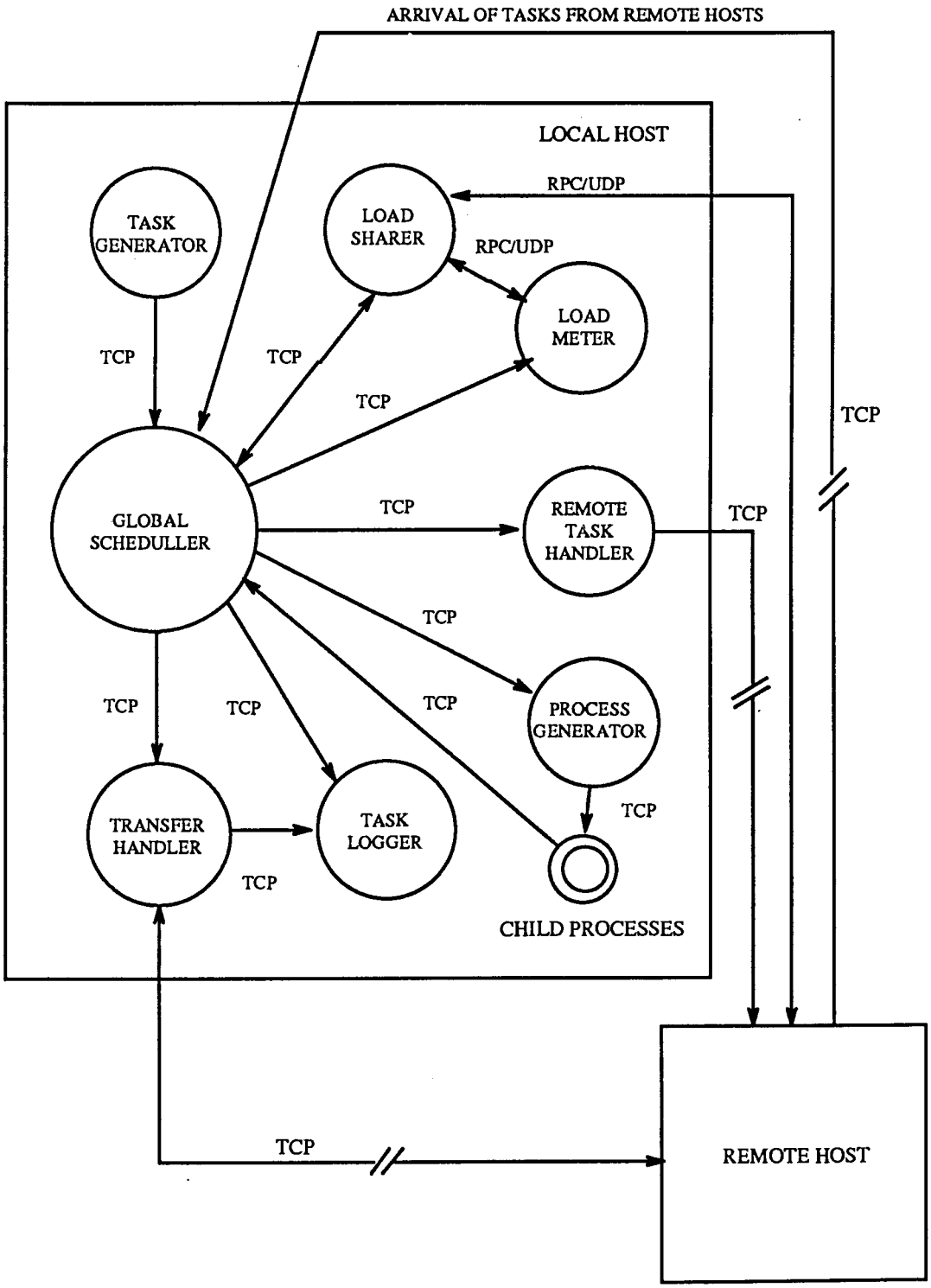


Figure 3.1 Software Architecture and Interprocess Communicatiuon

The *global scheduler* is the central entity in the collection of processes illustrated in figure 3.1. It receives tasks from the local *task generator* as well as from remote nodes. Since in our implementation a task is transferred only once, the global scheduler sends a remote task directly to the process generator for processing at the local processor. In case of a local task, it consults the load sharer, if necessary, and makes a scheduling decision. If the task is selected for transfer, it is handed to the *transfer handler* which establishes connection with the remote server and transfers the task; otherwise the task is sent to the *process generator*. On finishing execution the task returns back to the scheduler. If it is a local task, the scheduler sends it to the *task logger* for collecting statistics. If the completed task is from a remote node, it is sent to the *remote task handler*. The remote task handler establishes connection with the transfer handler of the node where the task originated and returns to it the completed task. A global scheduler, thus, acts as a coordinating switch through which all traffic flow.

As mentioned earlier, the connections established by the transfer and the remote task handlers are created and dissolved dynamically. The overheads associated with these connection establishments are included in the cost of a task transfer.

In the present implementation, we characterize the load of a node by the number of tasks currently being processed by the node. The load status is maintained by the *load meter*, and any change in the load status of the local node is communicated to the load meter by the global scheduler.

In order to select a server, a load sharer gathers load information from its own load meter as well as from those of the remote hosts. The communication between a load sharer and the load meters takes place by means of remote procedure calls.

The task generator precomputes the interarrival times and the service requirements of the tasks. During a trial, it sends the tasks to the global scheduler at the calculated intervals. The description of a task sent to the global scheduler consists of a task identifier, the time of departure

from the task generator, and the service requirement of the task.

A task gains access to the cpu (by which it is served) through the process generator. On receiving a task from the global scheduler, the process generator creates a child process, a copy of itself, which uses the cpu for the time interval determined by its service requirement. Since the process generator is designed as a very lightweight (or small) process, it can be duplicated with little overhead. The child process consumes cpu time by executing a loop whose size is specified by the the service requirement, obtains the record of its resource usage, and sends the task description back to the global scheduler on the connection inherited from the process generator.

The software architecture we have described so far applies strictly to the implementation of the threshold policy. Architectures used in the implementations of the broadcast and centralized policies are minor variations of the above architecture. The only difference lies in the absence of the load meter in the implementations of broadcast and centralized policies. In the broadcast policy, the global scheduler broadcasts any change in the load status, which is used by the load sharers in updating their system state vectors. In the centralized policy, any change in load status is also communicated directly to the central load sharer by the global scheduler. None of these policies, thus, needs a load meter.

CHAPTER 4

RESULTS

The three load sharing policies we have studied differ primarily in the amount of state information collected and maintained by each node in the system, in the type of communication used to collect this information, and in the control mechanism. We made the load sharing policies differ in these aspects quite deliberately. Our goal is to gain insight into how the performance of these policies under heavy load depends on the amount of state information maintained by the nodes, into how this information is collected and into the nature of communication.

Since our purpose is to study the policies under time-dependent inputs, it is useful to characterize the inputs in terms of task arrival rate, $\lambda(t)$, which is a function of time, t. As described in chapter 3, we modeled the task arrival process by a nonhomogeneous Poission process and generated the interarrival times, using $\lambda(t)$ as the rate function. In this chapter, whenever we refer to input, we mean this time-dependent rate function $\lambda(t)$.

The service requirements of the tasks are represented in terms of the number of times a task executes an empty loop. These service requirements are generated by exponential random variables with a mean of 700000. This, in our system, is equivalent to a mean task execution time of 5.67 seconds. In all the simulation runs (or trials) we have kept this service time distribution the same and varied the rate function under each policy. The workload for each of the eight nodes is, of course, generated using a different seed. We should note that a task, on the average, consumes a large amount of cpu time, and none of the tasks performs any input or output. The workload, thus, consists entirely of heavily cpu-bound tasks.

As for other parameters, we have used a threshold of 2 tasks in each of the policies, and a probe limit of 3 in the threshold policy. During each trial, each node has received 700 tasks over a period of approximately 2 hours. To minimize the effect of the initial and final transient phases, we have discarded the first and the last 30 tasks, so that our measurements are based on 640 tasks at each node for a total of 5120 tasks. Since the task arrival process for each node is generated using the same rate function, a particular node becoming heavily loaded implies that the rest of the system is also becoming heavily loaded.

As we are studying the performance of load sharing policies under time-dependent inputs, we are interested not only in the overall response of the system, but also in the response in different segments of time during a trial. For this purpose we have divided the time of a trial into several subintervals (of 5 minutes each) and measured various quantities related to the tasks arriving at the system during these time intervals. These measurements give us snapshots of the system's performance at different instances during a trial.

We shall discuss the performance of the system mainly in terms of the *response time* of a task, which is the time interval between the arrival of a task and the end of its execution. In particular, we shall express all results in terms of two types of averages of the response times. One is the *mean response time*, which is the sum of the response times of all the tasks processed in the system divided by the number of tasks. The second type of average is called the *mean running response time* and is defined as the sum of the response times of the tasks arriving at the system within a particular segment of time divided by the number of such tasks. We shall also describe some of the results in terms of other quantities such as the mean task transfer rate and the mean time spent in making a global scheduling decisions during a particular time interval.

In all our trials, the mean response times of the system under each of the load sharing policies are found to be better than the corresponding times obtained when there is no load sharing. In other words, we have found all three strategies to be effective, and as such we shall not discuss

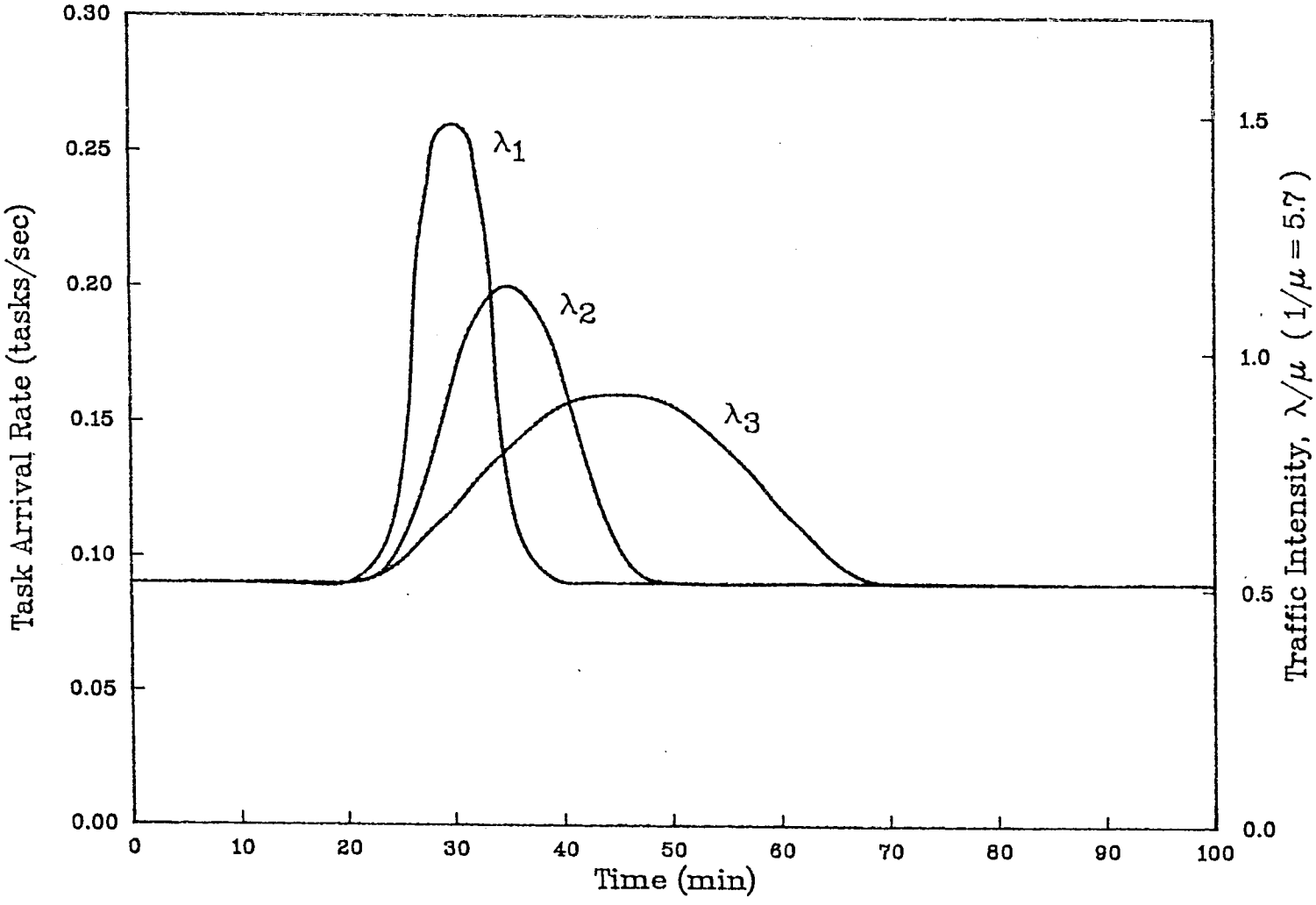
the performance of these policies relative to no load sharing. Rather, we shall compare their performances relative to one another.

We begin by describing the performance of the system under the three inputs shown in figure 4.1. (In subsequent figures we refer to them as *Input 1*, *Input 2* and *Input 3*). Each of the rate functions $\lambda_i(t)$, $i = 1, 2, 3$, corresponding to Input 1, Input 2 and Input 3 respectively, has a peak superimposed on a constant background. This type of input can be used to model time-of-the-day effect, a sudden burst of arrivals, etc., and can form the building block for constructing models to represent more general types of time-dependent arrival processes such as periodic bursts of arrivals, short-term fluctuations, etc. Thus a performance study of the load sharing policies under such input should enable us to understand how these policies would perform under a more general input.

We shall consider the rate function $\lambda_1(t)$ first. During the time intervals between 0 and 20 minutes and 40 and 120 minutes tasks enter the system at a constant rate of 0.09 tasks/second which corresponds to a *traffic intensity* (= arrival rate/service rate), $\rho = 0.5$. The arrival rate increases very rapidly between 21 and 30 minutes reaching a peak of 0.26 (traffic intensity ~ 1.5), and then falls rapidly to the background level in the next 10 minutes. Thus a very high traffic intensity is sustained for 5 minutes and a moderate one for about 5 minutes. If the same number of tasks were arriving at a constant rate during the 2 hour period, the corresponding arrival rate would be approximately 0.1 tasks/second with a traffic intensity of 0.6, and the interarrival times would be distributed according to a Poisson distribution. We shall denote the rate function of this Poisson distributio by λ_o.

Figure 4.1

Input: Task Arrival Rates, $\lambda_i(t)$



4.1. Performance of Threshold Policy

In figure 4.2 we show the performance of the system under the threshold load sharing policy. We compare the response time for $\lambda_1(t)$ with that for the constant rate λ_o We have divided the entire time of trial into 5 minute subintervals and plotted the system-wide mean running response time (the mean response time of the task arriving at the system within a particular time interval averaged over eight nodes) as a function of these subintervals of time.

The mean running response time of the system for the rate $\lambda_1(t)$ during the first 20 minutes and during the interval between the 50th minute and the end of the trial, is very similar to the response time for the constant rate λ_o. The response time, however, has a pronounced peak which corresponds very closely to the peak in the rate function $\lambda_1(t)$. We should note that, although the peak in the rate function falls to the background level around the 40th minute, the response time does not reach the value corresponding to λ_o until the 50th minute. Thus there is a time lag of roughly 10 minutes. In other words, the system under threshold policy with the given set of parameters takes about 10 minutes to settle back to equilibrium.

Two factors seems to contribute to this sharp increase in the response time. The first is that each of the eight processors is overloaded, and the processes running the tasks are spending more time being context switched than doing any useful work. The second reason is as follows. Since the local processor is busy, the load sharer has to communicate with other nodes in search for a less busy node. But the rest of the processors are also busy, so that the number of probing done by the load sharer more often than not exceeds the probe limit. In an overloaded system this *cost of load sharing* (the time interval between a task's arrival and its being scheduled to a remote or the local node) can be very large. To separate these two contributing factors we have measured the cost of load sharing during each 5 minute interval, and plotted the results in figure 4.3 which shows the system-wide mean running cost of load sharing as a function of time. It is clear that the cost peaks during the time when the arrival rate is very large and the system is very busy.

Figure 4.2

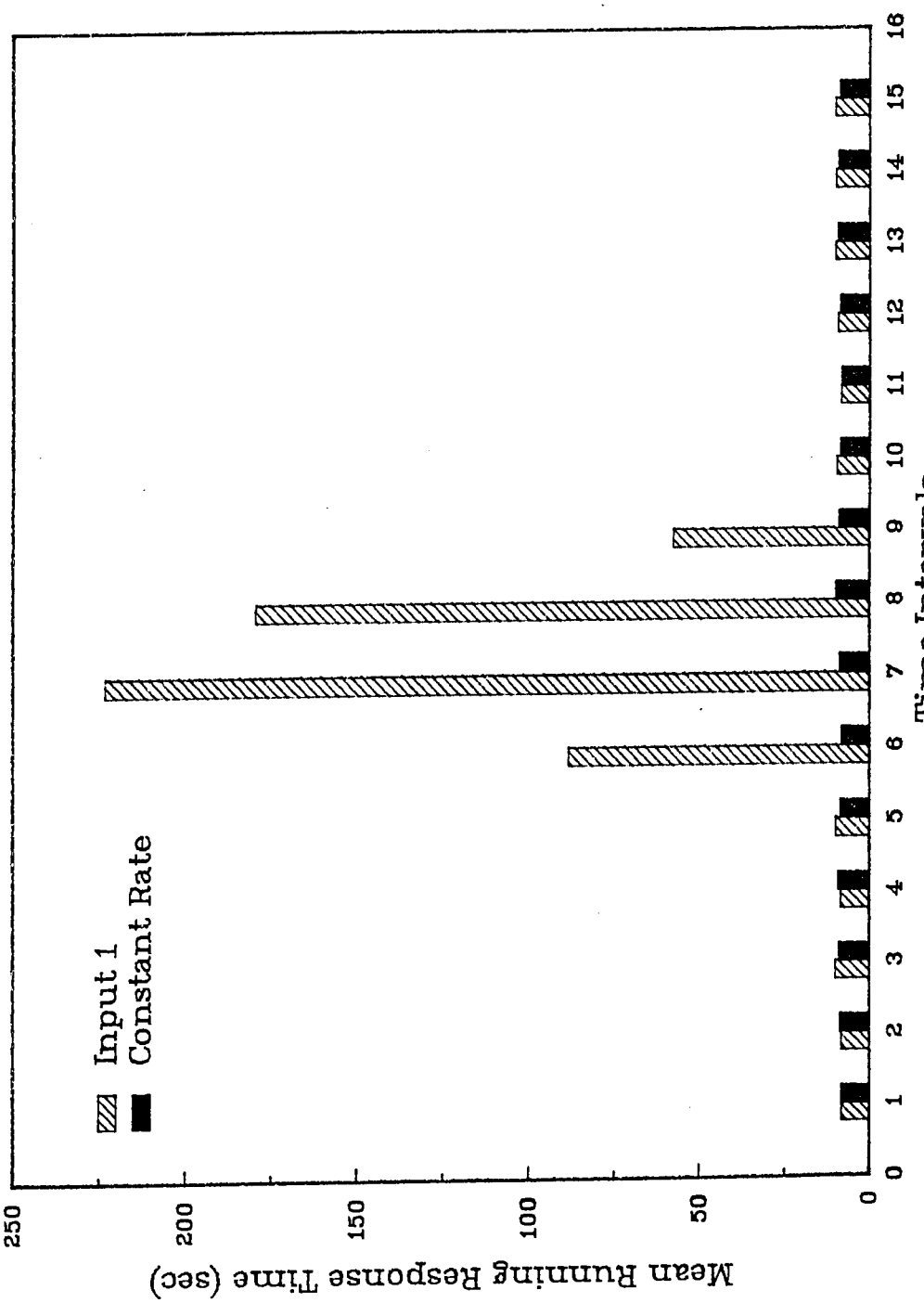
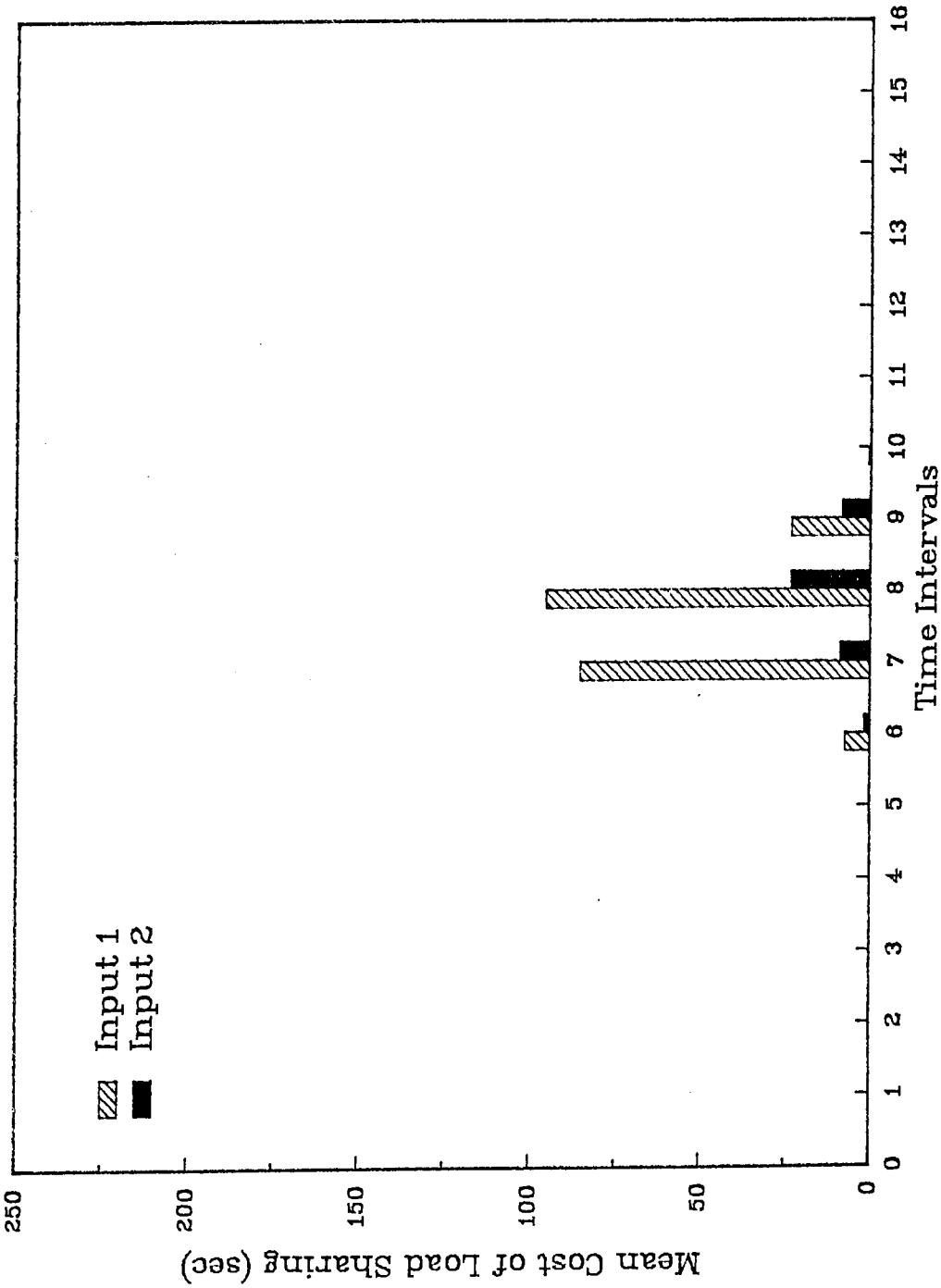

Threshold Policy: Running Response Time
Input 1
Constant Rate
Mean Running Response Time (sec)
250
200
150
100
50
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Time Intervals

Figure 4.3

Threshold Policy: Cost of Load Sharing

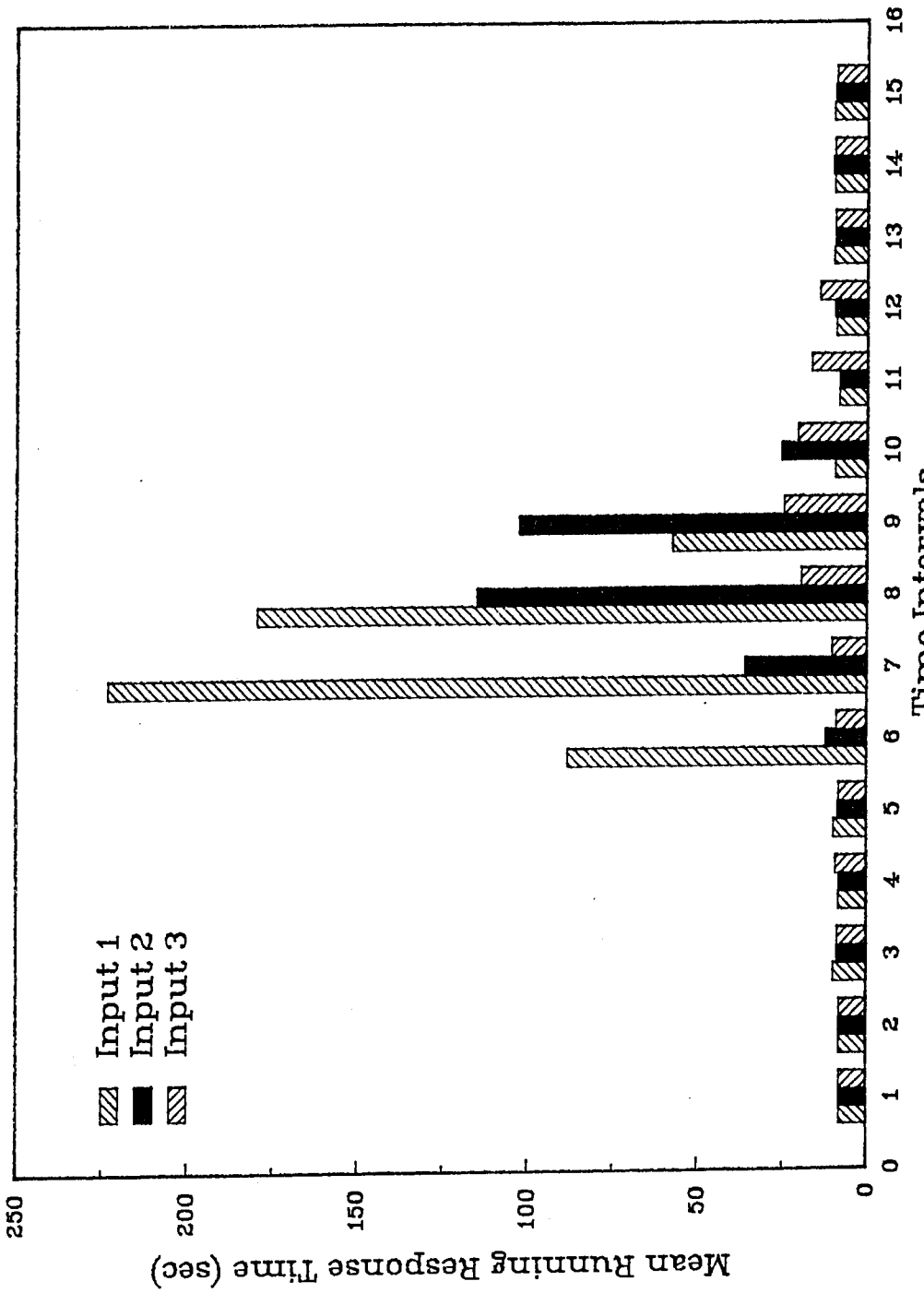


In order to understand the dependence of these results on the size and shape of the peak in the rate function, we measured the performance of the threshold policy under two more inputs, $\lambda_2(t)$ and $\lambda_3(t)$. In these inputs, we have reduced the height of the peak and increased its width in a way so that the total number of tasks entering the system under all three inputs are approximately the same. We have plotted the system-wide mean running response times under these inputs in figure 4.4. Even though the peak in $\lambda_2(t)$ still corresponds to a traffic intensity greater than 1, and it stays above 1 for a larger period of time than for $\lambda_1(t)$, the system response times during the time intervals corresponding to the peak show significant improvements. This is even more noticeable in the reduction of the costs of load sharing (which we show in figure 4.3).

This trend continues when in $\lambda_3(t)$ we reduce the height of the peak even further. These results indicate that given a fixed time interval, the performance of a system under the threshold policy is more sensitive to the rate of task arrival than to the total number of tasks entering the system during this time interval (2 hours in the present case).

Figure 4.4

Threshold Policy: Running Response Time



4.2. Performance of Broadcast and Centralized Policies

In figures 4.5 and 4.6 we have plotted the mean running response times of the system under the broadcast and centralized policies respectively. An examination of these results indicates that the system performances under these two policies are very similar. This is perhaps not surprising. For, these two policies use the same criteria to decide the eligibility of a task for transfer, and also maintains the same state information and use it in the same way to find a server for the task selected for transfer. The only difference between them is that the control in the broadcast policy is distributed, while in the centralized policy it is centralized. As their performances are similar, we have chosen to discuss the response of a system under these policies collectively.

The general behaviour of the mean running response time (as a function of time) under the broadcast and the centralized policies is similar to that under the threshold policy. In presence of a peak in the rate function, the response time becomes progressively worse as the peak is made higher and narrower (figures 4.7 and 4.8). Also, the times taken by the system to settle back to the level of constant arrival rate are very similar, being about 10 minutes.

With the broadcast and the centralized policies, the mean running response times during the intervals corresponding to the peak, however, are significantly (about 30%) smaller than is the case with the threshold policy. The mean response times under these policies are also 30% smaller than the mean response time under the threshold policy. In table 4.1 we have shown the mean response times under all three policies for different inputs including the case of constant rate of task arrival and compared them with the corresponding mean response times obtained when there is no load sharing. It is clear that for each input, the mean response time for the system under each of the three load sharing policies is smaller than the mean response time for the case of no load sharing. That is, these policies are found to be effective (under the conditions tested) even when the inputs contain sharp short-term fluctuations, and thus, they meet one of the requirements of stability discussed in chapter 1.

Figure 4.5

Broadcast Policy: Running Response Time

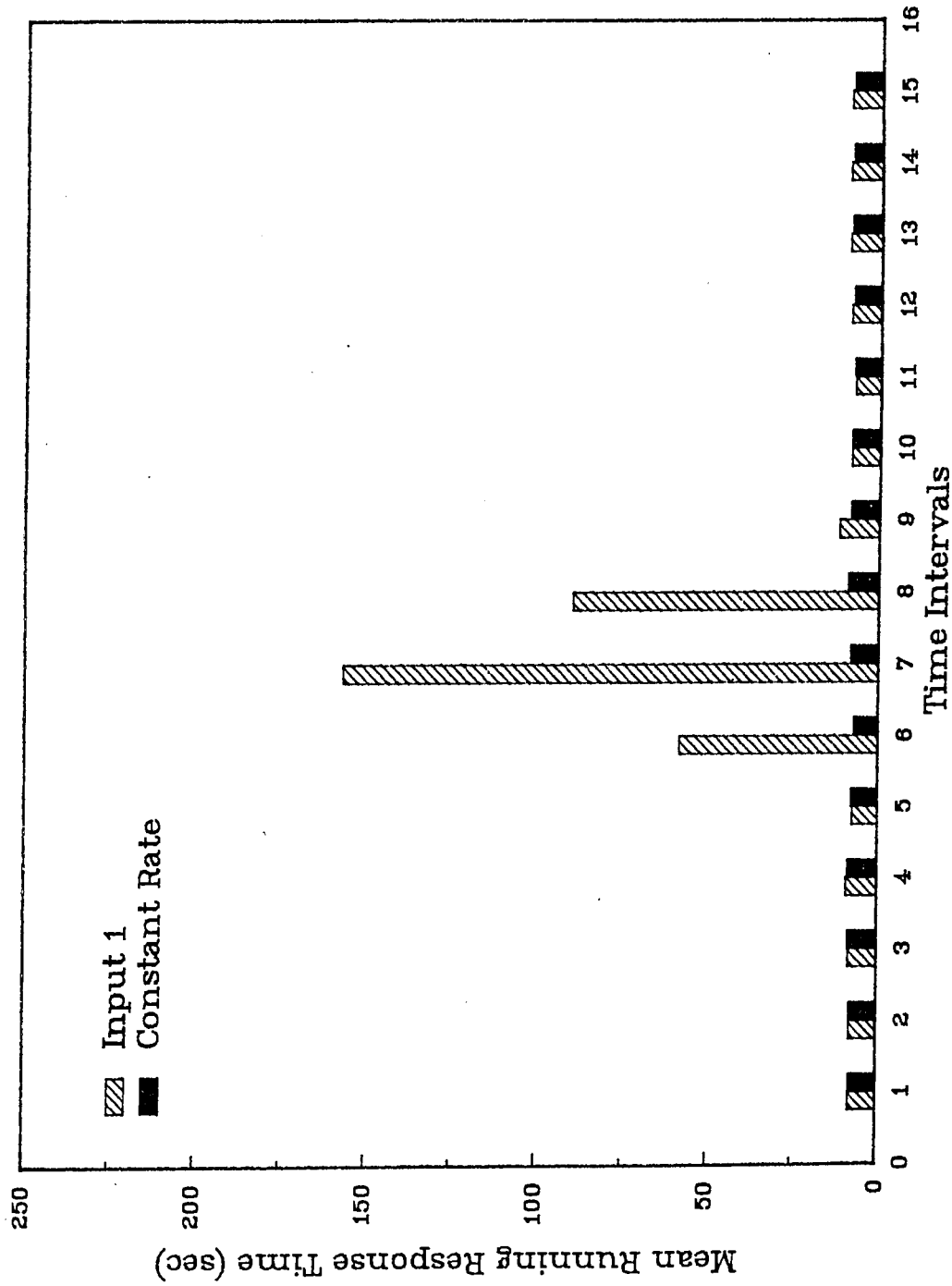
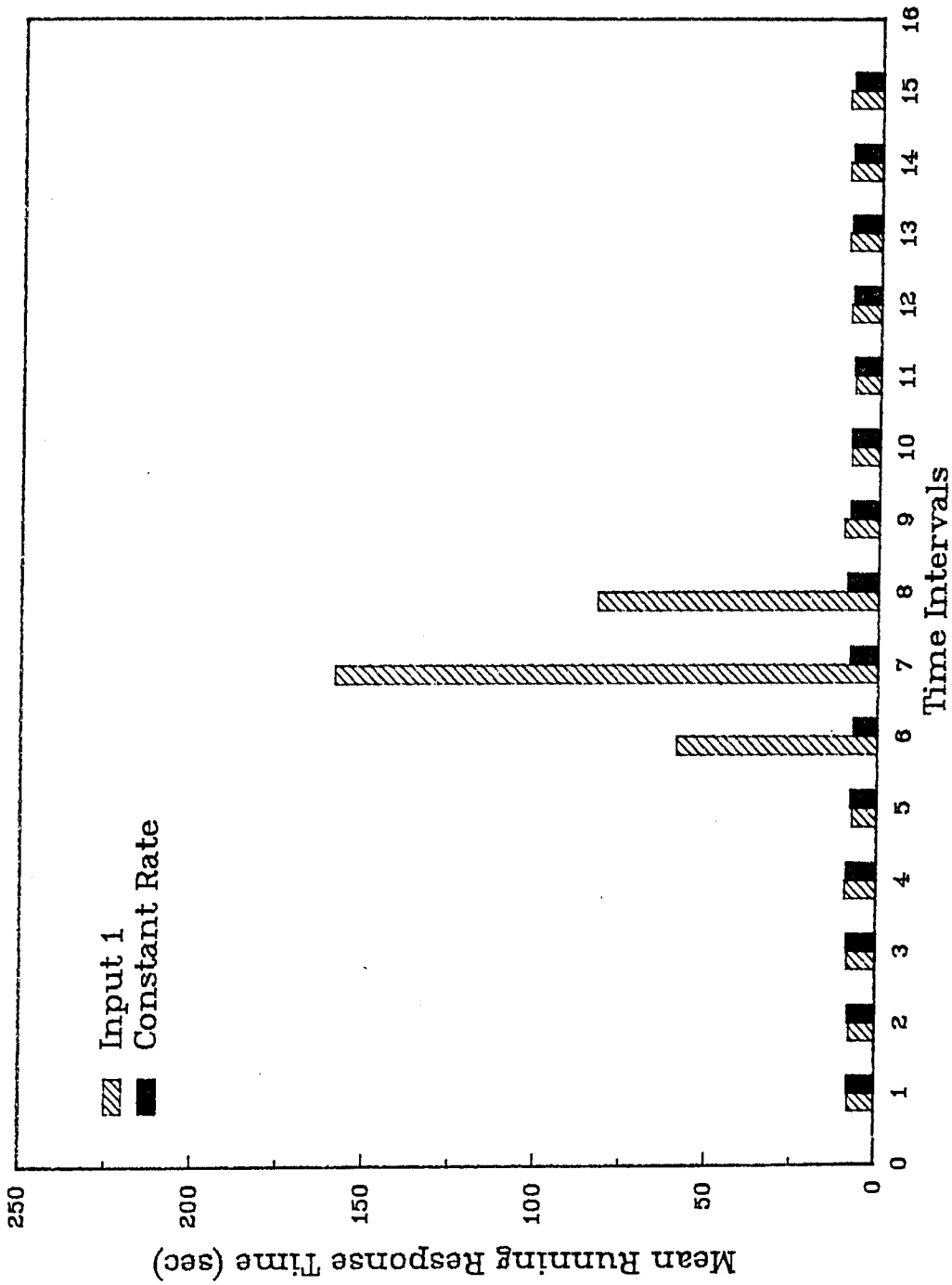


Figure 4.6


Centralized Policy: Running Response Time
Input 1
Constant Rate
Mean Running Response Time (sec)
250
200
150
100
50
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Time Intervals

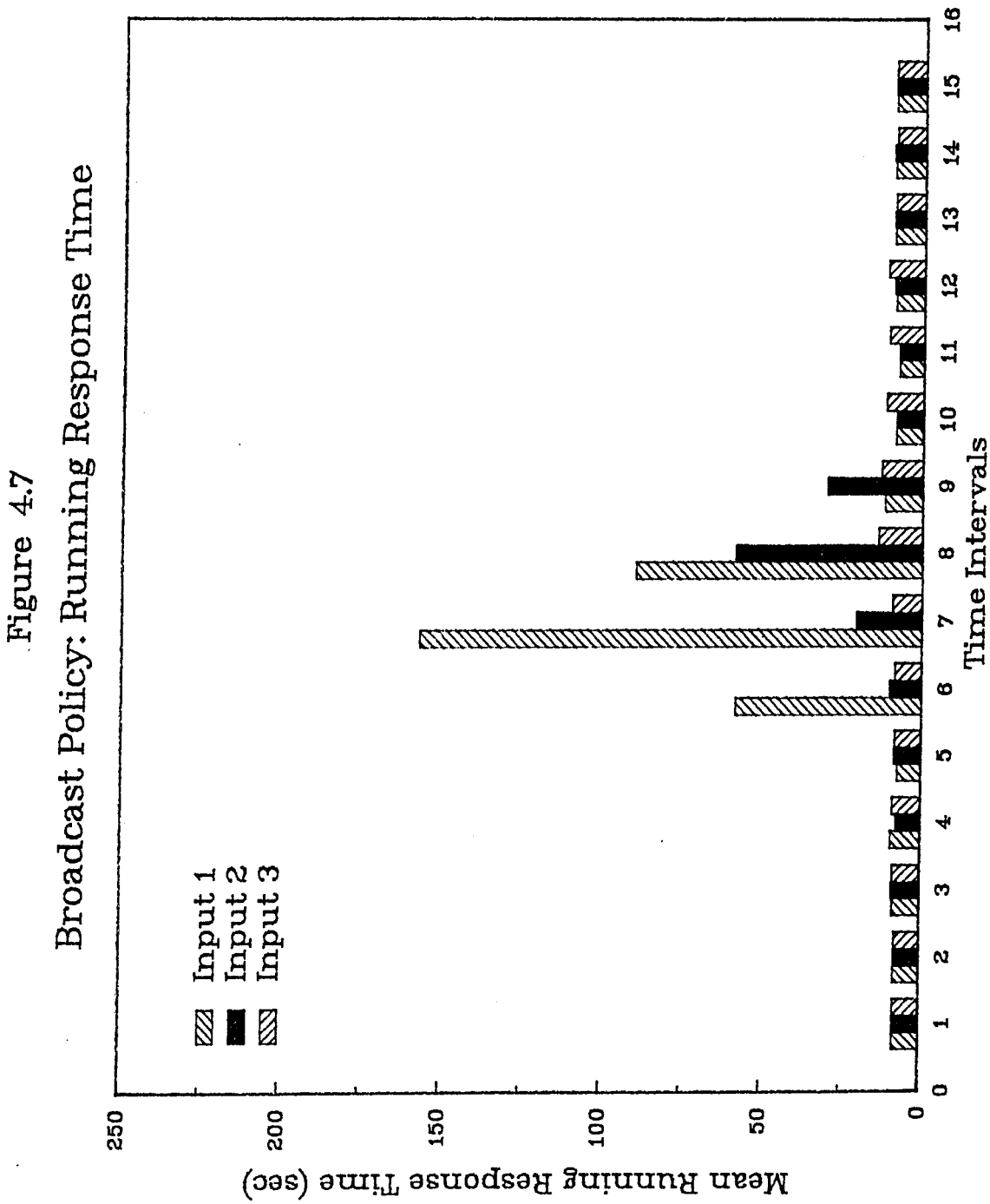
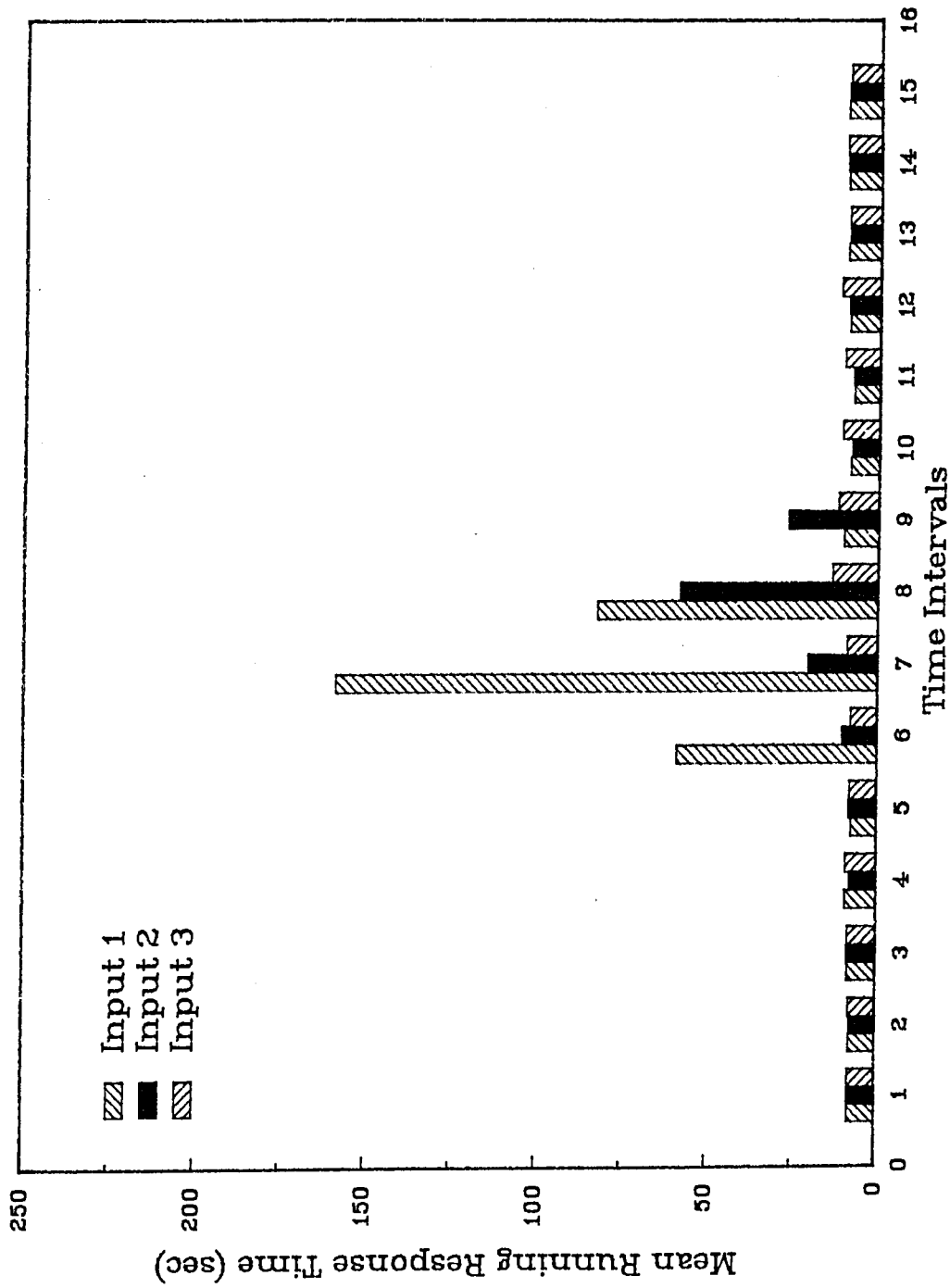

Broadcast Policy: Running Response Time
Input 1
Input 2
Input 3
Mean Running Response Time (sec)
250
200
150
100
50
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Time Intervals

Figure 4.7

Figure 4.8

Centralized Policy: Running Response Time



Policy	Input 1 (λ_1)	Input 2 (λ_2)	Input 3 (λ_3)	Constant Rate (λ_0)
Threshold	42.94	21.93	12.70	8.46
Broadcast	29.31	15.22	9.85	8.15
Centralized	29.09	14.91	10.06	8.14
No load sharing	52.53	27.74	21.26	15.23

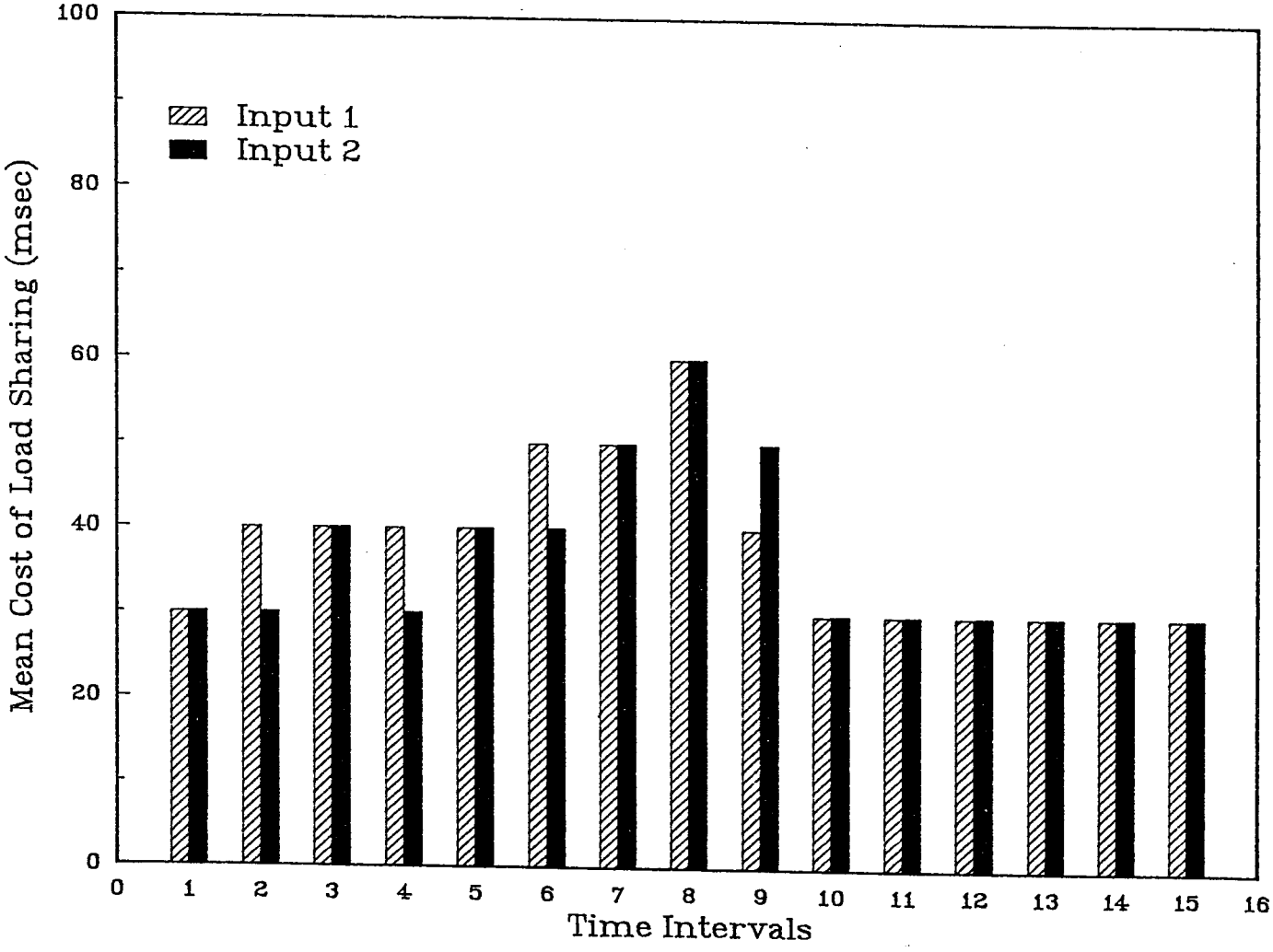
Table 4.1. Mean response times (sec) under different load sharing policies with various inputs

One of the more interesting aspects of the system performance under the broadcast and the centralized policies is that the presence of a peak in the rate function has very little effect on the costs of load sharing (figures 4.9 and 4.10). The reasons for this are quite simple. Recall that under the broadcast policy, the nodes try to maintain a global view of the system load by broadcasting the changes in load status with respect to the threshold. However, we should stress the fact that the nodes do not maintain the exact loads of different nodes. They effectively maintain a busy/idle status record. When a node continues to be busy because of task arrivals, it does not broadcast any message at all. Thus, when the entire system becomes heavily loaded, it generates very few broadcast messages.

Similarly, in the centralized policy the nodes help the central load sharer maintain a global view of the busy/idle status of various nodes in the system by communicating any change in the load status to the load sharer. A node, however, communicates with load sharer only if its load goes above or below the threshold. If a node is continuously busy or idle, it does not communicate with the load sharer at all. Thus by choosing appropriate nature of information, the broadcast and the centralized policies avoid the cost of collecting state information when the system is heavily loaded. Also, the nodes do not exchange any information in order to make scheduling decisions, further reducing the cost of load sharing.

Figure 4.9

Broadcast Policy: Cost of Load Sharing



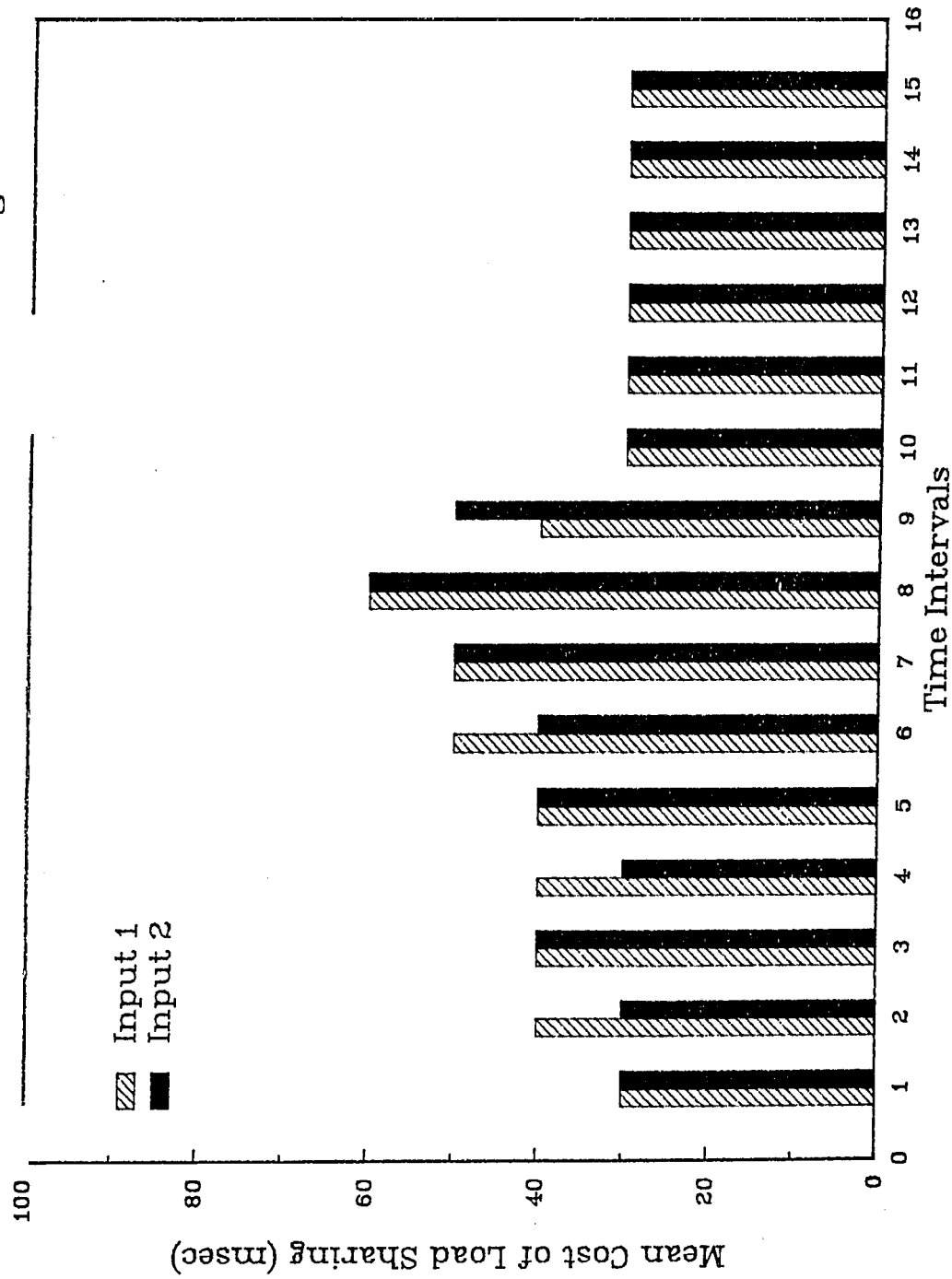


Figure 4.10
Centralized Policy: Cost of Load Sharing

The broadcast policy, however, has a potential weakness. In a simulation study, Livney and Melman [Livney & Melman 82] found that when the number of nodes in the network exceeds a critical value, a broadcast based policy performs poorly due to excessive communication overheads. To verify this for the present broadcast policy, we measured the response times for a system of 4 nodes in the network and compared them with the values corresponding to 8 nodes. Although the system with 8 nodes incurs somewhat larger overhead than the system with 4 nodes, this is more than compensated by the improvement in response time due to the availability of the additional four processors. We should, however, note that in Livney and Melman's simulation the critical number of nodes above which the performance starts to degrade is 20, and we probably should not expect to find evidence of this effect in an eight-node network.

One of the primary reasons for studying a centralized policy is to understand the effect of control mechanism on the performance. A policy with centralized control can potentially become a source of bottleneck. In the present study we have not detected any such sign. This, however, has more to do with the physical limitations of having only eight processors than any inherent characteristic of the policy. Earlier we have mentioned that the central load sharer is made to run on a ninth machine. When the load sharer is made to run on one of the eight machines involved in task processing, its performance degrades substantially. The performance of the node running the central load sharer also suffers, leading to a general system-wide degradation of performance.

4.3. Relative Performance of Threshold and Broadcast Policies

In figure 4.11 we have compared the mean running response times for $\lambda_1(t)$ of the system under the threshold and the broadcast policies. It is clear that when the task arrival rate has a sharp peak the broadcast policy performs significantly (about 30%) better than the threshold policy. There are two reasons for this. The broadcast policy maintains a global view of the system load and as a result is better capable than the threshold policy of recognizing the changing status of the system load and adapting to it. This becomes evident when we compare the mean running task transfer rates under the two policies for $\lambda_1(t)$ (figure 4.12). When the arrival rate increases sharply in the interval between 25 and 30 minutes, the broadcast policy transfers about 24% of tasks. The threshold policy, by contrast, transfers only 11%. We observe similar behaviour when the arrival rate begins to fall. Once again the broadcast policy recognizes the situation more effectively than the threshold policy and transfers a larger percentage of the tasks to remote nodes.

Although the difference in the transfer rates is a factor in the superior performance of the broadcast policy to that of the threshold policy, the major contribution comes from the difference in the cost incurred in making load sharing decisions in the two policies. A comparison of the costs during the intervals corresponding to the peak shows that substantial portion of the performance difference results from the difference in the costs of load sharing.

Figure 4.11

Comparison: Threshold & Broadcast Policies

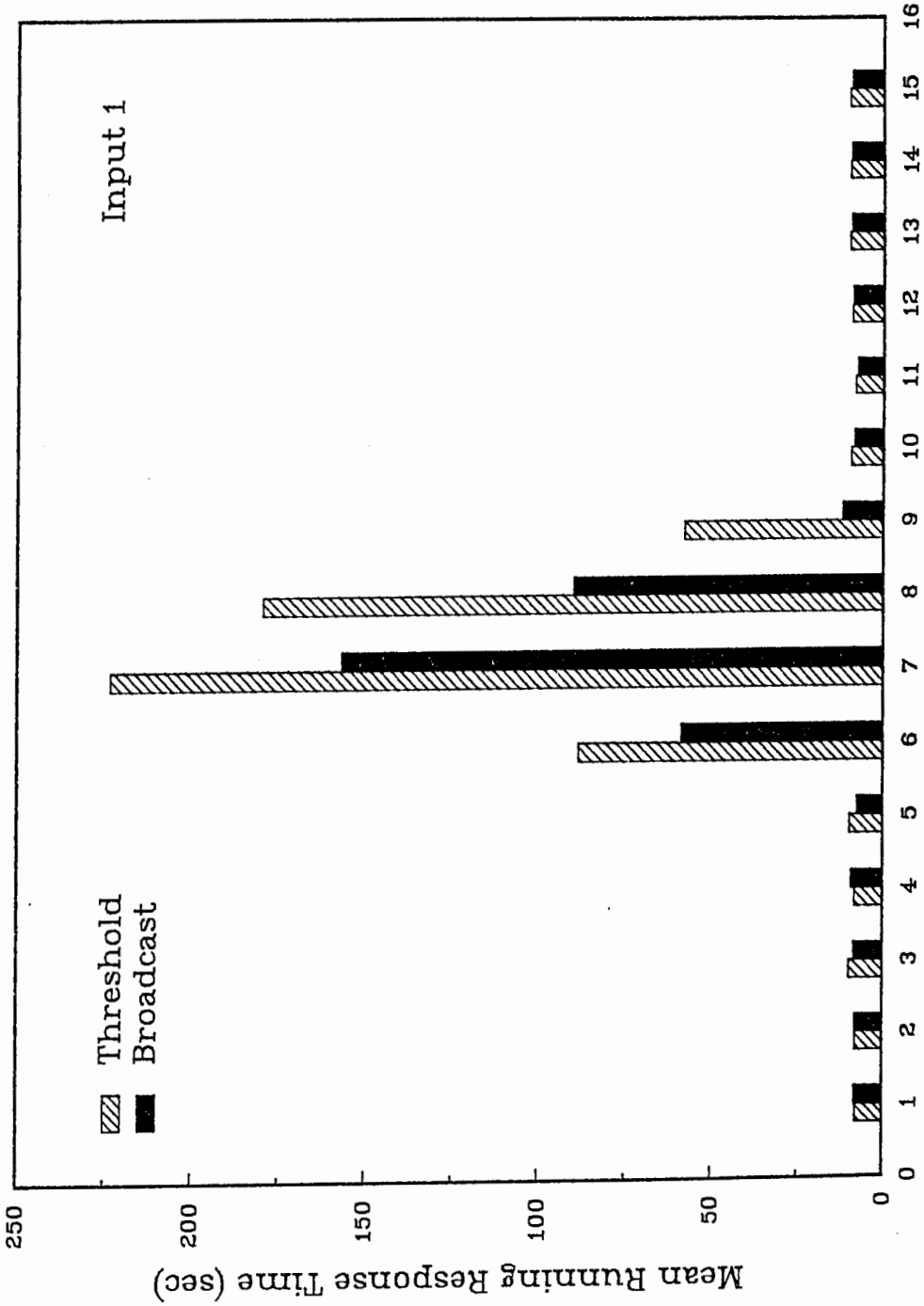
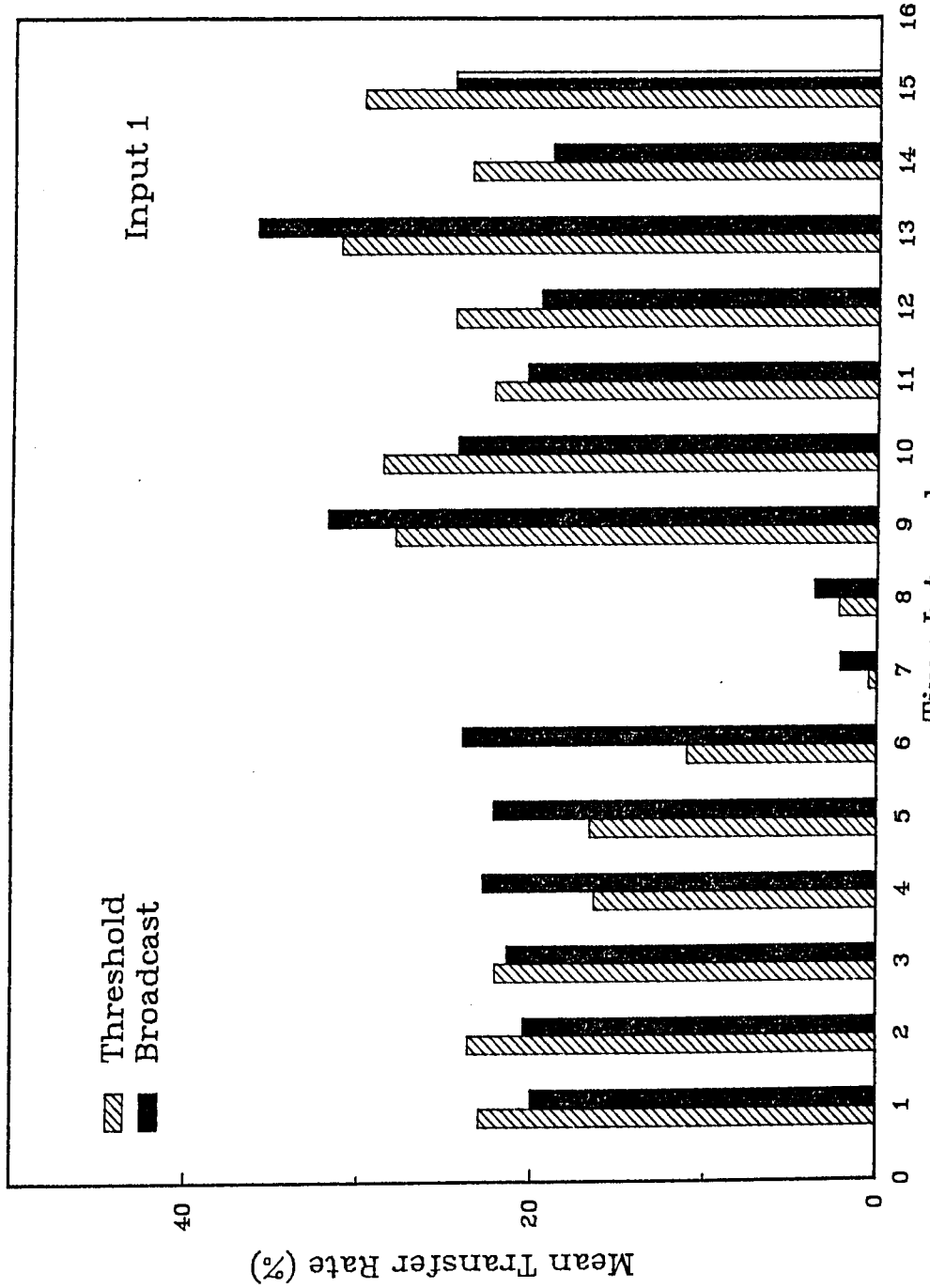


Figure 4.12

Transfer Rates in Threshold & Broadcast Policies



4.4. Probe Limit and Performance of Threshold Policy

Earlier we have mentioned that the increased number of probing done by a node under the threshold policy causes a significant degradation of performance in a heavily loaded system. Hence, a smaller probe limit should improve the response time of the system during the time intervals when the system becomes overloaded. To verify this claim, we have measured the cost of load sharing under the threshold policy with a probe limit of one. We show the results in figure 4.13, where we compare them with the costs obtained for the case of probe limit equal to three. It is clear that a lower probe limit results in a significant improvement in the response time during the busy intervals. The lower probe limit, however, causes an increase in the response times for the intervals when the system is lightly loaded. Because, with a low probe limit, the threshold policy may not always locate a node whose load is below the threshold even when the probability of such a node's existence is large.

The above discussion naturally leads to the question: can the overall performance be improved by modifying the threshold policy so that it would not attempt to locate a lightly loaded node when the system is busy? For the very special case, in which all the nodes become heavily loaded at the same time, we have modified the threshold policy and measured the system response time for input $\lambda_1(t)$. We have artificially prevented the threshold policy from exchanging state information and from attempting to share load during the time interval between 25 and 40 minutes. That is, we have reduced the probe limit during this time interval to 0. As figure 4.14 shows, this results in a large (about 25%) improvement in the mean response time of the system during the time interval corresponding to the peak in $\lambda_1(t)$. (From now on, we shall call the time interval corresponding to the peak in the rate function λ_i the *peak period*). Lowering the probe limit to 1 during the peak period also leads to an improvement in the response time, although by an smaller amount than obtained with probe limit equal to zero. The corresponding measurements for the input $\lambda_2(t)$ yields similar results.

Figure 4.13

Threshold Policy: Cost of Load Sharing

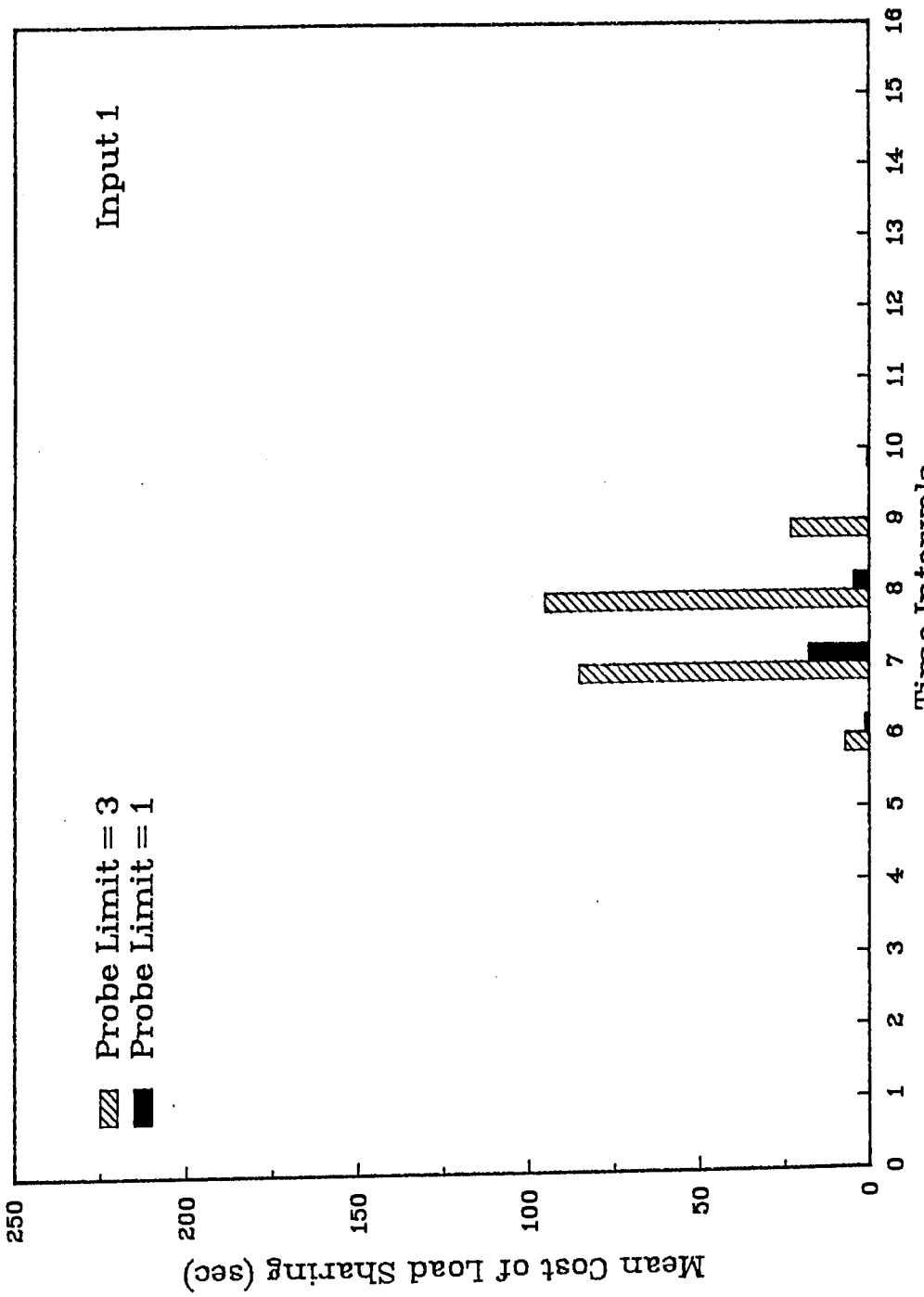
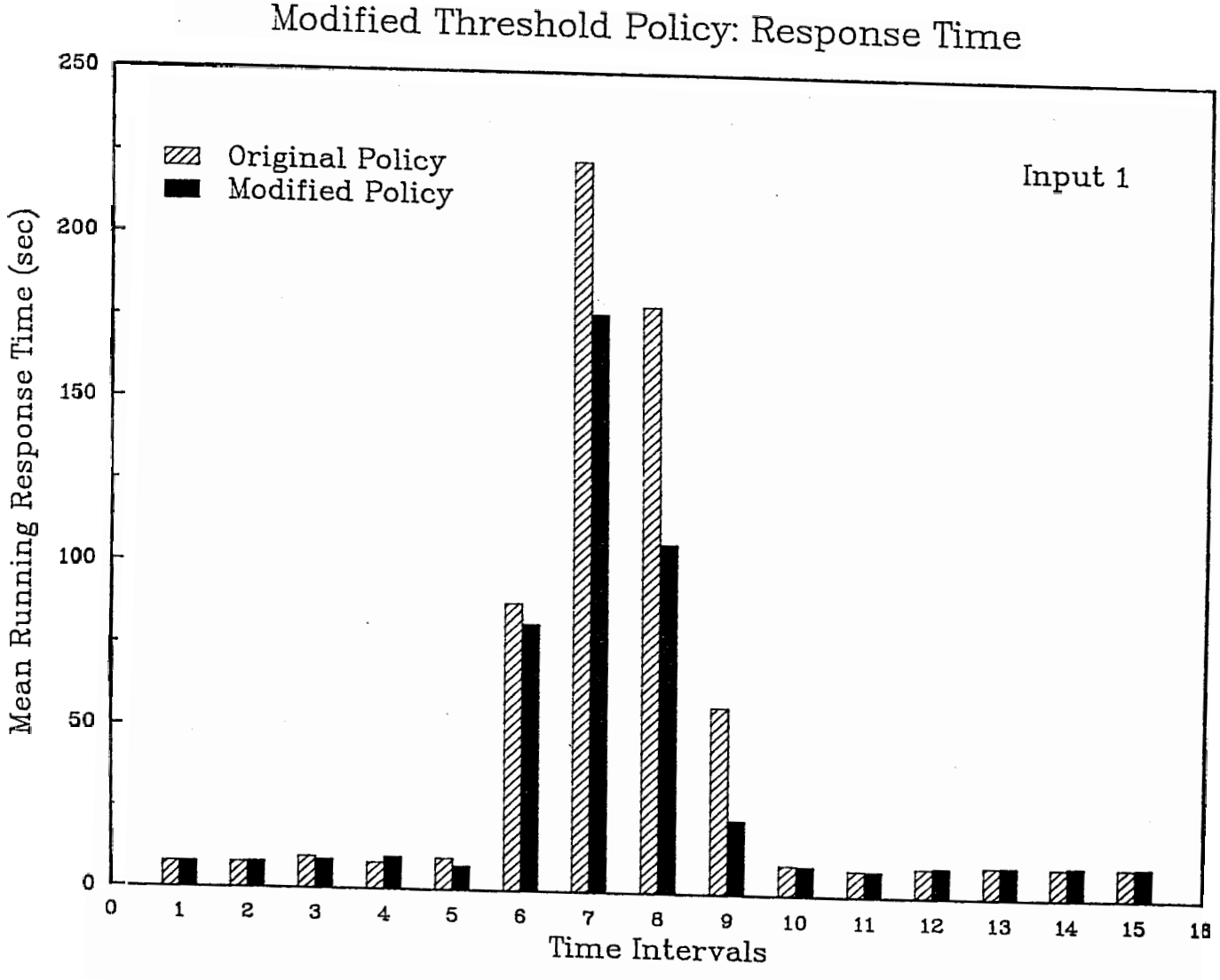


Figure 4.14


Modified Threshold Policy: Response Time
Original Policy
Modified Policy
Input 1
Mean Running Response Time (sec)
250
200
150
100
50
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Time Intervals

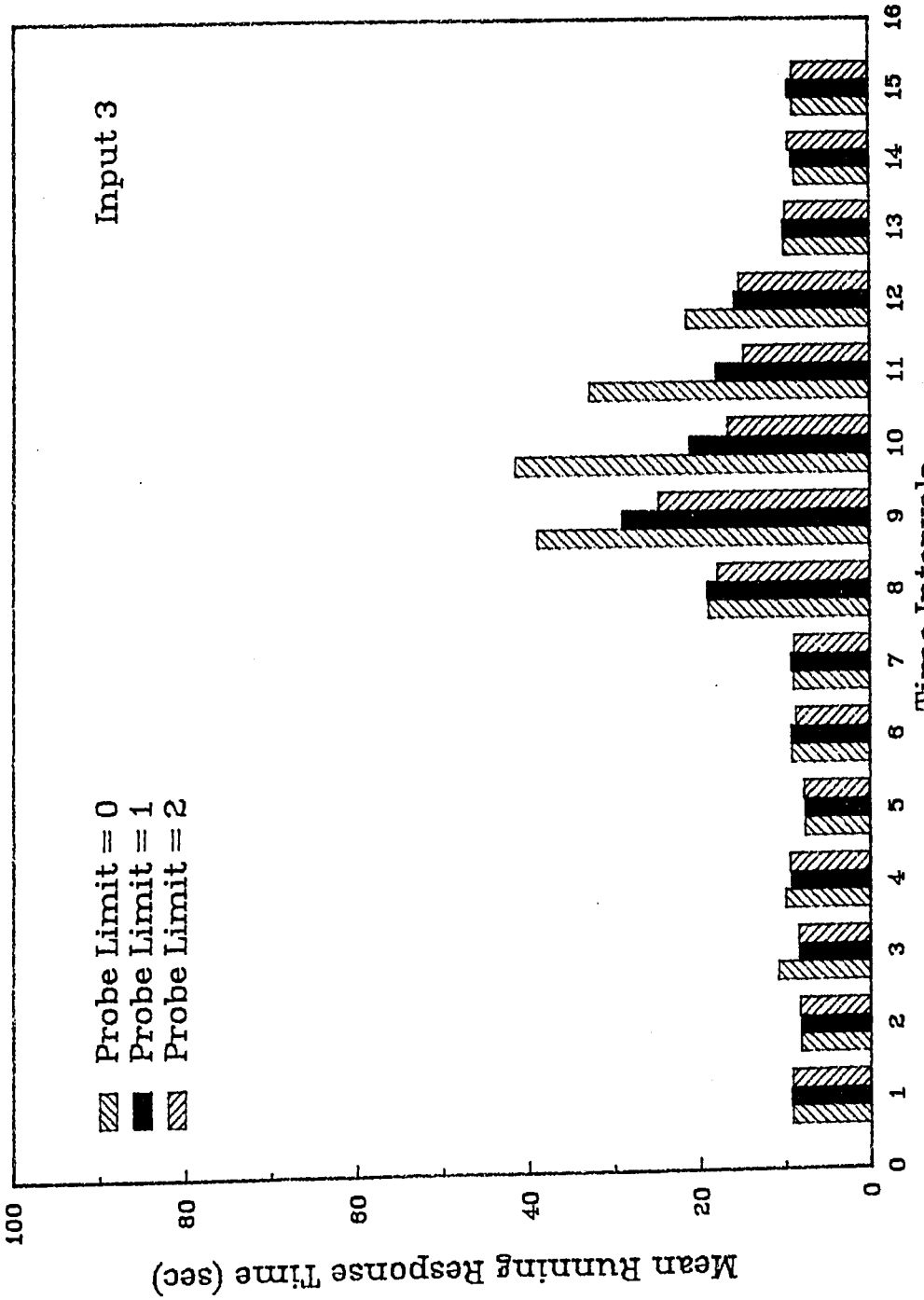
The result that the response time under the threshold policy can be improved by not attempting to share load during peak periods applies strictly to the case when the system is *very* busy (as in the cases with inputs $\lambda_1(t)$ and $\lambda_2(t)$). But, when the system is moderately busy (e.g. under input $\lambda_3(t)$), decreasing the probe limit to 0 during peak period has the opposite effect. In figure 4.15 we have plotted the mean running response times for input $\lambda_3(t)$ corresponding to different values of the probe limit during the peak period. Clearly, for $\lambda_3(t)$ the modified threshold policy with a probe limit of 2 performs better than the corresponding policy with a probe limit equal to 0. Also, the mean response time (for the 2 hour interval) for the probe limit equal to 2 is superior to those for probe limits equal to 0, 1 and 3 (see table 4.2). A probe limit of 0 during the time intervals corresponding to the peak in $\lambda_3(t)$ results in an increase in the response time.

Probe Limit	Mean Response Time (sec)
0	16.40
1	12.90
2	11.95
3	12.70

Table 4.2. Response time for different probe limits for λ_3.

We can explain this increase in the response time as follows. Under the threshold policy (with a probe limit of 3 and a threshold of 2 tasks) the task transfer rate increases monotonously with increasing traffic intensity ρ before it reaches a maximum of 36% around $\rho = 0.86$ and then falls off rapidly [Speakman 86]. Thus, when the system is moderately busy, there is a finite probability that at any instant some of the nodes will be lightly loaded and the heavily loaded nodes can improve performance by transferring part of their workload to these lightly loaded nodes. By setting the probe limit to 0 during the peak period, we prevent the nodes from transferring any task. In case of $\lambda_3(t)$ the peak corresponds to a traffic intensity $\rho \sim 0.9$, and the task transfer rate for this intensity is approximately equal to 25%. Thus, by reducing the probe limit to 2, we are allowing the system to redistribute enough tasks to achieve an improvement in the response time.

Figure 4.15: Modified Threshold Policy
Response Time & Probe Limit



These results clearly indicate that if the threshold policy is modified so that it can gauge the busyness of a system and adjust its probing accordingly, a significant improvement in performance can be obtained.

CHAPTER 5

CONCLUSIONS AND DISCUSSIONS

In this thesis we set out to evaluate the relative performance of three simple load sharing policies with respect to selected stability issues, by measuring the response of a system using the load sharing policies under heavy and fluctuating load. In particular we wanted to determine how the performance of a system depends on the nature and the amount of system state information maintained by the nodes of the system, on the environment in which the load sharing policies are used, and on the nature of communication. Our purpose was to identify the characteristics of these policies, the understanding of which should facilitate the design of stable policies.

To this end, we implemented the three load sharing policies in a network of eight processors running under UNIX, and measured the system performance under a set of time dependent inputs in which the rate of task arrival had very sharp peaks. The results demonstrate that these load sharing policies are capable of improving the overall performance of a system even when they are subjected to severe inputs, thus indicating their resilience. Although the performance degrade considerably when the system becomes heavily loaded for a finite time, all three policies are found (under the conditions tested) to be capable of restoring the system to equilibrium without taking an unduly long time after the surge has ceased to exist.

The degree of degradation that the system suffers depends partly on the quantity of state information maintained by the nodes. It is generally expected that the more state information a node has, the better capable it becomes of making good scheduling decisions. This is borne out by the better performance of a system under the broadcast and the centralized policy compared to that under the threshold policy. Under a heavy load the broadcast and the centralized policies are

better equipped to quickly find an lightly loaded node than the threshold policy.

But the extent of degradation depends much more strongly on how and when the nodes in a system collect the state information. To deliver a reasonable performance, it is very important that a load sharing policy does not allow the nodes to collect or exchange state information when the system as a whole experiences a surge of heavy load.

This is clearly demonstrated by the difference between the performance of a system under the threshold policy and that under the broadcast policy. To find a server, the threshold policy needs to query remote nodes even when all the nodes are busy and the probability of finding a lightly loaded node is very small. This leads to a substantial degradation in the performance. By not allowing a node (under the threshold policy) to exchange state information during a burst of arrivals, we were able to reduce the gap between the performances of a system under these two load sharing policies.

Next we address the question: how should the nodes in a system communicate? That is, should they use broadcast or point-to-point communication? Given our result that the overall mean response times under the threshold and the broadcast policies differ by less than 5% when the system is lightly or moderately loaded, the nature of communication may not be a critical factor in designing a load sharing policy, specially if we restrict the system not to exchange state information during a very busy period.

The choice of one mode of communication over the other, however, is very implementation dependent. One may argue that in an environment in which broadcast is carried out more efficiently than in UNIX, one may obtain better overall performance. But, if the actual communication overhead is only a very small percentage of the total service time, it is unlikely that a more efficient broadcast mechanism will lead to significantly different results. For very small tasks, on the other hand, the service time can be comparable to the communication overhead per task. If the workload consists primarily of such small tasks, a more efficient broadcast mechanism may

have a large effect on the performance. The cost of transferring such a task, however, would exceed its service time, and it would be expedient to process it locally.

Based on the result of this experiment, it is tempting to conjecture that either the broadcast or the centralized policy would be a better choice than the threshold policy for load sharing. But, with these policies we have to confront other issues which make them the less attractive choices. In a large network the broadcast policy may breakdown due to excessive communication overheads and the centralized load sharer may form a bottleneck. That we have not detected any sign of these effects because of the smallness of our network suggests an alternative. We can partition the nodes of a network into several groups, each of which is small enough that the communication overhead would remain reasonable, yet large enough to achieve good performance through load sharing. Then, instead of broadcasting a load status change to all the nodes in the entire system, a node can use selective multicast to communicate with the nodes in the group to which it belongs and share load with them.

Similarly, in the centralized policy, the single load sharer may be replaced by several load sharers, one for each group of nodes. Whether or not any performance benefit will result from the modified centralized policy is not intuitively clear. In order to achieve acceptable performance, we may need to run the load sharers on a separate processor dedicated solely for this purpose. Otherwise, running them on the processors which also process regular tasks may lead to unacceptable degradation.

Robustness is another issue we need to confront when choosing one policy over another. Of the three policies, the centralized one is the most vulnerable to a node crash, specially if the node on which the load sharer is running fails. Using a backup load sharer seems to be the only solution to this problem.

In case of the broadcast policy, the failure of a node may become a problem depending on the load status of the node prior to the crash. Except for a minor degradation of overall perfor-

mance, the failure of a busy node (load greater than the threshold) does not affect the system very much. The reason is that, the status of the failed node as perceived by other nodes is that of a busy node, and none of the nodes will attempt to transfer a task to this node. But, the failure of a lightly loaded node may affect the performance of a system adversely. In this case, the rest of the system will continue to view the failed node as a lightly loaded node and will try to transfer tasks to it. These transfer attempts will fail and the transferring nodes will have to find alternative servers or process the tasks themselves. In either case, it may become expensive.

The threshold policy, on the other hand, is more resilient to a node crash. Any attempt to query a failed node will be unsuccessful, and an alternative node will be selected for probing. If the nodes do not exchange information when the system load is high, this will cause little degradation in performance.

We have so far mentioned the fact that if the threshold policy is able to gauge the busyness of a system and adjust its probe limit accordingly, it can achieve significant performance improvements. But, we have not discussed how we may modify the policy to enable it to do so. In the controlled environment, in which this experiment was conducted, it is relatively easy to detect any increase in traffic intensity. Also, we assumed that all nodes become very busy at the same time. That is, an increase in load at one node implies that the rest of the system is experiencing similar increases. In a more realistic situation, the decision to share load cannot be based solely on the traffic intensity of the local node. It should also take into account the busy/idle status of the rest of the system.

There are several ways we may determine the busyness of a system. One of the ways is to keep track of number of times a node fails to locate a server consecutively and set an upper limit for this number. We call this number the failure threshold. If a node exceeds this threshold, it may consider the system to be overloaded. Combining the failure threshold with the local traffic intensity (which a node can easily monitor), a node may be able to determine whether or not the

system as a whole is overloaded. If it finds the system overloaded, it can refrain from attempting to share load for a predetermined time or lower its probe limit.

Alternatively, a node can use two thresholds on the local load. The smaller one of these thresholds can be used to decide whether a task should be transferred to a remote node or processed locally, and the larger threshold can be used to decide whether or not a node should try to locate a lightly loaded node.

The values of these additional thresholds suitable for a particular system depend on the nature of the workload in that system and may be obtained by a method of trial and error. Finding a suitable metric or a combination of metrics that will gauge a system's busyness is an area we have not explored and should be addressed in further research.

References

[Agarwala & Tripathi 81]
A.K. Agarwala and S.K. Tripathi, "On the optimality of semidynamic routing schemes," *Inform. Processing Lett.*, vol. 13, pp.20, 1981.

[Bryant & Finkel 81]
R.M. Bryant and R.Finkel, "A stable distributed scheduling algorithm," *Proc. 2nd Int. Conf. Distributed Comput. Syst.*, pp. 314-323, April 1981.

[Chou & Abraham 82]
T.C.K. Chou and J.A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 401, 1982.

[Chow & Kohler 79]
Y.C. Chow and W.H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, vol. C-28, pp. 354, 1979.

[Chu, Holloway, Lan & Efe 80]
W. Chu, L.J. Holloway, M.T. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Computer*, vol. 13, pp. 57, November 1980.

[Cinlair 75]
E. Cinlair, *Introduction to stochastic processes*, Prentice Hall, Englewood Cliffs, New Jersey, 1975.

[Cox & Lewis 66]
D.R. Cox and P. A. W. Lewis, "The statistical analysis of series of events," Methuen, London, 1966.

[Eager, Lazowska & Zahorjan 85]
D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated dynamic load sharing," Tech. Report 85-5, Department of Computational Science, University of Saskatchewan, April 1985.

[Eager, Lazowska & Zahorjan 86]
D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol.SE-12, pp.662-675, May 1986.

[Lewis & Shedler 79]
P.A.W. Lewis and G.S. Shedler, "Simulation of nonhomogeneous Poisson processes by thinning," *Naval Research Logistic Quarterly*, vol. 26, pp. 403-413, 1979.

[Livney & Melman 82]
M. Livney and M. Melman, "Load balancing in homogeneous broadcast distributed systems", *Proc. ACM Computer Network Performance Symposium*, pp. 47-55, April 1982.

[Ni & Abani 81]
L.M. Ni and K. Abani, "Nonpreemptive load balancing in a class of local area networks," *Proc. Computer Networking Symposium*, December 1981.

[Powell & Miller 83]
M.L. Powell and B.P. Miller, "Process migration in DEMOS/MP," *Proc. 9th ACM Symposium on Operating System Principles*, pp. 110-119, October 1983.

[Ramaritham & Stankovic 84]
K. Ramamritham and J.A. Stankovic, "Dynamic task scheduling in hard real-time system," *IEEE Software*, vol. 1, pp. 65, July 1984.

[Speakman 86]
T. Speakman, "Load sharing strategies for a distributed computer system," M.Sc. Thesis, School of Computing Science, 1986.

[Stankovic 85]
J.A. Stankovic, "Stability and distributed scheduling algorithms," *IEEE Trans. Software Eng.*, vol SE-11, pp. 1141-1152, 1985.

[Stone & Bokhari 78]
H.S. Stone and S.H. Bokhari, "Control of distributed processes," *IEEE Computer*, vol. 11, pp. 97, July 1978.

[Wang & Morris 85]
Y. Wang and R.J.T. Morris, "Load sharing in distributed systems," *IEEE Trans. Computers*, vol. C-34, pp. 204-217, 1985.

[Yum 81]
T. Yum, "The design and analysis of a semidynamic deterministic routing rule," *IEEE Trans. Commun.*, vol. COM-29, pp. 498, 1981.