1991

An Experiment in the complexity of load balancing algorithms

Charles Carlino

An Experiment in the Complexity of Load Balancing Algorithms

by

Charles Carlino

A Thesis

Submitted to the Graduate Computer Science Department,
School of Computer Science and Technology
Rochester Institute of Technology

1 January 1991

Approved by:

(Dr. Peter Lutz, Committee Chairman)

(Dr. Andrew Kitchen, Reading Member)

(Dr. Peter Anderson, Department Chair)

TABLE OF CONTENTS

1. Introduction

During the operation of a network of computers, it is likely that some of the processors will have a heavy processing load while others will be lightly loaded or idle. When this happens, potentially useful resources (processors) are being wasted and the response time of the overall system is being degraded. The process of reducing this waste and improving response time by redistributing computing load has come to be known as processor *load balancing*. Load balancing has also been applied to other resources such as secondary storage, but this paper deals only with the cpu.

Processor load balancing can be more precisely defined as any procedure which attempts to take advantage of the concurrent processing capability of a *multicomputer* to optimize some performance and/or utilization criterion by rearranging the computing load among the system's processors. A *multicomputer* is defined to be a loosely coupled set of processors, each capable of running independently of the rest. The next three subsections define three broad categories of load balancing problems.

1.0.1. Process to processor matching to maximize system performance

The defining characteristic of problems in this category is the lack of task-related information which may be assumed when devising a solution. Here, the goal of the load balancing algorithm is to optimize system performance by transferring work from one processor to another. The designers of these algorithms assume that little or no information about a given task is available beforehand (no a priori knowledge). Any predictions about the future characteristics of a task (e.g. how much longer it will run) must be estimated from past task performance (e.g. how long it has already run) or current characteristics (e.g. memory size). Most algorithms in this category do not concern themselves with user level inter-process communication (IPC) issues. That is, the problem of determining the effect of IPC on the "optimal" assignment of jobs to processors is generally not addressed. This is usually because it is assumed that information on user IPC is not available. The systems which do deal with this problem either try to predict future IPC from the process's IPC history, or they fall into the next category.

1.0.2. Distributed program module to processor matching to maximize program performance

In this class of load balancing problems, much more task-specific information is assumed known. Here we have a distributed program composed of modules which run concurrently and coordinate with each other to perform some activity. It is assumed that the level of communication among the modules and their precedence relationships are known ahead of time. This information might come from a sophisticated distributed programming language system. The goal of these algorithms is to find a mapping of the modules onto the processors of the distributed system (as is shown in figure 1-1) in order to optimize performance of the distributed program. This is in contrast to the goal of the previous class: to optimize performance of the system. Algorithms in this category are called task assignment, task allocation, and task scheduling algorithms in the literature.

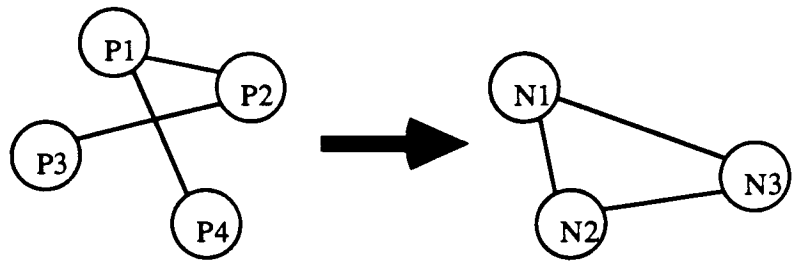


Figure 1-1. Optimizing a Distributed Program Composed of 4 Modules

1.0.3. Maximizing performance subject to system-specific constraints

The last category of load balancing algorithms are those whose goals and assumptions are intimately related to the type of distributed system on which they will run. For example, if you are trying to load balance a distributed database system, your goal might be to minimize the average query time of the system. Typically, these systems are specialized and require solutions specific to the type of system. That is, some characteristics of the system are used in formulating the load balancing policies.

The remaining work in this proposal applies to the first category of load balancing algorithms in which little or no a priori process information is known.

1.1. Problem Statement.

In this work, as in [Eager 86], we wish to determine an appropriate level of complexity for a load balancing policy on a homogeneous network of computers. In addition, certain situations under which load balancing may be avoided with low probability of significant system degradation are investigated to determine the feasibility of startup criteria for load balancing algorithms.

To achieve the former goal, load balancing algorithms of varying degrees of complexity are implemented, their performance compared, and a level of complexity beyond which the gain in performance is small is determined. Results are compared to worst-case and best-case analytical models so that some absolute measure of performance may be obtained. This parallels to some degree the modelling work done by Eager, et.al. in "Adaptive Load Sharing in Homogeneous Distributed Systems" [Eager 86] and provides some real-life verification of their results.

To provide some insight into startup criteria for load balancing algorithms, a brief investigation is performed making use of an analysis done to determine the likelihood of situations where load balancing would be of the most benefit. The results of the analysis are formulae which are tested by comparison with the observed behavior of the system. That is, tests are run with and without load balancing under varying load situations, and if there are substantial performance differences, these should be predicted by the analysis. This analysis is an extension of work by Livny and Melman in "Load Balancing in Homogeneous Broadcast Distributed Systems" [Livny 82] and an application of standard queueing theory results.

The implementation of these policies is done without modifying in any way the underlying UNIX operating system. Therefore, only shell-level export of processes will be supported.

In the proposal, it was planned that an "idealized" algorithm would be implemented in which knowledge of existing and/or future task load situations might be used to obtain nearly optimal performance. This plan was abandoned, however, due to concerns over implementation complexity and time requirements. Also, the performance the optimal M/M/c model (a theoretical upper bound on performance) with c=5 nodes was much closer to the system's actual performance than was shown in [Eager 86] with c=20 nodes. This reduced the need for an "idealized" algorithm.

1.2. Previous Work

Figure 1-2 is a conceptual model of how a load balancing algorithm might work. This diagram is introduced not to attempt to completely describe all possible policies, but rather to establish a conceptual framework for further discussion. In the diagram, tasks move along the thin lines while system state information flow along the heavy lines. Also, the textured lines refer to movement between processors while solid lines indicate intra-processor movement.

As can be seen from the arcs labelled T1, T2 and T3, tasks arrive at a processor in one of three different ways. They are initiated at the local processor, they are sent to the local processor from another processor before beginning execution, or their execution on another node is interrupted and they are migrated to the local processor. At this point, it makes sense to distinguish between tasks before they have begun executing and tasks once execution has started. So, for the remainder of this paper, the term *job* will refer to a task before execution has begun, the term *process* will apply to a task after it has started running, and the term *task* will be used if either may be true. In the two cases in which the task has not yet begun executing, we identify a *job move* function whose purpose it is to decide whether a given job should be kept local or moved on to another processor. In the case of a task which has already started executing, the *process move* function decides whether to keep it, or move it along. This function also serves to determine which, if any, of the tasks running locally should be shipped out to another node. In addition to these two functions, an *information* function might exist if either move function needs to know the state of other nodes in the system in order to make its decisions.

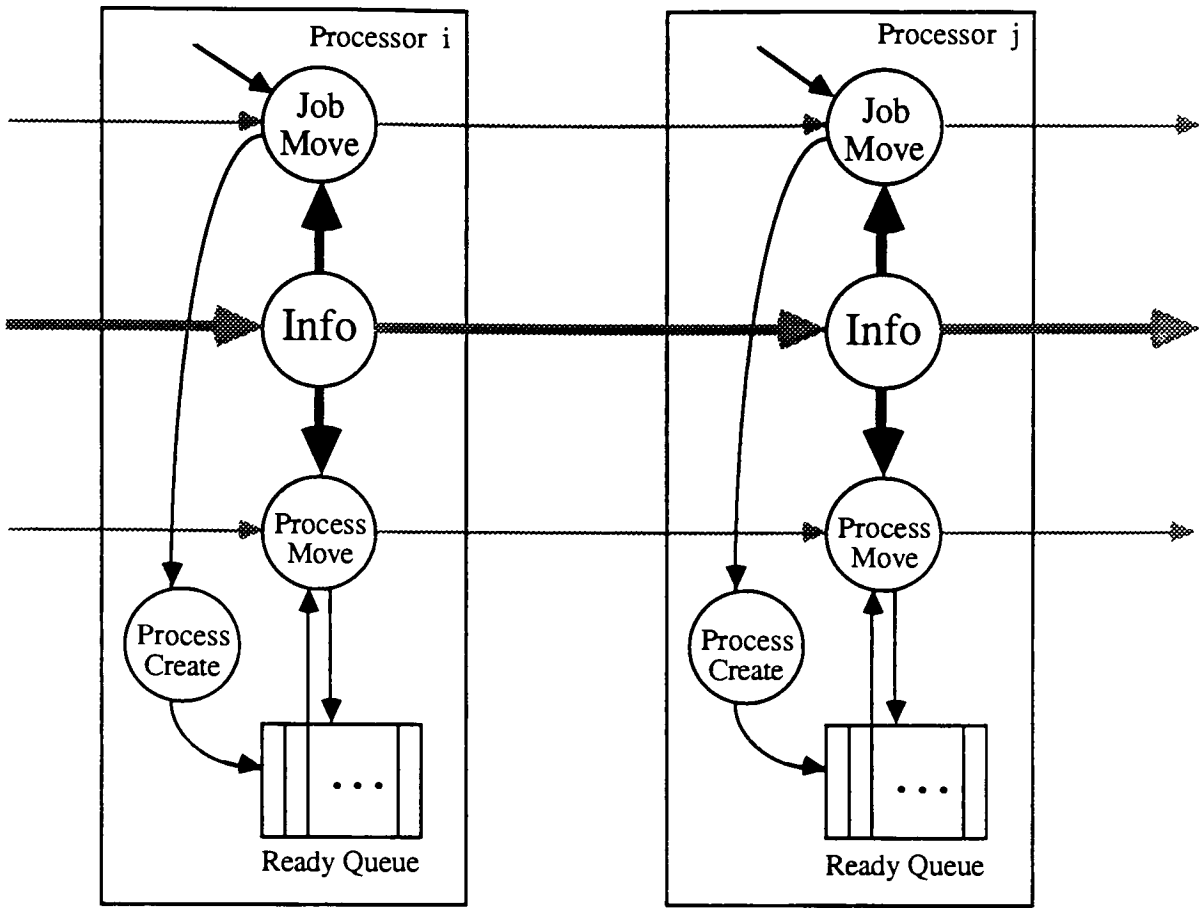


Figure 1-2. A Conceptual Model of Load Balancing

Load balancing policies have been classified many ways in the literature. A policy might be called process scheduling, preemptive or *dynamic load balancing*, process migration, or task migration if the "process move" function exists within it. A policy without that function but with the job move function is sometimes called job scheduling or non-preemptive or *static load balancing*. A policy is called *adaptable* or system state dependent if the "info" function exists and is used by job or process move, and *non-adaptable* or state independent otherwise.

If the job move or process move function selects the target node according to some probability distribution, the algorithm is said to be *probabilistic*. A *deterministic* technique uses some predefined nonrandom algorithm to make the choice. A policy is called *sender-originated* if the node exporting the task initiates the load shifting procedure and *receiver-originated* otherwise. An algorithm may be *optimal* or suboptimal (*heuristic*), depending on whether it is a proven mathematical solution to the problem, or just a technique for improving the system's performance. (There are no workable optimal solutions to this class of load balancing problems.)

As with other distributed operating system functions, the load balancing policy can be distributed among the processors in a number of ways. A policy is *democratic* if all the nodes have equal load balancing authority. It is *hierarchical* if the nodes are arranged like a company's management structure, with manager, supervisor, and worker nodes. Load

balancing is called *central* if there is one node executing the policy on behalf of the entire system. Also, the balancing algorithm can be divided among the nodes *functionally*, or it can be *replicated*, with each node performing all functions of the policy.

The load balancing policies implemented in the thesis are *static, heuristic, adaptable, deterministic, sender-originated, democratic,* and *replicated* algorithms.

1.2.1. Providing system information

The function of gathering and distributing system state information can be split roughly into three parts, as is illustrated in figure 1-3. The input part serves to collect information from the other nodes in the system and, if necessary, translate it into a form usable by the job move and/or process move functions. The local part monitors the local load and provides that information to the move algorithms. The output function makes local load information available to other nodes in the system. The analogous functions in the thesis system design are Get Remote Load (2.3), Maintain Local Info (2.1), and Send Local Load (2.2), respectively.

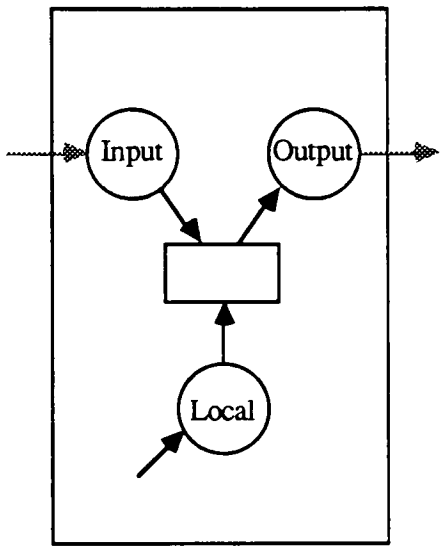


Figure 1-3. The Info Function

1.2.1.1. Goals.

The objective of the info function is to allow the move functions to use load information from other nodes in the system in making their decisions. A second, and equally important goal is to minimize communication overhead while achieving the first goal. This issue is sufficiently important to be considered a major design goal because of what has been called the saturation effect [Chu 80]. Figure 1-4 illustrates the effect wherein as the number of processors increases, communication costs dominate the gain due to concurrency and actual throughput decreases. An ideal system without such costs would experience a linear gain in throughput. This implies that any load balancing algorithm will have a limit on the number of processors which it can service before adding another one will decrease performance [Livny 82]. (This effect is due to task movement as well as passing state information.)

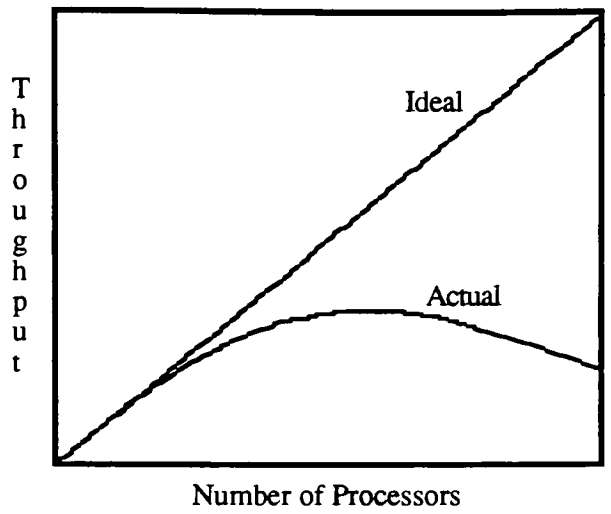


Figure 1-4. The Saturation Effect

1.2.1.2. Issues.

In addition to the communication problem described above, there are several issues regarding the collection and distribution of load balancing information. A starting point in any state transfer mechanism is the determination of the local load. A parameter or set of parameters must be calculated which indicate how much work the processor has to do, but which smooth out any rapid fluctuations due to, for example, the blocking and unblocking of processes in a multiprogramming environment. Such fluctuations may not reflect the true state of the processor and may cause instabilities (bad job movements) in the distributed system [Barak 85b]. In the thesis project, the load will change only when a job starts or completes. This should avoid any spurious load fluctuations.

Also, there is a fundamental relationship between the duration of a load fluctuation which can be responded to effectively and the time it takes to move a task. If you try to respond to a change which will have disappeared by the time the job you sent gets there, you will have created an instability. Hence any fluctuations in the local load which come and go in less than the time it takes to move a task should be filtered out. This problem is not dealt with in the thesis system.

Another stability problem which must be dealt with when balancing the processor load arises when one processor becomes suddenly lightly loaded and every other node that finds out about the situation sends jobs to that processor. This results in a swamped node which itself then exports tasks. One way of dealing with this problem is to construct an information exchange policy which limits the number of processors that hear about the lightly loaded node [Barak 85b, Bryant 81]. There are also ways of dealing with this issue in the job/process move functions, which will be discussed in a later section. This problem as it pertains to the thesis project will be discussed when the algorithms themselves are presented.

Other less significant issues which may be addressed in the info function design include:

(1) dealing with out-of-date information (i.e. making it useful or throwing it away),
(2) making the per-node overhead as small as possible (i.e. efficient algorithms and minimal interruption of normal activities), and

(3) dealing with the potential increase in the amount of information transfer in a heterogeneous system over a homogeneous one.

1.2.1.3. Approaches

Information distribution

There are many mechanisms for exchanging load information to be found in the literature. *Broadcast* is one of the simplest in which each node sends either its local load or its view of the system to all other nodes in the system. This method, while very simple, is very expensive. For each broadcast, all nodes must interrupt their normal processing to absorb the new information. Also, the number of messages in a point-to-point system would grow very large as the number of nodes increases. It does, however, allow all nodes to adapt quickly to changes in the system load. Unfortunately, the second stability problem mentioned in the previous section is made worse by this very characteristic.

Notify-when-idle algorithms and *bidding* algorithms are special cases which are similar yet opposites. In both cases, system state information is exchanged when a task exchange is desired. In *notify-when-idle* (also called *drafting*)algorithms, lightly loaded processors request more work from other nodes in the system [Livny 82]. In *bidding*, heavily loaded processors request takers for their excess work [Stank 84b]. The decision on whether or not to send out a request can be based on either local or system information. In the latter case, another distribution mechanism must be in place to provide the required information. These methods are fairly sophisticated, although in *notify-when-idle* the processor which does most of the work is lightly loaded. This is in contrast with *bidding* where the heavily loaded processor must bear most of the burden. On the other hand, *bidding* should respond more quickly to load surges on a particular node since, in *notify-when-idle*, a heavily loaded processor must wait until it receives a request for more work before it can export any of its processes. Overhead issues in this method rely greatly on the communication method used to transmit requests.

In *pairing*, nodes form temporary pairs for the purpose of exchanging load information, and possibly moving jobs [Bryant 81]. This kind of technique has relatively low per-node and communication overhead, and is very stable. But it allows the system only a relatively slow and limited adaptability to system load changes.

In *random scattering*, a node will select a node or small set of nodes randomly and send local and/or system information to it [Barak 85b]. Overhead, stability, and response should be about the same as pairing. However, random scattering could allow any particular node to go without new state information for a long time, and this effect increases as the number of nodes increases.

Probably the simplest scheme, and the scheme used by most of the algorithms implemented in the project, is *polling* or *probing*. Here the node which wants to export a task selects potential target nodes and requests load information from them. It then uses this information to make its target decision (see [Eager 86]).

There are also feedback schemes in which after a job is moved, the target node tells the source node whether or not the movement was beneficial [Mirch 86]. This is used in addition to one of the other techniques and causes minimal additional overhead. The source node uses the feedback to improve future movement decisions.

Information structure and content

Defining the local load is a necessary first step in determining the system load and can be done in many ways. There are several examples [Barak 85b,Livny 82,Stank 84a] of using the length of the ready-to-run queue to estimate local load. A more sophisticated approach is to try to estimate a virtual response time for a typical job [Bryant 81,Chow 79,Hwang 82,Juang 86,Wah 85]. Another method is to examine each process running on the local processor individually to determine if its performance here is good or bad [Stank 84b]. This method requires a good deal of process-specific information.

System state information can be stored in the nodes of a distributed system in several different ways. First, every node can maintain its own view of the state of the system (or part of it) [Barak 84b,Mirch 86,Ouste 80]. Another method is to maintain a hierarchy of information, as is illustrated in figure 1-5 [Witti 80]. Alternatively, there could be a single node which contains all the load status information. A variation on this idea using monitor nodes is presented in [Stank 85].

As stated earlier, the thesis project keeps track of all job entries and completions to maintain a current count of jobs at each node. No attempt is made to make this count predictive. Each node requests remote load information when the load balancing algorithms demand; none of this data is retained for future use.

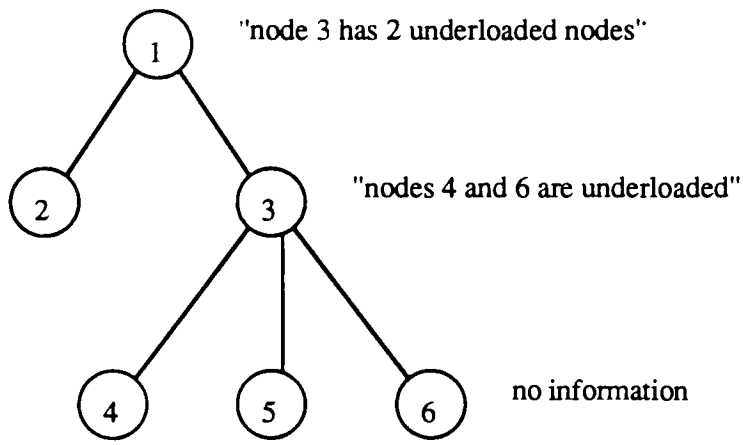


Figure 1-5. Example of hierarchical info organization

1.2.1.4. Results

In addition to those mentioned in the previous section, there are a couple of interesting results in the literature. In calculating local loads, Barak and Shiloh show a simple extension from number of jobs ready to run to an estimation of response time using processor speeds [Barak 85b]. Also, Mirchandaney and Stankovic illustrate in [Mirch 86] a characteristic of their algorithm in which performance is insensitive to the information exchange frequency up to a point: the point at which the transmitted state information has little correlation with the actual state of the system. They point out that this characteristic limits their algorithm to smaller (or partitioned) networks before the cost of distributing state information becomes prohibitive. In their simulation, each node periodically broadcasts its local load to the other four nodes in the system.

1.2.2. Export and target decisions

The process move and job move functions can both be broken down further. Since both functions have the same basic goals and many of the issues involved are the same for both, they will be treated identically from this point forward unless specifically mentioned. As figure 1-6 suggests, both functions need the following four capabilities:

(1) export: decides whether a task should remain in the local processor,
(2) target: chooses a node for remote task execution,
(3) local: prepares tasks to run locally, and
(4) remote: sends tasks out for remote execution.

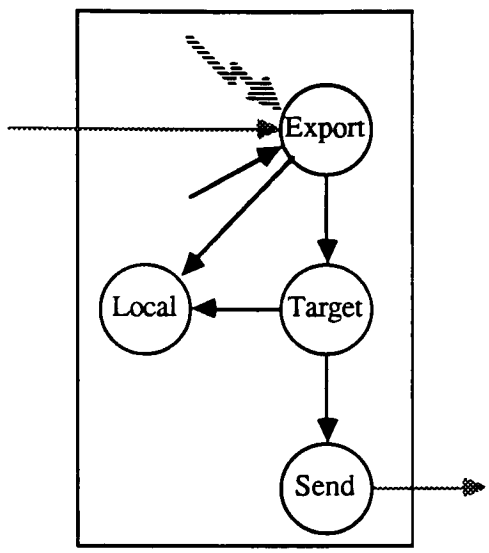


Figure 1-6. The Job Move and Process Move Functions

In this project, the export and target functions are combined into a single function called the LB Algorithm (4.2). The send and local functions correspond to the Send Job (6) and Run Job (5) functions, respectively.

1.2.2.1. Export Goals

The objective of the export decision is to select a task to remove from the local node so that overall system performance will be improved. (Selecting no jobs is a valid decision.) As with any part of the load balancing problem, keeping communication costs to a minimum goes hand in hand with this primary goal. The overall objective then becomes to export the minimum number of tasks necessary to achieve effective load balancing.

1.2.2.2. Target Goals

While optimizing system performance is the ultimate goal, selecting a destination node for a job in order to minimize the run time of the job is frequently seen as the same problem. The tempering secondary goals of avoiding instabilities and minimizing network congestion make the choice of a target algorithm a more subtle one.

1.2.2.3. Export and target decision issues

There are many issues to deal with when building the export and target algorithms. Merely deciding when there are too many tasks in the local processor can be complicated. This decision can depend on the load of the overall system, or just on the load of the individual processor. It can be made a part of the target decision or a part of the export decision. For example, if a policy is triggered by jobs entering the system, and if the policy estimates response times for the local node versus the remote nodes, then the policy has incorporated the determination of a local load threshold into the target algorithm.

There are also several stability issues to accommodate. If a task spends all its time wandering around from node to node and never stops anywhere, it never gets any constructive work done. When this problem becomes epidemic among tasks, processors also spend all their time on load balancing overhead. This is called processor thrashing. On the other hand, the inevitable uncertainty of the state information gathered may cause tasks to be incorrectly transferred. This problem conflicts with processor thrashing because in this case, we would like to be able to re-route the task to a more suitable processor. An appropriate balance between non-excessive task movement and recoverability from improper movements must be struck. The project algorithms all limit the number of times a job can be migrated to once.

One of the stability problems described in section 1.2.1.2 can also be dealt with here. Altering the target decision could reduce the chances of having many nodes dump jobs on a suddenly lightly loaded node. For example, implementing a maximum hop count to eligible destinations would limit the scope of nodes eligible to receive a job. This would effectively limit the number of processors which could send a job to any given node.

There are several factors the target algorithm may want to consider in making its selection of a destination node. If the system keeps track of process communications to other processors in the system and if the load balancing policy is preemptive, the target algorithm could choose a node with which the process has exchanged a large number of messages [Barak 85b]. Next, the suitability of the processor to handle the task (e.g. memory requirements) might have to be considered. This project's algorithms neither collect nor make use of such information.

1.2.2.4. Approaches

There are many ways to approach the problems of deciding which tasks to move to which processors. In probabilistic schemes, the target is selected according to a probability distribution which may be altered to reflect the system state [Chow 79,Livny 82,Mirch 86,Stank 85]. The pairing, bidding, and drafting algorithms discussed in section 1.2.1.3 are adaptive and deterministic. Preemptive [Barak 85b,Bryant 81,Stank 84b] and non-preemptive [Chow 79,Hwang 82,Jones 79,Juang 86,Livny 82,Mirch 86,Ouste 80,Stank 85,Stank 84a,Witti 80] techniques can be used. Preemptive algorithms sometimes require a process to run a minimum time before it can be migrated, either to improve stability [Barak 85b] or to gather statistics [Barak 85b,Bryant 81]. This project considers any job entering the system as eligible for export, and uses a simple load threshold test to make the decision.

A target decision can include many factors, the most popular being an estimated response time for the task on the eligible target nodes. Inter-process communication, memory requirements, cost of migrating the job, hardware requirements, software requirements, etc. can also be considered. A target decision can be either a final destination, or just the "best" nearby node. The nearby node may then move it along to an even better node [Bryant 81,Ghafo 86]. Several different target decision algorithms are implemented in the project and are described in section 1.3.1.

1.2.2.5. Results

Some early results in the literature tend to justify the concept of load balancing and point out some fundamental limitations. Figure 1-7 [Livny 82] points out the need for load balancing by illustrating the high probability of having one node doing nothing while another node has too much to do in an unconnected system of computers.* In that chart, one can also see that at very high and very low node utilizations, load balancing may not be appropriate, especially for smaller N (number of nodes). This result is generalized to include other situations in which system performance should be enhanced by load balancing in section 1.3.2. It has also been shown that even when the average arrival rates at each of the nodes are identical, load balancing can provide significant improvement in system performance [Bryant 81].

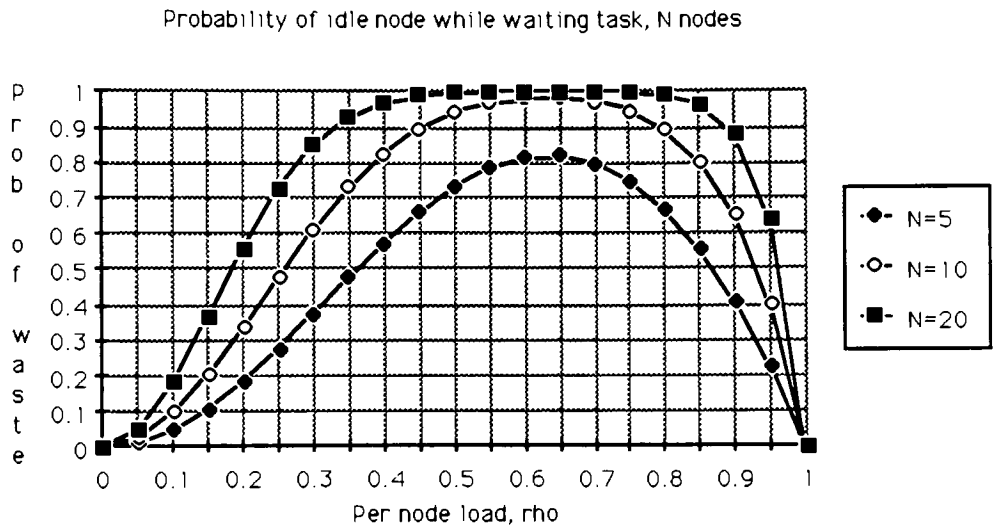


Figure 1-7. Wasted Resources in an Unconnected System.

However, the improvements load balancing achieves do not come for free. To achieve optimal balance, many job migrations are going to be necessary [Livny 82], and the saturation effect described earlier and shown in figure 1-4 is going to take its toll on system performance. This suggests that "every [load balancing] algorithm reaches a point at which an increase in the number of servers decreases the performance of the system" [Livny 82]. It also suggests that algorithms should be designed with a strong emphasis on minimizing communication costs. It has been suggested that "the expected transmission delays of the balanced system" be used in determining a load balancing policy [Livny 82].

With the importance of communication overhead in mind, it is sensible that algorithms have been developed specific to a particular type of network [Juang 86,Livny 82,Wah 85]. One interesting result here is an algorithm on a multiaccess network (e.g. Ethernet) whose overhead is independent of the number of nodes in the network [Wah 85]. This result is not considered in this project because to implement it would require specialized hardware.

---

* This chart is not based on the formula given in [Livny 82], but more closely resembles the chart therein. The [Livny 82] formula has a typographical error. The corrected formula is derived in Appendix A.

Several results specific to a particular type of algorithm are available. Non-preemptive techniques which are triggered by new job arrivals can be modelled by a multi-server queueing system with job routing [Juang 86,Chow 79] as shown in figure 1-8. State dependent or adaptive policies show better performance than state independent ones [Chow 79] and can be almost as good as a single fast computer [Juang 86]. Finally, in a bidding system, Stankovic and Sidhu demonstrated that adapting the distance a bid travels to the system load improves the performance of the system [Stank 84b].

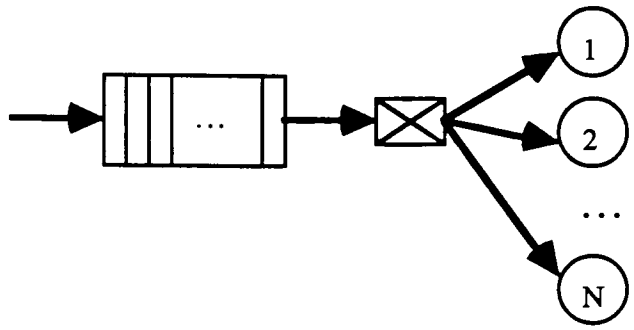


Figure 1-8. Job Move Triggered by Job Arrivals.

1.2.3. Task movement

Task movement is the actual mechanics of migrating tasks from one node to another. This function is fairly straightforward in the case of non-preemptive load balancing, so the remainder of this section will deal primarily with preemptive policies.

1.2.3.1. Goals

The main goal of task movement is to cause a task which is executable on one node of the system to become executable on another node in a transparent manner. Neither the user or the task should have to do anything to accommodate the change. As with all load balancing functions, per-node and communication costs must be minimized.*

1.2.3.2. Issues

There are two main issues involved in the design of a task migration function. The first is the isolation of the process and its environment from the hardware it is running on. Things like open files and mounted devices must be handled in such a way as to allow them to be accessed in the same way from any node. The implication here is that the structure of the operating system should take into account these process migration requirements. [Barak 85a] provides a good description of an approach to this problem.

Also important in the movement of tasks is the network's capability to handle those transfers very efficiently. Since we already know that many transfers are necessary to maintain a balanced load, high efficiency in handling those transfers is going to be vital to the performance of the system.

---

* Disclaimer: I found only 1 article [Barak 85a] dealing with process migration. This section results from examining that paper.

1.2.4. "Adaptive Load Sharing in Homogeneous Distributed Systems"

This paper [Eager 86] by Eager, Lazowska, and Zahorjan deserves special mention here because it provides all the test algorithms that are implemented in this project. It is also the original motivation behind the entire project.

In their work, Eager et. al. claim and demonstrate that the use of very simple policies which use very little system state information "yield dramatic performance improvements relative to the no load sharing case" and "yield performance close to that which can be expected from complex policies".

Their system model consists of a homogeneous collection of processors connected via a broadcast network (such as Ethernet). All nodes have Poisson arrivals with a common mean interarrival time and serve those jobs in exponentially distributed time, again with an average common to all nodes. The cost of job transfers is modelled as a processor cost rather than a communication cost (these are neglected). This last assumption is supported by reference to experimentation and by analysis. The cost of probing a node for load information is assumed to be negligible. Job transfer activity at each node is given preemptive priority over job processing. The model doesn't care which particular service discipline is used for processor scheduling, as long as it selects tasks without concern for their actual service time.

In the decomposition of their model (done to simplify the analysis), they assume "that the state of each node is stochastically independent of the state of any other node". They claim this approach "is asymptotically exact as the number of nodes increases". They further claim validation of their major results for networks of as few as 20 nodes. Tests in the thesis are run on fewer than 20 nodes and thus an attempt is made to expand the applicability of their results.

This project does not attempt to distinguish between the processor and communication costs of transferring a job since an existing communication package is used to implement transfers. However, the assumption that one predominates over the other might tend to break down when the system becomes very busy and collisions on the Ethernet become more frequent. Therefore, if significant deviation from their analysis occurs mainly during high load system states, it may be an indication that the communication costs are becoming non-negligible. Their assumption of negligible probing (polling) costs is tested by direct measurement.

2. Theoretical Development

There are two distinct areas of effort described in this paper. The first is the design, implementation and testing of several load balancing policies on a network of UNIX-PC's. The second is a queuing theory analysis of possible startup criteria for load balancing algorithms in general. Sections 2.1 and 2.2 respectively cover these topics. Section 2.3 speaks briefly on ensuring fairness in comparing the algorithms. Section 2.4 describes the analytical models used in this paper.

2.1. The Load Balancing Algorithms

As was stated earlier, we are attempting to determine a level of algorithm complexity beyond which the gain in performance is small, and so we will test algorithms of increasing complexity and measure any corresponding increase in performance. In order to demonstrate that the algorithms are in any way beneficial, control tests will be run in which no load balancing will be done. We will also compare results to a "worst case" analytical model (c M/M/1 queues) as well as to an analytical best case (M/M/c queue).

All algorithms are taken from [Eager 86] and are described in the sections below. Any differences between the algorithms used here and those in [Eager 86] are described herein.

2.1.1. Random

This is the simplest algorithm which uses no system state information at all. It simply decides to migrate an entering job if the local load exceeds a given threshold. We will henceforth refer to this technique of making the export decision as the Threshold Test. The target node is selected at random. As shown in [Eager 86], such an algorithm is unstable if nodes are permitted to export jobs received from another node. This instability can be removed by placing a limit $\mathbf{L_t}$ on the number of times any given job may be migrated. After a job has been moved $\mathbf{L_t}$ times, it must be processed wherever it is. $\mathbf{L_t}$ is set to one in this implementation.

2.1.2. Threshold

This algorithm uses the same technique to make the export decision as in *Random*, but potential targets are selected at random and polled until one is found which can accept the job without its local load going over the threshold. A static limit $\mathbf{L_p}$ is placed on the number of potential targets which may be polled. If that limit is reached without finding an acceptable target, the job in question must be processed locally. The job, once transferred, must be processed at the target node. No forwarding of jobs is allowed in this algorithm.

2.1.3. Shortest

In this algorithm we again use the Threshold Test for the export decision as in *Random*, but a group of T potential targets is selected and polled to determine their local loads. The job is transferred to the node having the lowest local load, if doing so would not bring the destination's load over the threshold. An improvement over the algorithm presented in [Eager 86] is that if a node is found with 0 jobs, polling stops and the job is transferred to that node. In this algorithm, as in *Threshold*, the destination node must run the arriving job locally, regardless of the node's state when the transfer is completed.

None of the above algorithms deal in any way with indiscriminate dumping of jobs onto a processor which has recently become lightly loaded. Since polling is used as the way of getting load information, a node which becomes free may not be found by many nodes. Although this is no guarantee that the situation will not occur, it should limit its frequency of occurrence. The job migration limit, however, does deal with the processor thrashing problem.

2.1.4. Analytical Models

The M/M/1 queueing model has a well known solution (see [Ross 72]) for response **R** versus system utilization ρ.

$$R = \frac{1}{1 - \rho}$$

This is a lower performance bound which is expected to approximate the performance of the no load balancing control case but to fall short of all other models and experiments. The upper bound M/M/c queueing model has the solution

$$R = \frac{(c\rho)^c}{c!c(1-\rho)^2}\left\{\sum_{k=0}^{c-1}\frac{(c\rho)^k}{k!} + \frac{(c\rho)^c}{c!(1-\rho)^2}\right\} + 1$$

which is derived from the formula in [Tijms 86] for the average delay in queue ($E[W_q]$) of a customer in such a model. From the average time in the queue, one adds the average service time ($1/\mu$) to get the average time in the system and then multiplies by μ to get the response, which is just the average time in the system divided by the average service time. Manipulations such as these are discussed in [Ross 72].

2.2. Criteria for Using Load Balancing

The goal of this section is to define criteria for predicting the usefulness of load balancing in various situations and in so doing, define startup criteria for load balancing policies. This is done by attempting to characterize situations in which load balancing would be useful and calculating the probability that such situations will exist. This analysis is an extension of work by Livny and Melman in "Load Balancing in Homogeneous Broadcast Distributed Systems" [Livny 82].

In that paper, Livny and Melman provide a rough characterization of useful load balancing situations.

> In a distributed system it might happen that a task waits for service at the queue of one resource while at the same time another resource which is capable of serving the task is idle. A load balancing algorithm whose goal is to minimize the expected turnaround time of the jobs will tend to prevent the system from reaching such a state.[1]

They show the general usefulness of load balancing by calculating the probability that "the system is in a state in which at least one customer waits for service and at least one server is

---

[1] [Livny 82] p. 48.

idle."[2] One can generalize this idea by saying load balancing will be useful whenever there exists a node with **t** or more jobs while another node has **m** or fewer jobs (**t>m+1**). In other words, load balancing will be useful whenever one node is very busy while another node is not so busy. "Very busy" and "not so busy" are implementation-dependent parameters. Typically, UNIX-PC's of the type in the GCSD lab can handle 2 concurrent jobs before the wait becomes prohibitive. They may be considered "not so busy" with 1 or 0 jobs running.

The formula resulting from this analysis should reduce to the [Livny 82] result when t=2 and m=0.

2.2.1. The FIFO Server Model

The intent is to calculate the probability that 1 or more of the nodes in the system has **t** or more jobs and 1 or more nodes has **m** or fewer jobs (call this probability $\mathbf{P_{twm}}$). Assume **n** identical unconnected processors modelled as M/M/1 FIFO servers.

Let I_i be the probability that some subset of **i** processors has **m** or fewer jobs and let B_i be the probability that **i** processors have m+1 or more jobs with one or more of them having **t** or more jobs. $\mathbf{P_{twm}}$ is calculated by considering all the possible ways that these situations could arise:

1 node with m^- jobs, n-1 nodes with $m+1^+$ and 1 or more with t^+ jobs
2 nodes with m^- jobs, n-2 nodes with $m+1^+$ and 1 or more with t^+ jobs
3 nodes with m^- jobs, n-3 nodes with $m+1^+$ and 1 or more with t^+ jobs
. . .
i nodes with m^- jobs, n-i nodes with $m+1^+$ and 1 or more with t^+ jobs
. . .
n-1 nodes with m^- jobs, 1 node with $m+1^+$ and 1 or more with t^+ jobs

Here, the notation $\mathbf{x^+}$ should be read "x or more" and $\mathbf{x^-}$ "x or fewer". So since there are $\binom{n}{i}$ ways the i^{th} situation could occur, the desired probability is

$$P_{twm} = \sum_{i=1}^{n-1} \binom{n}{i} I_i B_{n-i} \qquad \text{(Eq. 2-1)}$$

Before getting into the actual derivation, here is some queueing theory background for M/M/1 FIFO servers.

Let ρ be the utilization of the node (arrival rate divided by service rate). Then the probability that an M/M/1 FIFO server is idle is $\mathbf{P_0}=(1-\rho)$ and the probability that a node has exactly k customers is well known to be

$$P_k = P_0(1-P_0)^k = (1-\rho)\rho^k \qquad \text{(Eq. 2-2)}$$

The probability that a server has k or more customers is

---

[2] [Livny 82] p. 48.

$$P(k+) = 1 - \sum_{j=0}^{k-1} P_k = 1 - (1-\rho)\frac{\rho^k - 1}{\rho - 1} = \rho^k \qquad \text{(Eq. 2-3)}$$

The probability that a server has between **i** and **j** customers is (j>i)

$$P(i:j) = P(i^+) - P(j+1^+) = \rho^i - \rho^{j+1} \qquad \text{(Eq. 2-4)}$$

Now, in terms of the probabilities calculated above, I_i can be expressed

$$I_i = P(m^-)^i = (1 - P(m+1^+))^i = (1 - \rho^{m+1})^i \qquad \text{(Eq. 2-5)}$$

To calculate B_i, consider all the ways a set of **i** processors could have **m+1** or more jobs while at least 1 of them has **t** or more jobs:

1 node with t^+ jobs, i-1 nodes with between m+1 and t-1 jobs
2 nodes with t^+ jobs, i-2 nodes with between m+1 and t-1 jobs
3 nodes with t^+ jobs, i-3 nodes with between m+1 and t-1 jobs
. . .
j nodes with t^+ jobs, i-j nodes with between m+1 and t-1 jobs
. . .
i-1 nodes with t^+ jobs, 1 node with between m+1 and t-1 jobs
i nodes with t^+ jobs, 0 nodes with between m+1 and t-1 jobs

Then

$$B_i = \sum_{j=1}^{i} \binom{i}{j} P(t^+)^j \, P(m+1:t-1)^{i-j}$$

Using the binomial expansion theorem and subtracting the j=0 term yields

$$\begin{aligned} B_i &= [P(t^+) + P(m+1:t-1)]^i - P(m+1:t-1)^i \\ &= [P(t^+) + P(m+1^+) - P(t^+)]^i - [P(m+1^+) - P(t^+)]^i \end{aligned} \qquad \text{(Eq. 2-6a)}$$

and finally using Eq. 2-3 gives

$$B_i = (\rho^{m+1})^i - (\rho^{m+1} - \rho^t)^i \qquad \text{(Eq. 2-6)}$$

So now substituting I_i and B_{n-i} (Eq's 2-5,6) into the formula for P_{twm} (Eq. 2-1) results in

$$P_{twm} = \sum_{i=1}^{n-1} \binom{n}{i} (1 - \rho^{m+1})^i \, [(\rho^{m+1})^{n-i} - (\rho^{m+1} \quad \rho^t)^{n-i}]$$

Again, applying the binomial expansion theorem and subtracting the i=0 term, this time after splitting the sum into two sums yields

$$P_{twm} = (1-\rho^{m+1}+\rho^{m+1})^n \quad (\rho^{m+1})^n - (1-\rho^{m+1}+\rho^{m+1}-\rho^t)^n + (\rho^{m+1}-\rho^t)^n$$

$$P_{twm} = 1 - (\rho^{m+1})^n - (1-\rho^t)^n + (\rho^{m+1}-\rho^t)^n \qquad \text{(Eq. 2-7)}$$

Note that this does indeed reduce to the [Livny 82] formula as given in the Appendix when t=2 and m=0.

2.2.2. The Round-Robin Server Model -- Late Arrivals

The problem with the above analysis is that it is based on a calculation of the odds that a multiprocessing computer has n concurrent jobs. However, we are modelling that computer with a FIFO M/M/1 queue and calculating the odds that the number of customers in the server is n. This probability is more accurately calculated using a round-robin server model.

To review this model, consider a queue containing 4 jobs. Every **Q** seconds (1 quantum), the job currently executing is interrupted, and if it still needs more processing, it is returned to the end of the queue. The next job in the queue (determined in FIFO fashion) is then run for **Q** seconds before it is interrupted, and so on. Arrivals are handled by adding any which arrived during the last quantum to the queue after the job currently executing has been requeued (if necessary). Kleinrock terms this a "late arrival" system in [Klein 64]. Note that this model assumes that no more than 1 job may arrive during any quantum period. While this assumption is not strictly realistic, it approaches realism as the size of the quantum decreases.

Given the above model, the probability that there are n jobs at the node is given in [Klein 64] as

$$P_n = (1-a)a^n \qquad \text{(Eq. 2-8)}$$

where $$a = \frac{\rho\sigma}{1 - \lambda Q}$$

and $$\rho = \frac{\lambda Q}{1 - \sigma}$$

ρ is as before the utilization of the node and λ is the arrival rate. σ is the probability that a job, having reached the end of a time slice, will require more processing time to complete. **Q** is the length of a quantum.

Note that the formula for P_n is completely analogous to the same probability in a FIFO server with **a** replacing ρ everywhere. Thus, the probability that 1 or more round robin servers will have t or more jobs while some other round robin servers have m or fewer jobs is

$$P_{twm} = 1 - (a^{m+1})^n - (1-a^t)^n + (a^{m+1}-a^t)^n \qquad \text{(Eq. 2-9)}$$

where **a** is as above and n is the number of nodes in the system. Note that this formula cannot be reduced to a function of ρ alone. This model therefore suggests that to predict situations in which load balancing will be useful, the arrival and service rates must be considered independently, not merely as a ratio.

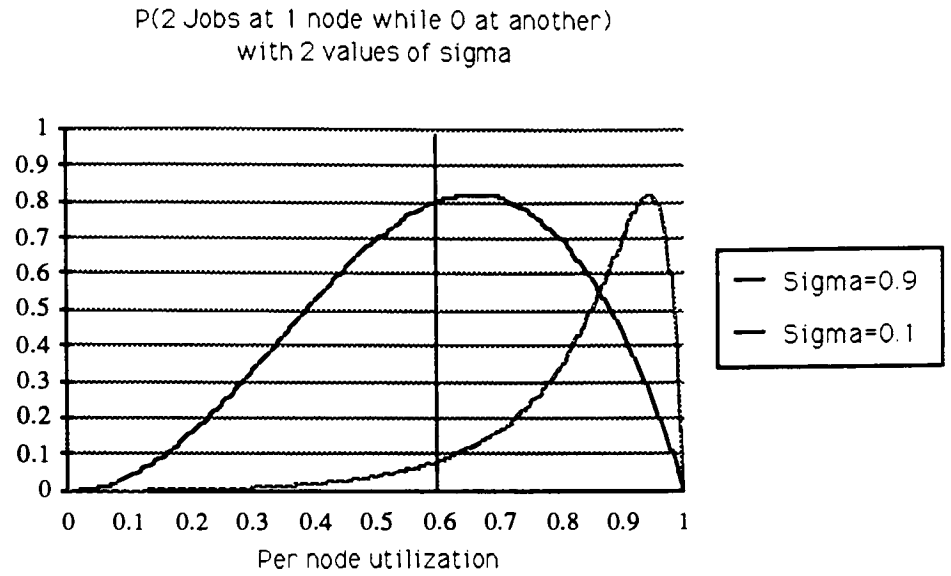


Figure 1-9. Utilization Independence of P_{twm}.

This second point is illustrated by the graph in figure 1-9 which shows two plots of P_{twm} (t=2, m=0) over all utilizations, each with a different constant value for σ. For a given utilization, say 0.5, one can see that the odds of having a good load balancing situation can vary enormously with σ.

2.2.3. The Round-Robin Server Model -- Early Arrivals

This model is essentially the same as the Late Arrival model with the exception that any job arriving during a quantum is added to the queue before the process currently executing is rescheduled. The effect of handling arrivals in this manner on the probabilities of interest are given in [Klein 64] as:

$$P_n = \begin{cases} 1-\rho & (n=0) \\ \left[\frac{1-\rho}{\sigma a^n}\right] & (n>0) \end{cases} \qquad \text{(Eq. 2-10)}$$

where ρ and σ are given as before. The probability that a node has k or more jobs is then

$$P(k+) = 1 - \sum_{j=0}^{k-1} P_k$$

which after a bit of algebra yields

$$P(k+) = \rho a^{k-1} \qquad \text{(Eq. 2-11)}$$

Substituting this back into equation 2-5 yields

$$I_i = (1 - P(m+1^+))^i = (1 - \rho a^m)^i \quad \text{(Eq. 2-12)}$$

B_i is similarly calculated by substituting Eq. 2-11 into Eq. 2-6a. Then

$$B_i = (\rho a^m)^i - (\rho a^m - \rho a^{t-1})^i \quad \text{(Eq. 2-13)}$$

Then Eq. 2-12 and Eq. 2-13 are substituted into Eq. 2-1 and the same technique as in section 2.2.1. is used to arrive at

$$P_{twm} = \sum_{i=1}^{n} \binom{n}{i} (1-\rho a^m)^i \left[(\rho a^m)^{n-i} - (\rho a^m - \rho a^{t-1})^{n-i}\right]$$

Splitting this into two sums and using the binomial expansion theorem after subtracting the i=0 terms gives

$$P_{twm} = (1-\rho a^m+\rho a^m)^n \quad (\rho a^m)^n - (1-\rho a^m+\rho a^m-\rho a^{t-1})^n + (\rho a^m-\rho a^{t-1})^n$$

which after a little algebra yields

$$P_{twm} = 1 - (\rho a^m) \quad (1-\rho a^{t-1})^n + (\rho a^m-\rho a^{t-1})^n \quad \text{(Eq. 2-14)}$$

Note that, as with Eq. 2-9, this formula also depends not only on ρ, but on the arrival and service rates independently.

Equations 2-7, 2-9, and 2-14 will be evaluated over a broad range of load conditions. High probabilities generated by these formulae should correlate well with improved performance yielded by load balancing. If any of the formulae exhibit this correlation, it is a viable candidate for a startup criterion for load balancing.

2.3. Fairness of Comparison

To ensure fairness when comparing the load balancing algorithms defined in section 2.1, common methods of exporting jobs, measuring local loads, and distributing state information are used in all the algorithms. See the Functional Specification section for a description of these methods.

2.4. Performance Measures & Tests

The measure of performance for any job run in the implemented system is the turnaround time for the job divided by the minimum required processing time, which is analogous to the system response calculated in the analytical models. The turnaround time is measured as the time interval beginning just after job process creation until just before the exit status is returned to calling process. It will be measured using standard UNIX time calls. The minimum required processing time for a job is the time required for the job to run on a single, unloaded processor running UNIX. It also is measured using standard UNIX time calls.

3. The Software Project.

3.1. Functional Specification

3.1.1. Information Distribution

Each node in the system is responsible for maintaining a measure of its local processing load. This measure is updated upon the arrival and departure of jobs.Each node provides load information to any other node which requests it. Each node requests load info from other nodes whenever the load balancing algorithm requires it. A suitable interface (function call) for the algorithms to use was implemented.

The load measure used is a self-contained count of the number of jobs running. It will therefore ignore any processes outside the experiment (including system processes and the load balancing processes themselves). No effort is made to make the measure predictive of future system behavior.

The information distributed is a single node's load. No "view of the system" is maintained or distributed.

Each node's view of the network is static. The net is assumed not to change.

3.1.2. Load Balancing Decisions

This function is specific to the algorithm being tested. However, all algorithms must decide whether or not a given process will be run remotely, and if so, to which of the nodes it will be sent. The algorithms are described in section 2.1.

3.1.3. Job Movement

Each node is capable of moving a job to a remote processor for execution at that processor. A job is defined as a command line, part of the user's environment, and any files (input or program) which can be determined from the command line and environment. The result of that execution (exit status of the job process) is passed back to the program which generated the job. Each node is capable of executing jobs initiated either remotely or locally.

Full support for remote jobs was not implemented. Support for I/O, concurrent processing, IPC, etc. is beyond the scope of this project. Jobs are moved before their execution has begun. No dynamic load balancing was attempted.

3.2. Functional Analysis

The system data flow is given in Figures 3-1 through 3-8. These diagrams are a functional specification of the system and not a layout of processes or modules. The last 2 figures describe the data used as arc labels in the earlier diagrams.

The top level of functionality is given in Figure 3-1. It consists of seven sub-functions which will be introduced here, some of which have their own diagrams. The shell interface's purpose is to generate jobs with appropriate durations and arrival times to

generate the desired system utilization. The Info function is in charge of maintaining the local node's load, obtaining the load of any node requested by the load balance function, and providing the local node's load to any node which requests it. The Receive Job function must accept any jobs sent to the local node from a remote node and provide the load balance function with the job-specific information required to decide whether to keep and run the job. The run job function actually executes the job, returning the exit status either to the calling program or to the originating node. Send Job must send out job context and task parameters and wait for the exit status to return from the executing node.

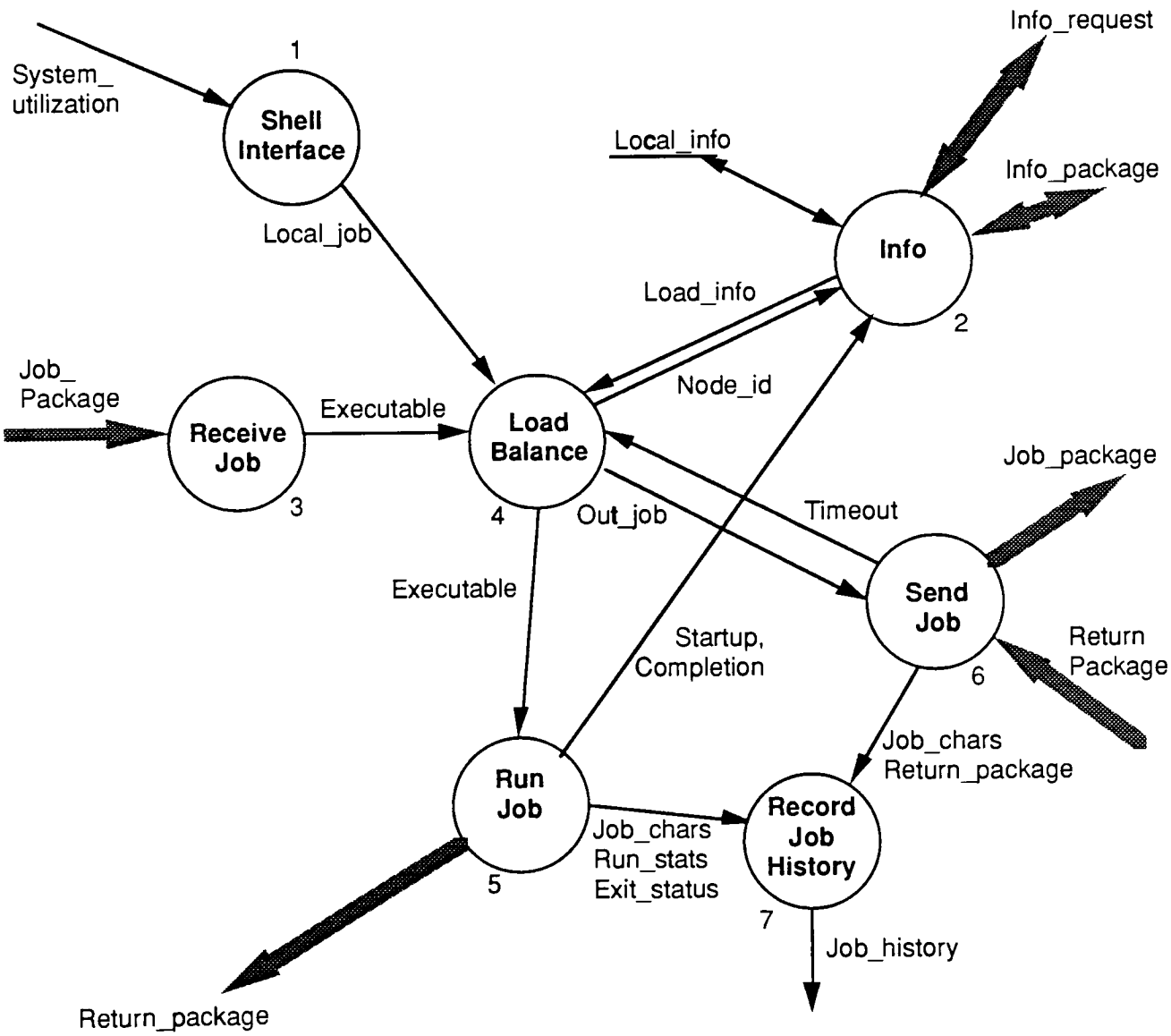


Figure 3-1. Top Level Data Flow Diagram

3.2.1. The Info function

In order to make current system information available to requesting jobs and nodes, three basic tasks must be performed. First, the current load value of the local node must accurately reflect the number of jobs running on the local node. This is done by the Maintain Local Info function (2.1). Next, this local load must be made available to jobs on other nodes by the Send Local Load function (2.2), which sends an Info_package (containing the load information) in response to an Info_request sent by the other node. Local jobs obtain their own node's load via the Get Local Load function (2.4). Finally, load information from remote nodes is provided to local jobs by the Get Remote Load function (2.3), which sends an Info_request to the desired node and receives the Info_package.

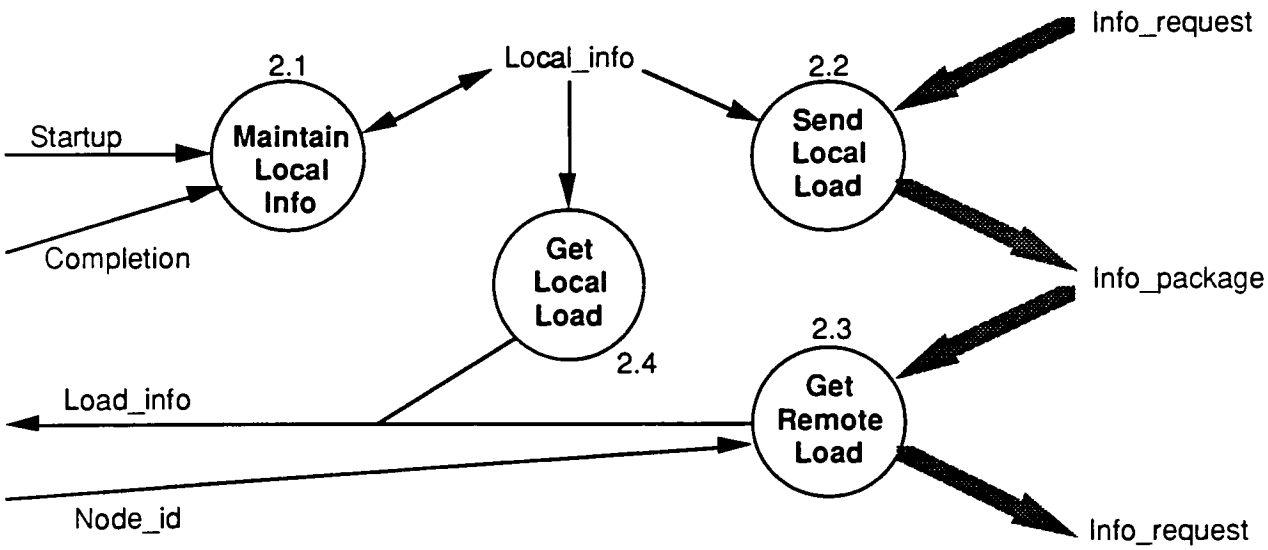


Figure 3-2. Info Function

3.2.2. The Receive Job function

When a node is about to receive a job, it first receives a message (3.1) called a Jobstartup_package which tells it that another node wishes to send a job. The message contains the information needed to receive the job. The receiving node must then establish a TCP connection with the sending node (3.2) so that it may receive and set up the context within which the job will run (3.3). Data is then passed to the receiving node to simulate a remote *exec* (3.4).

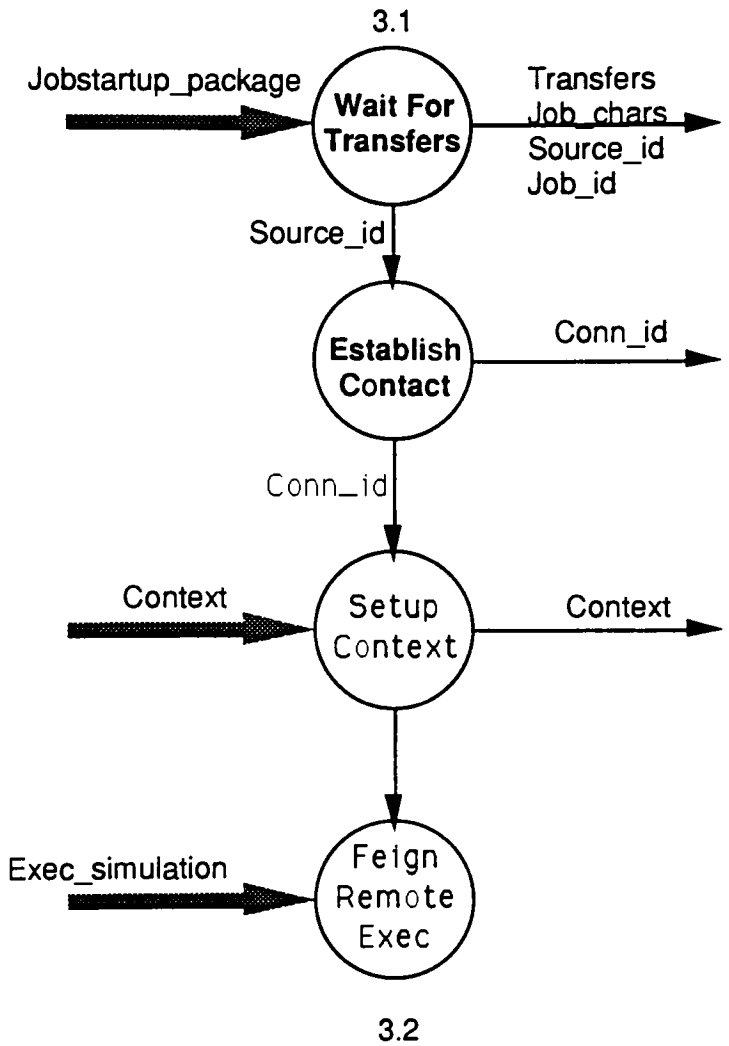


Figure 3-3. Receive Job Function

3.2.3. The Load Balance function

When a job is initiated at a node, either through the Shell Interface or via the Receive Job function, a decision must be made as to where to run the job: locally or on some other node. Within the Load Balance function, first an export decision is made (4.1) which determines whether running the job locally is desirable. If it is, the job is made to run locally by the Effect Decision function (4.3) which routes it to the Run Job function (5).

Otherwise, a destination for the job is chosen by the Make Target Decision (4.2), which gets any required local or remote load information through the functions in section 3.2.1. The result of this function may be a remote node, but may also be the local node if the running algorithm so decides. Once this is done, the Effect Decision function (4.3) packages the job either as an Executable, which runs locally, or as an Out_job, which is sent to another node for possible execution.

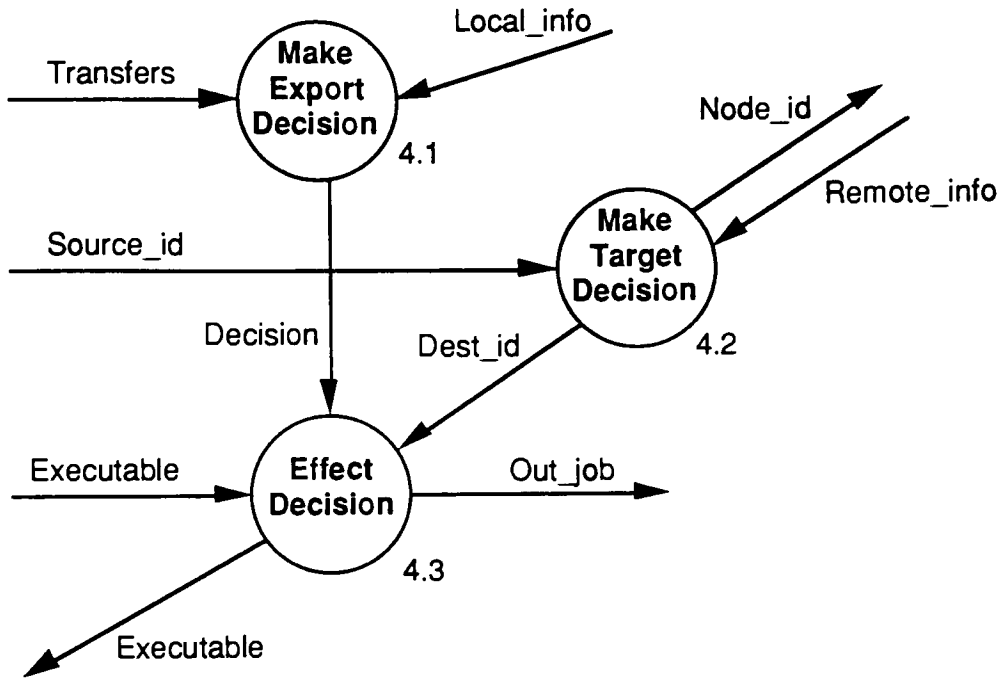


Figure 3-4. Load Balance Function

3.2.4. The Run Job function

Once it is decided that a job is to execute locally, the local load must be updated (5.4), symbolized here by a Startup being sent to the Info function. The job is run (5.2) and its exit status and all statistics collected (5.4) are returned to the calling process (if the job was originated locally) or the the originating node (5.3). A Completion is then sent to the Info function, so that it may maintain an accurate representation of the local load.

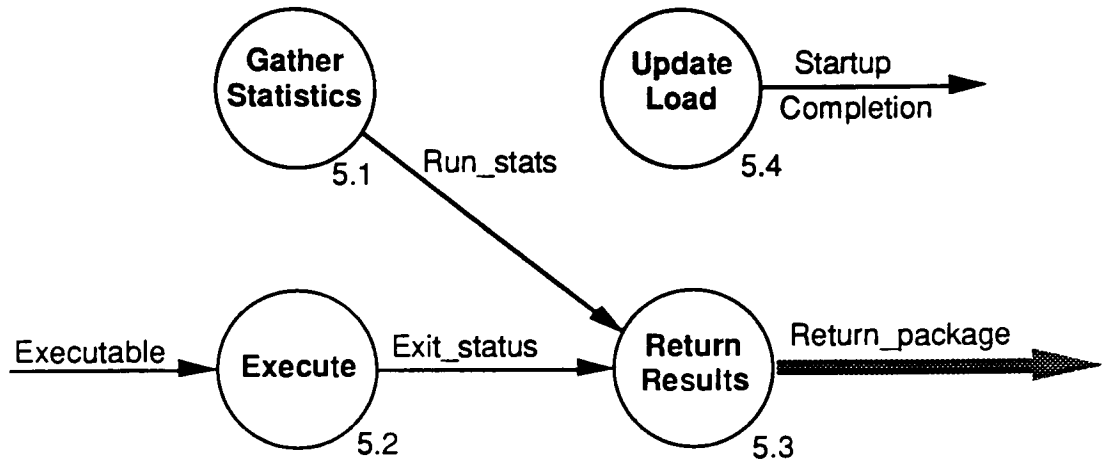


Figure 3-5. Run Job Function

3.2.5. The Send Job function

When it has been decided that a job is to be sent out for remote processing, a Jobstartup_package is sent out to the target node (6.1). A connection is then established (6.2) with the target (possibly different from the original target) and all remaining information required to run the job, including context, is sent (6.3). If the target node refuses to accept the job, it is given back to the Load Balance function as a Timeout for further consideration. If a successful transfer was accomplished, then the statistics and exit status of the job execution is received (6.4) and passed to the Record Job History function (7).

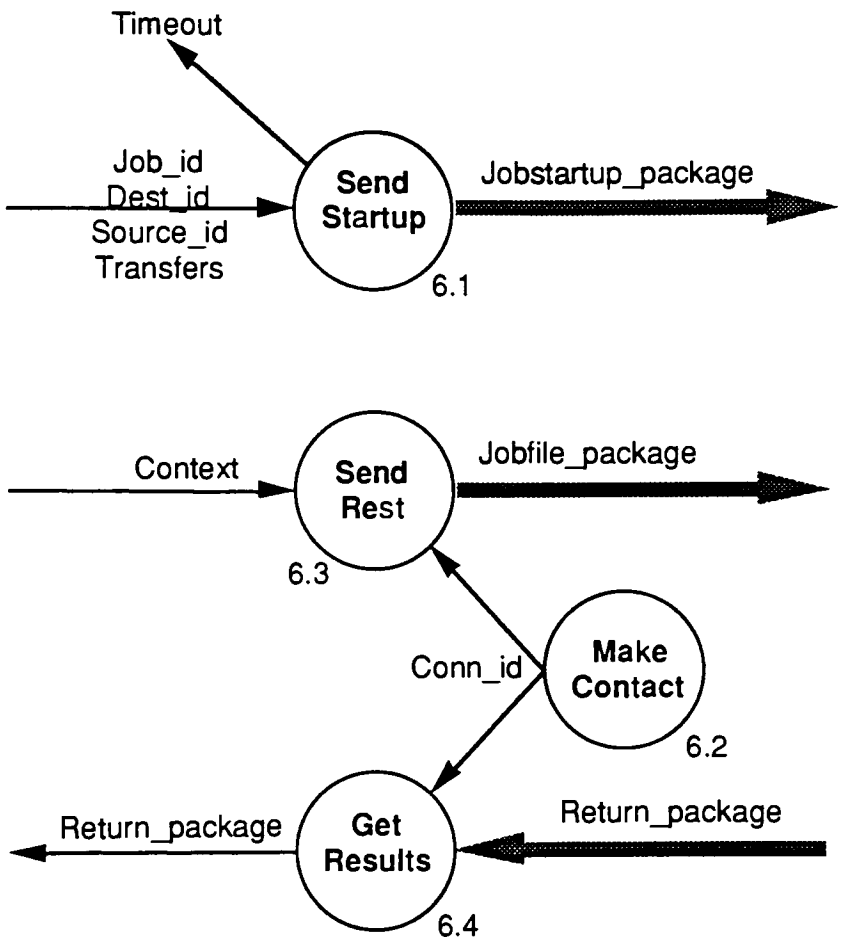


Figure 3-6. Send Job Function

3.2.6. Data Definitions

The data flows shown in the previous diagrams are defined in the next 2 figures. The first figure shows the hierarchical composition of some of the data. For example, a Job_package is composed of 2 pieces of information: a Jobstartup package which alerts the potential node that job files are on the way and the job's context.

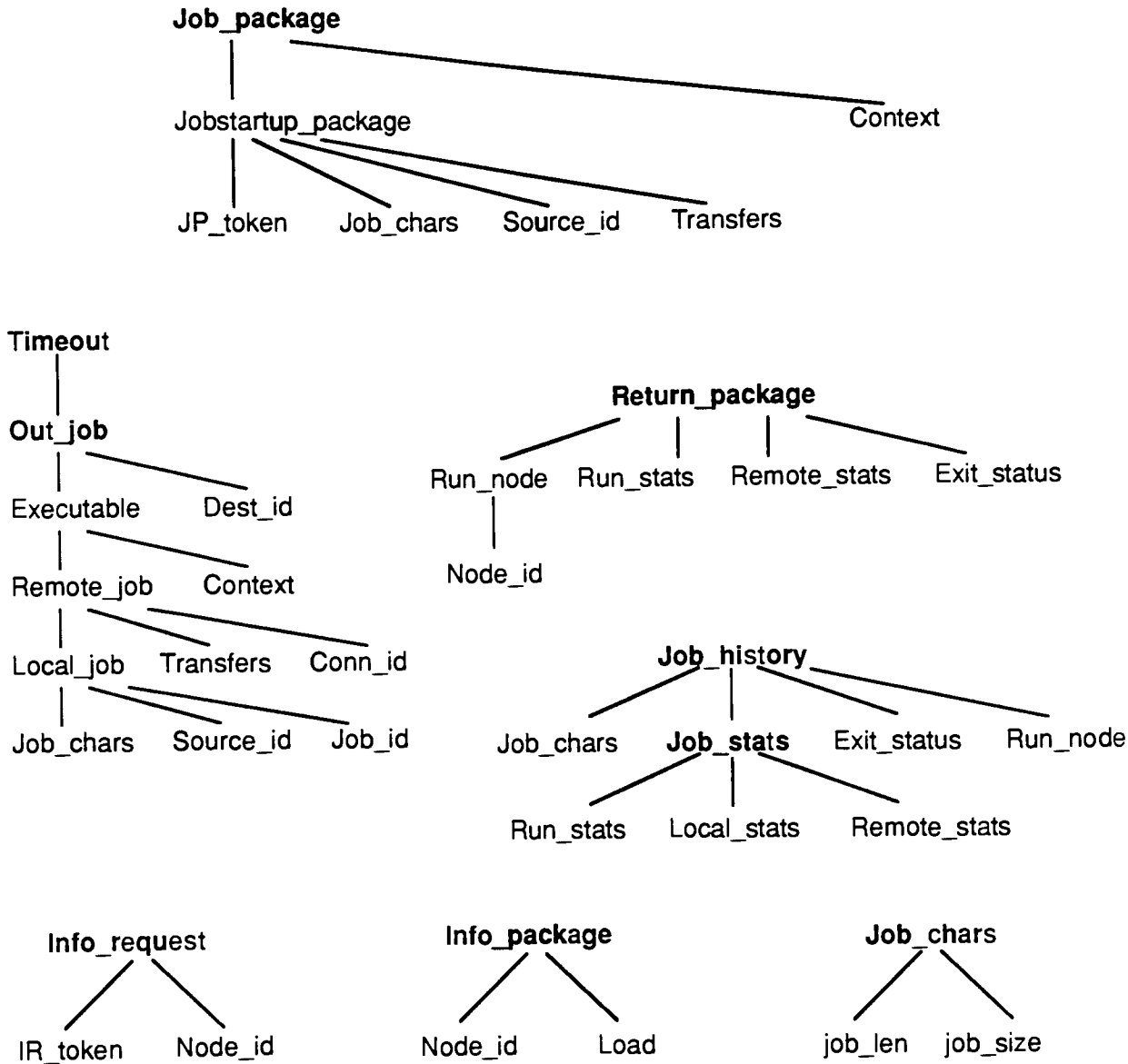


Figure 3-7. Data Dictionary, Data Hierarchy

Command_line	What the user wants executed
Completion	Indicates a job has just completed
Conn_id	A descriptor for the connection between between the remote and local processes of a remote job
Context	The job's context at the originating node
Decision	Whether job will be run locally or not
Dest_id	Where the job is to be sent
Exec_simulation	Fake text to simulate a remote *exec*
Exit_status	Exit return of the job executed
Job_id	With Source_id, uniquely identifies job
Job_len	CPU intensity of job in seconds
Job_size	Memory intensity of job in bytes
Load	The number of jobs executing at a node
Load_info	The number of jobs executing at a node
Local_info	The current load at the local node
Local_stats	Execution statistics at the source node
Node_id	Identifies a node
Nodes	Number of nodes in the network
Path	User's PATH environment variable
Remote_info	The load at a remote node
Remote_stats	Execution statistics at the remote node
Run_stats	Statistics about the job's execution on Run_node
Run_node	Node_id of the node which ran the job
Source_id	Node_id of the node which originated the job
Startup	Indicates a job is about to begin
System_utilization	A measure of the desired system load
Totals	Statistics about the entire load-balanced job run
Transfers	Number of times this job has been transferred

Figure 3-8. Data Dictionary, Atomic Definitions

3.3. Architecture.

3.3.1. Processes.

The overall design of the software is shown in Figure 3-9. The names used here apply to the version of the software used in the experiments (as opposed to the demo system).

The *jobsrc* (*j*ob *s*ource) process was designed to initiate jobs with randomly selected durations after a randomly selected delay. In order that the results of the experiments could be compared with M/M/c and M/M/1 results as well as with the results in [Eager 86], the job durations followed an exponential distribution, as did the interarrival times. Thus, jobsrc creates jobs with exponentially distributed duration as a Poisson process, just as the M/M/1 and M/M/c models.

After having decided on a job duration and after the appropriate delay, *jobsrc* forks an *lb* (*l*oad *b*alancing) process which makes the load balancing decisions. It is here that the algorithms described in section 1.1.3.1 are implemented. If *lb* decides the job is to run locally, it forks the *job* process, waits for its completion, writes the job statistics to a file and exits with the same value as did the *job* process. If *lb* decides the job should be exported to another node, it initiates the job transfer protocol, which is described in the next section. It then wait for the job's run statistics to be returned, writes them to a file and exits as before.

The *job* process takes care of any redirections of input or output and "runs" the job. In the experiments, running the job consists of entering a loop whose sole purpose is to expend cpu time. The number of times this loop is executed is the called the job length. The job length was calibrated to elapsed time (see sections 4.1.1 and 4.1.2) so that job durations (service times) could be accurately predicted.

The *xlbd* (*e*xperiment *l*oad *b*alancing *d*aemon) process is responsible for responding to remote requests for load information or job transfer. It also synchronizes the start of the experiment among the nodes by holding a semaphore until it receives a GO message. The *jobsrc* process attempts to get that semaphore just before it starts producing jobs, so the experiment is held until each *xlbd* sees a GO. *xlbd* forks a *receiver* process in response to a job transfer request. That process then executes the receiving end of the job transfer protocol, which results in the creation and execution of a *job* process and the return of the associated statistics to the sending *lb* process.

The shared memory shown contains experiment and algorithm parameters, as well as load and configuration information. The *xlbd* process sets up the shared memory and fills it with all required data. The data is used by the *jobsrc*, *lb* and *receiver* processes. The node's current load is also kept there and is write protected by a semaphore.

The demo system is structurally identical to the experiment system except that the *jobsrc* process is replaced by the user's shell, no statistics are taken, and the *lb* and *receiver* processes are enhanced to deal more effectively with asynchronous user events. Also, the *job* process *exec*s the requested executable rather than just running a busy loop.

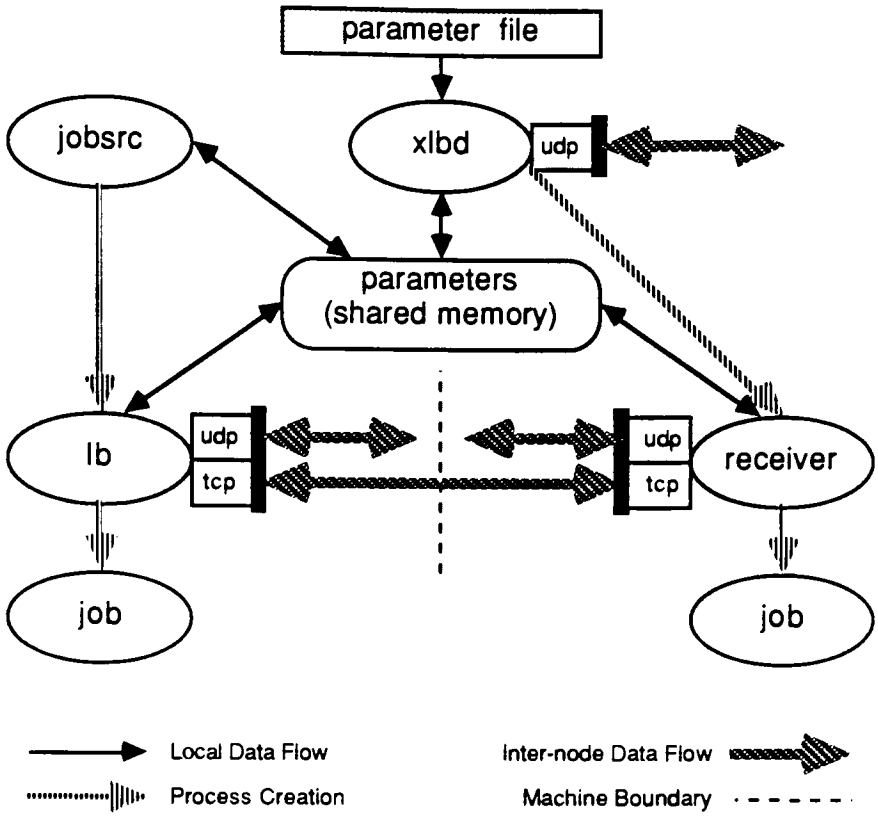


Figure 3-9. Process Structure of Experiment System.

The *xlbd* process sets up the shared memory and fills it with all required configuration data. This data is used by the *jobsrc*, *lb* and *receiver* processes. The node's current load is also kept there and is write protected by a semaphore.

3.3.2. The job transfer protocol.

When jobs are moved from one node to another the following protocol is followed.

1. The sending *lb* process opens a TCP port, records that port number in a UDP message (the job startup message) and sends that message to the *xlbd* process on the receiving node after opening a UDP port.
2. The *lb* process then listens on the TCP port for a connection.
3. The *xlbd* process forks a *receiver* process, which opens a TCP port and attempts to connect with the TCP port on the sending machine.
4. When TCP establishes the connection, the *lb* process sends all contextual information to the *receiver* process through that connection.
5. If the running algorithm is RANDOM, then the *receiver* process decides whether or not to accept the job. If it decides not to, it forwards the job startup message to another randomly selected node and sends a reject indication back to *lb* process, which then closes the connection and goes back to step 2. If the algorithm is not RANDOM or the decision was to accept the job, an acceptance indication is sent back to the *lb* process.

6. In the experiment system only, a block of data is then sent from the *lb* process to the *receiver* process to simulate a remote exec. The *receiver* process allocates memory for this data and simply writes the data there.
7. The lb process then just waits for job completion while the receiver process sets up the environment (directory, uid, gid) for the job and runs it.
8. The *receiver* process sends the exit status and run statistics back to the *lb* process.
9. The *lb* process sends a confirmation indication back to the *receiver* process.

If there is a problem at step 3 on the receiving end, *lb* will time out its listen (step 2) and run the job locally. Once the connection is established and before *lb* receives the exit status and run statistics of the job (step 8), if *lb* detects a failed connection, it fails the transfer and again runs the job locally. If the *receiver* process ever detects a failed connection, it aborts the job and writes whatever info it has to an abort file.

3.4. Verification & Validation

3.4.1. Procedures

Validating the software required testing the various components of the system for basic functionality and for reasonable performance. The four major components of the system were tested: information distribution, target selection, local execution of jobs, and remote execution of jobs. A special "job generator" program was written and modified as required by each of the tests. This took the place of the "job source" program used for the experiments.

In order to validate the target selection function, the following requirements had to be met. The export decision had to be local when the local load was less than the threshold and remote for any other local load. For the *threshold* algorithm, the target selected had to be the first node polled whose load was less than its threshold. For the *shortest* algorithm, the node selected had to have the lowest load of those polled, unless a node was encountered with a zero load, in which case polling is aborted and that node selected. For any algorithm, the software had to run the job locally in the event polling failed. A test of the randomness of polled node selections was deferred until the experiments, when it will simply be measured. No important effects due to a lack of randomness were expected.

To verify information distribution, all that was needed was to show that load information is sent by the daemon in response to requests and that the load received by the client is the same as that sent by the daemon. Debug output was sufficient to achieve this.

For local and remote job execution, there were two aspects of validation. To test functionality, the debug output of all running programs was compared against the expected sequence of events. To test performance, the times written into the statistics files were compared against each other and against a stopwatch for "reasonableness". The same runs were used to test functionality and performance.

3.4.2. Test Runs

Three lab machines were used for the tests: walnut, balsa and mimosa. Each was supplied with the executables and data required for any experiment run. Each machine was configured to send remote jobs to the other two and no others. For all test runs, walnut was the only machine to generate jobs for the system.

The first two tests were designed to validate the information distribution, export decision and target selection functions for the *shortest* and *threshold* algorithms, respectively. For these tests, the job generator was made to start up 7 jobs, sleeping 1 second between them. The job duration was kept fixed at about 50 seconds and the threshold was set to 2. This guaranteed that virtually every aspect of both functions would be exercised, since a job would be generated under all the following conditions: (1) zero local load, (2) local load one below threshold, (3) local load at threshold -- polling required, (4) zero polled load -- polling aborted (shortest algorithm), (5) polled load one below threshold, (6) polled load at threshold -- multiple polling required (threshold algorithm), (7) both polled loads nonzero and below threshold -- first selected (shortest algorithm), (8) one remote load at threshold and the other below threshold -- choice required, and (9) both remote loads at threshold -- local run after polling required.

The third test was intended to test local and remote job execution. The job generator was modified to run 2 jobs with the same duration as before, this time waiting for each to complete before creating the next one, and to set the local load equal to the threshold after the first job. This would cause the first job to run locally and the second remotely, even though there was nothing else going on in the system.

3.4.3. Results

The first test (valid1.2) ran without errors. (A previous attempt at this test (valid1) used incorrect parameters). For each job that polled for load information, the load received upon each request (as given by the job statistics file), was the same as the load printed out by the lb daemon when the associated response was made. Since knowledge of each change in load is recorded in the job stats file, it was also possible to verify that the load numbers provided did indeed equal the number of jobs on that node at that time. This validated the information distribution function.

Jobs 1 and 2 were both started when the local load was below threshold, and they were both run locally without polling. The remaining jobs were all started when the local load was at or exceeded the threshold. For each of these, polling was initiated and the destination on which the job run was determined from the results of that polling. This validated the export decision (all algorithms).

Job 3 polled one node, mimosa, found the load to be zero, aborted polling, and sent the job there. This verifies that the zero load optimization of the shortest algorithm works. Job 4 also polled mimosa, found the load to be one, polled balsa, found its load to be zero and sent the job to balsa. Job 5 polled both nodes (balsa first), found both loads to be one, and sent the job to balsa. Job 6 polled both nodes (mimosa first), found balsa with load 2 and mimosa with load 1, and sent the job to mimosa. In all of these cases, the node with the lower load was selected. Job 7 polled both nodes (balsa first), found both had load two, and ran the job locally, which is what the algorithm demands. This validated the *shortest* algorithm.

The second test (valid2) also ran without errors. Jobs 1 and 2 ran locally, as expected. Job 3 polled mimosa, found a zero load and sent the job there since the load was below threshold (the only criterion for this algorithm). Job 4 also polled mimosa, found the load at 1 (still less than threshold) and sent the job there. Job 5 polled balsa, found a zero load and sent the job there. Job 6 polled mimosa, found the load at 2, polled balsa, found the load to be 1, and sent the job to balsa. This verified that the algorithm would continue after finding an unacceptable node. Job 7 polled balsa, found the load to be 2, polled mimosa, found the load there to be 2 also and ran the job locally. This validated the *threshold* algorithm.

The jobs numbered 7 in both of the preceding tests both reached a situation where they had run out of nodes to poll for an acceptable recipient for the job. They had run into a hard polling limit of two even though the algorithmic limit was set to three in the parameters file. The behavior of the software in the face of both these limits was made identical by having each daemon, at startup time, adjust the polling limit to take into account a small number of nodes. The job statistics files and the logs of the test runs both reflected this. Since the experiments involved more than 4 nodes, this adjustment was not made by the daemons during the experimentation phase of the thesis project.

The third test was run twice (valid3 and valid3.2) to account for an apparently unreasonable discrepancy between the total elapsed time and the total cpu time for the first job, run locally, and also to get a better precision on the "stopwatch" elapsed time for the jobs. Job 1 (local) had an elapsed time of 54.45 seconds and a cpu time of 49.90 seconds while job 2 (remote) had an elapsed time of 51.15 seconds and the cpu times summed to 49.32 seconds. It is unreasonable, in general, for a local job to have more cpu time expended than a remote job of the same duration. Also, it seemed difficult to account for the difference between elapsed and cpu times for job 1. It was assumed, therefore that the first job required more overhead activity (e.g. paging) than the second. Also, the executables being run contained full debugging symbols, making them about 3.5 times as large as their stripped equivalents (over 200k bytes unstripped). This might cause an inordinate amount of paging input, and hence the discrepancy between elapsed and cpu times. So, to remove these effects, the executables were stripped, and three jobs were run, the first two locally. Then only the last two jobs were used in the evaluation.

The results were indeed much better. Job 2 had an elapsed time of 49.85 seconds and a total cpu time of 49.6 seconds. Job 3 had an elapsed time of 49.63 seconds and cpu times totalling 49.38 seconds. The stopwatched times for the two jobs were 50 seconds each. All these numbers show very good correlation and seem quite reasonable. That the remote job had a slightly smaller elapsed time is a bit strange, but the difference is small enough (about 0.45%) to be attributed to scheduling quirks. It was also learned that project software overhead for job 2 was about 0.05 cpu seconds and for job 3 was about 0.93 cpu seconds where polling cost about 0.07 cpu seconds and transfer cost about 0.75 cpu seconds. Elapsed time costs for polling and transfer of job 3 were, respectively, 0.28 seconds and 0.7 seconds. These all seemed quite reasonable.

The debug output of each job was used to verify the correct sequence of events. Job 2 first tested the local load against the threshold, then incremented the local load value, ran the job to completion, and finally decremented the local load value. Job 3, on the local node, tested the local load against the threshold, polled mimosa, sent a job message to mimosa, waited for the mimosa job (daemon spawned) to establish a TCP connection with the local node, sent job context information, received mimosa's acceptance of the job, waited for the results of the job, and exited. The remote side of job 3 received the job message, incremented its local load, established a TCP connection with the originator of the job, received the job context, send its acceptance of the job, ran the job to completion, decremented its local load, and sent the job results back to the originator. Each of these was correct.

3.5. Utilities.

Several programs other than the two used for the actual load balancing were required in the running and analyzing of the experiments. These are divided into two categories and are described in the following two sections.

3.5.1. Data Reduction.

Two programs were written to analyze the data produced by the load balancing tests. The first, called *repn*, reduced the data on a node by node basis. This was used to get detailed information on specific jobs and failure conditions. It alone was used in analyzing the validation and calibration tests. The second, called *repe*, worked on data from all nodes in the experiment. This was used to calculate all results from the experimental data, except for the polling data, which was readily available from *repn*.

repn was capable of producing the following reports from the data that was saved on a given node: (1) aborts, a detailed dump of all information saved following fatal failures in any of the load balancing modules; (2) dump, a detailed dump of all data collected for a specific job, or for all jobs originating on the node on which the data was saved; (3) polls, a count of the number of times every other node was polled for load info by the data's node; and (4) standard, a table of important information about every job originating on the data's node. The polls report was used to extract the data for the probe rate chart in section 4.3.

repe generated two reports which which yielded all the remaining results in section 4.3. The first, called the stability report, produced a table of departures, arrivals and their ratio within a window which was slid across the total time of the experiment. This was used to determine when, if ever, the experiment reached a steady state. It also spelled out any errors or unusual conditions (such as timing anomalies) it detected. The results of the sliding window calculation was used to approximate the largest region of relative stability during the time of the experiment. The response report then calculated all desired information within that region. It calculated system load by accumulating the amount of time each processor was busy and dividing by the region size and by the number of nodes. That is,

$$\rho = \frac{\lambda}{\mu c} = \frac{\dfrac{\sum_{i=1}^{n} \text{service}_i}{n}}{\dfrac{\sum_{i=1}^{n} (\text{arrival}_{i+1} \quad \text{arrival}_i)}{n} c} = \frac{\sum_{i=1}^{n} \text{service}_i}{(\text{arrival}_{n+1} - \text{arrival}_1)c}$$

where ρ is utilization, μ is the average service rate, λ is the average arrival rate, c is the number of nodes and n is the number of jobs in the area of interest.

It also accumulated the elapsed time required for the completion of each job completely within the region and the number of such jobs; the ratio of these two gives the average response time of the experiment. Finally, the response report also calculated the number of transferred jobs and the number of failed transfer attempts (due primarily to TCP failures) over the entire time of the experiment.

Both the stability and response reports required the construction of an events list, which contained some information about each job arrival and departure as well as each load increase and decrease. These events were named START, FINISH, UP and DOWN, respectively. The timing anomalies mentioned above occurred when a job's UP event apparently occurred before its START or its FINISH before its DOWN. This was possible for remote jobs whose run node "saw" a slightly different experiment start time than its origination node. An approximate experiment start time was calculated for each node using

the send and receive times of the GO messages. The difference between the send time for the first GO message sent and the GO message for the node in question was subtracted from the receive time for the GO message. This would give an exact experiment start time in the absence of any variation in the transmission delay for the GO message among the nodes. In other words, if each GO message arrived exactly n seconds after it was sent, the approximation would have no error.

In order to deal with the detected anomalies, the START and FINISH times of the afflicted jobs were made equal to the UP and DOWN times, with the adjustments recorded. This allowed the events list to be constructed in a sensible fashion. When the elapsed time was calculated for an afflicted job, the adjustments were used to return the START and FINISH times to their original values. The only place besides this calculation and the events list creation in which these times were used was in the determination of which jobs were to be included in the region of interest. Thus it is possible that a job or job arrival might have been included in the region when it should not have been. This is a flaw, but since the boundaries of the region were quite approximate, it was not an important one.

3.5.2. Others.

Several other small programs were written to assist in the actual experimentation. *xseeds* generated all random number seeds for the experiments. A separate set of seeds was created for each node and was read by *xlbd* at the start of each experiment. This enhanced the reproducibility of each experiment. *xstrt* sent GO messages to all nodes participating in an experiment. *xpoll* sent load inquiries to all nodes to determine if they were active. *xstop* sent an END message to each node to halt *xlbd* after an experiment had completed.

3.6. Tools and Configuration

This project was implemented on a collection of AT&T Model 3B1 UNIX PC's, each with a 40 megabyte hard disk. These computers were networked via Ethernet.

Software development was accomplished using the AT&T UNIX PC Utility package which includes a C compiler, the make, lint, and other utilities. Standard UNIX filters and utilities were also used. The AT&T Transport Level Interface Library was used to implement communication among the computers.

4. Experiments.

To gather performance data on the four test algorithms, experiments were run on five of the UNIX PC's in the graduate lab. However, before these experiments could be configured, several parameters had to be determined. The first was the transfer cost. Since a goal was to duplicate [Eager]'s assumption that the average transfer cost would be about 10% of the average job duration, tests were run to determine the actual cost incurred by the software in moving a job to a remote node. The job duration was then fixed at ten times that amount. The system load was varied by varying the job arrival rate. Next, in order to calculate the response time of the jobs, the minimum response time had to be calculated as a function of the input job length. The response time of a job would then be the ratio of the actual elapsed time and the minimum response time for that job's length. Data from the transfer cost tests were used for this calculation. Also, to verify the assumption that probing costs are negligible, tests was run to measure probing costs directly. Finally, to account for any costs due to measurements taken during the jobs, a test was run without most of those measurements and the results compared against results of the transfer cost tests.

Section 4.1 goes into detail regarding these calibration tests, while sections 4.2 and 4.3 deal with the actual experiments.

4.1. Parameters.

To determine the parameters required by the experiments, several calibration tests were run on the UNIX PC's. These tests involved one local source node which would generate all the jobs in the test and zero or more remote nodes, which would only accept jobs for remote execution. The tests were run on unloaded nodes, one job at a time, so that differences in elapsed times could be used as a measure of various costs. Any effects due to external activity on the systems were eliminated as described in the following paragraph. A constant job size of 20000 bytes, a threshold of 2, a probing limit of 3, and a transfer limit of 1 were used for all calibration tests, except as noted. These parameters are identical to those used in the main experiments.

4.1.1. Transfer Costs.

In [Eager], a transfer cost of 10% of the average processing cost of the jobs was used. In order to maintain this ratio, the cost to transfer a job to a remote node had to be calculated so that an average job length could be identified. Jobs were run locally and remotely on unloaded systems over a broad range of job lengths with the difference in elapsed times between local and remote representing the transfer cost. It was expected that the transfer cost would be independent of job length, so the mode value of that difference could be used as transfer cost. This did not, however, turn out to be the case.

The job source program was configured to run local and remote jobs alternately at each job length and to repeat the entire sequence of job lengths five times. A daemon process (*xlbd*) was started on both the source and destination machines and the job source process (*tester*) was started on the source machine. The test was then started by running a program (*xstrt*) on the source machine which sent a GO message to each of the machines. When an amount of time sufficient to complete all the jobs had elapsed, the daemons on each machine were probed (by running *xpoll* on the source machine) to verify that all activity had ceased. The shell which ran all these programs was an *rlogin* process and was alive during the entire test.

An interesting phenomenon was observed during this calibration test. Local jobs required about 5% more cpu time than remote jobs of the same length. There was no keyboard or screen activity during the tests and both *xlbd* and *tester* execute the setpgrp() system call as part of their initialization. This effect was not seen when the procedure was changed so that the shell that started *tester* was terminated and *xstrt* was run from a third machine. This presumably demonstrates that some cpu overhead is required for dealing with a process's control terminal (polling for input?), at least when that control terminal is a remote login pseudo-tty.

With the corrected procedure, test xfer_cost_13 was run at 3:26 am, on Sunday, 12/3/89 with results which were surprising in that they showed a large amount of variation in elapsed time between remote jobs of the same length. To attempt to explain this, the difference between elapsed time and total CPU time (hereafter called the *differential*) was calculated for each job so that activity external to the test might be detected. Quite a bit of activity was detected among the remote jobs. (The differentials of the local jobs were very consistent.) So a second test (xfer_cost_14) was run at 12:41 am on Tuesday, 12/5/89. Also, two additional tests (xfer_cost_15 and xfer_cost_16) with a transfer size reduced to one byte were run at 12:41 am Wednesday, 12/6/89, and at 2:25 am Thursday, 12/7/89 so that effects due to network traffic might be minimized.

The following graphs show statistics about the remote job differentials at each job length for the 20k byte tests and the one byte tests, respectively. Specifically, the low value at a given length; the median, mean and high values; and the standard deviation are plotted. They illustrate the high degree of variation in the differential, showing that there was indeed much activity external to the tests, despite the time of day during which they were run. In fact, the standard deviation of the differential samples is typically larger than the lowest sample and in some cases approaches the median.

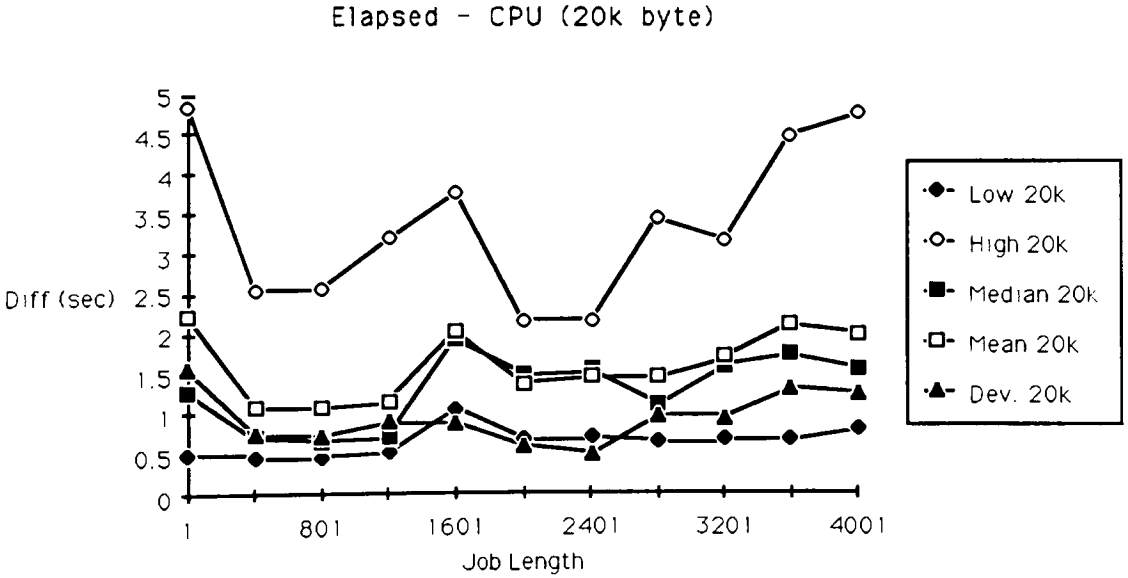


Figure 4.1. Differential Statistics for 20k-byte Remote Jobs

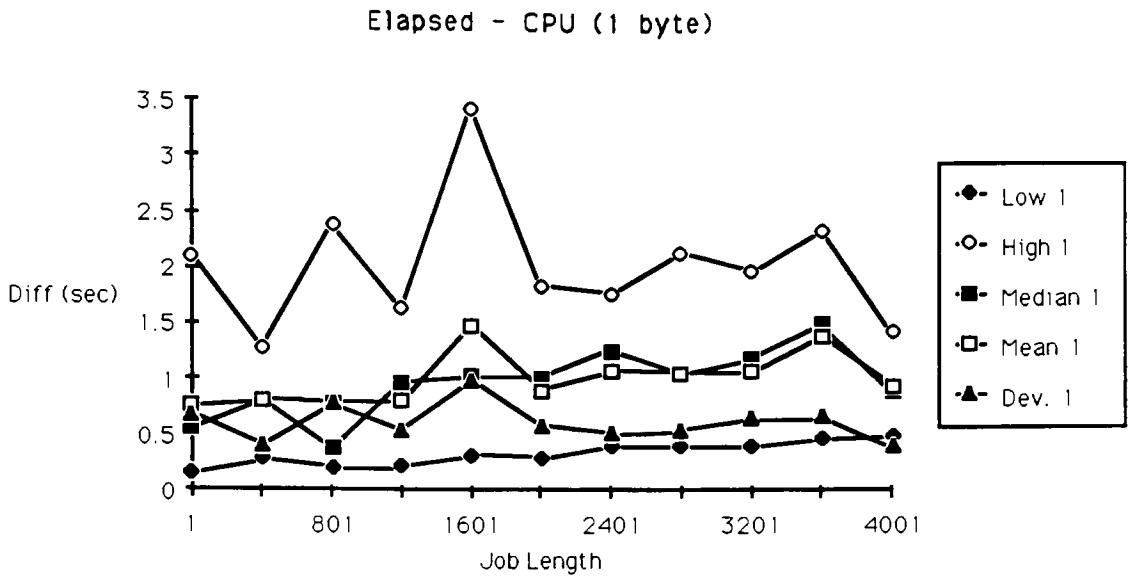


Figure 4.2. Differential Statistics for 1-byte Remote Jobs

Note also that the plots of low differentials at each job length display much less variation than the other plots. This indicates that using job samples with minimum or close to minimum differential is desirable for calculating the transfer costs, since external activity should be minimal for those jobs. The next graph shows only the "Low" plots of each of the previous two graphs with least-squares line estimates.

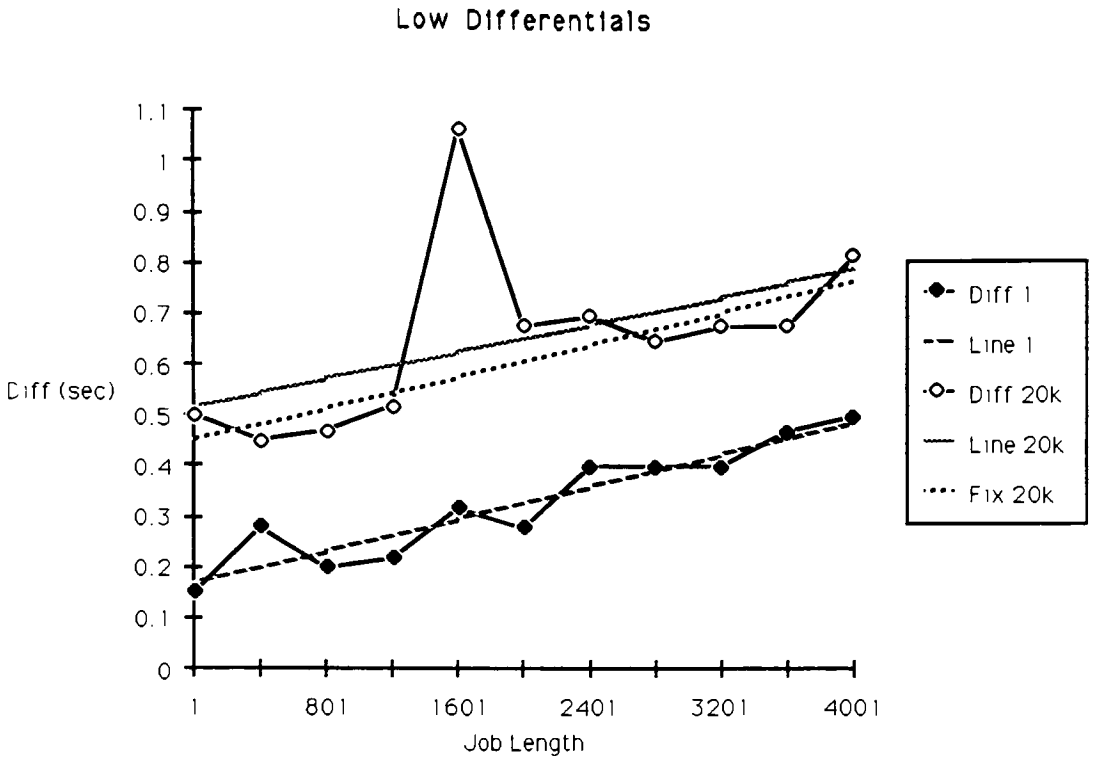


Figure 4.3. Minimum Differentials.

An important observation is that there is something wrong with the 20k byte samples at length 1601. It is likely that all samples at that job length experienced delays due to external activity. Transfer costs calculated for the 20k byte jobs at length 1601 will be neglected because of this activity. The grey line is the line fit including the 1601 sample and the dotted line is the line fit without that sample. The dashed line is the line estimate for the 1-byte remote jobs.

A second observation is that delays increase with job length (note the positive slope of all line estimates). Scheduling overhead or other periodic system activity might account for this.

A last observation is that the slope of the corrected 20k byte line estimate and that of the 1 byte line estimate are close to identical (.0000802 and .0000807 seconds, respectively). Thus the cause of the increased delays due to job length is independent of the job size (number of bytes transferred).

At this point, a somewhat arbitrary decision was made to calculate the transfer cost at any job length as the difference between the smallest elapsed time remote job sample and the smallest elapsed time local job sample. One reason for this decision was that in all cases, the differential of the sample with the smallest elapsed time was within .05 seconds of the smallest differential at that job length. Another reason was that variations in the total cpu time among the samples at any given job length were observed, and it is conceivable that these variations were caused by extra scheduling due to competition for the cpu. Lastly, there was a slightly better line fit to the results when the minimum elapsed time was used.

The next graph shows the calculated experimental transfer costs of 1 byte and 20k byte jobs, line estimates for both cases, and sample distance from line estimates (a measure of variation). As with the differential graph, two line fits were drawn for the 20k byte jobs transfer cost, one including the results at length 1601 (dashed line), and one neglecting them (dotted line). On the average, data points (excluding 1601) were within 5.0% of the line including the 1601 result and within 4.5% of the line neglecting that result.

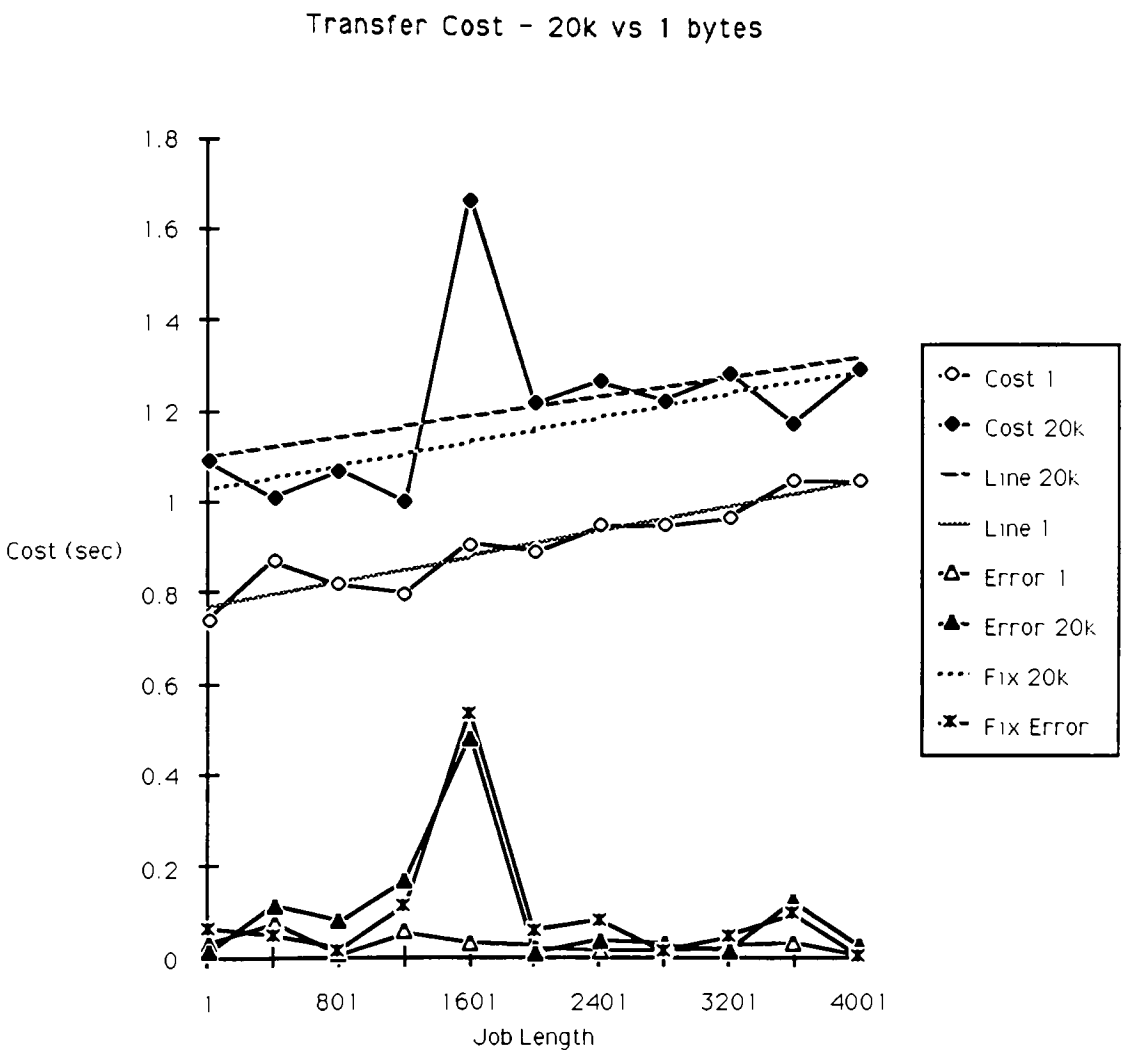


Figure 4.4. Transfer Costs.

4.1.2. Minimum Response Time.

The tests run to determine transfer costs were sufficient to generate a formula for the minimum response time. No other test runs were required.

The following graph shows data and least squares line approximations for both minimum and median elapsed times of locally run jobs. The resulting formula follows that graph.

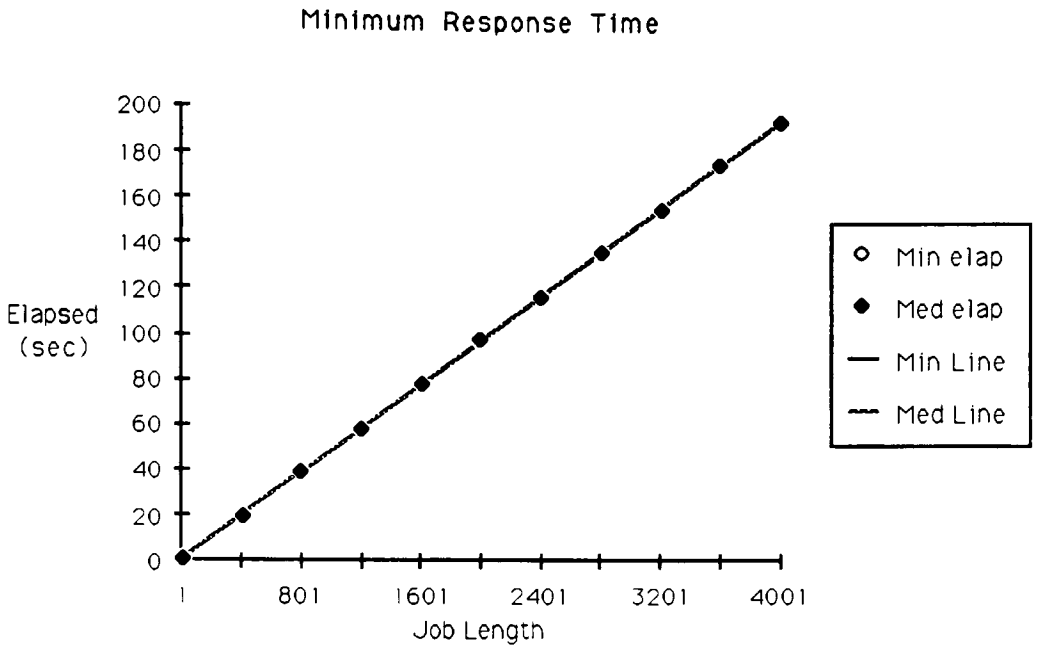


Figure 4.5. Required Service Time vs. Job Length Parameter.

Thus the formula for minimum response time is

$$\text{min RT(len)} = .04848*\text{len} + .2288$$

4.1.3. Data Recording Costs.

In order to determine the effect that taking measurements on the jobs would have on their performance, most of the *times()* calls and their associated data structures (to hold the results) were compiled out (the -DFULLSTATS flag was removed from the compile command) and 20 jobs of length 1 were run of which 10 were local and 10 were remote. These performance of these jobs was compared with that of corresponding jobs in the transfer cost tests. The difference in total elapsed times was on the order of 1/20 of a second, which is negligible compared to an average job duration of 12 seconds and is small compared to a transfer cost of 1.2 seconds.

4.1.4. Probing Costs.

To calculate the cost of getting a remote load's node, 2 techniques were used in tandem. First, a *times()* call was made before all probing and once after each probe. This gave an explicit measurement of probing costs. Second, the total elapsed time of jobs which ran locally but probed first was compared with jobs which ran locally but without probing. This gave a gross estimate of the effect of probing on job response time. 10 of each type of job were run and the minimum sample of total elapsed time was used.

The first run was made with the maximum probes constant, *MAXPOLLS*, set to 3 when 5 was desired. The second run showed up a bug in the targeting module which caused some of the jobs to run remotely. The third run completed without error and gave the following results. The first probe completed in about 13 clock ticks (a clock tick is about 1/60 of a second) while the remainder completed in about 2.5 ticks. This is because the first probe includes the overhead of opening and setting up the udp port which is used for the communications. Adding these together gives a total probing cost of about 23 ticks. The total elapsed time for the jobs with probing was about 29 ticks while total elapsed time for jobs without probing was about 9 ticks, which means probing had an effect of about 20 ticks on total elapsed time. Strangely, this 3 tick difference is completely accounted for in the elapsed time required for the actual running of the job, which was 9 ticks without probing and 6 ticks with probing. It is a mystery why probing should reduce the elapsed time of a job. Perhaps some paging or other I/O occurs during probing which, if there was no probing, would occur during the actual job execution.

In any event, roughly half of the cost of probing, about 10 clock ticks, is caused by the establishment of a local udp port through which communication is done. If the job is to be transferred, then this cost must be borne anyway, and so becomes part of the transfer cost. The remaining 10 to 13 ticks, about 1/5 of a second, is less than 1/100 of the average job duration. This is considered negligible. If the job is not transferred, then the cost is about 1/50 of the average job duration which is also very small. Costs such as these should have very little effect on the performance of the system.

4.2. Test Runs.

To measure the performance of the system with the 3 test algorithms and the no load-balancing control case, tests were run on five of the UNIX PC's in the GCSD lab. Although a few more UNIX PC's exist in the lab, five was chosen since that number should have always been readily available and since lab assistance would not be available during the times of day in which the tests were run. This number is well below the lower limit of applicability of twenty nodes given in [Eager 86], so results differing from that work would not be surprising.

The procedure described in the following section was executed twice (2 trials) at applied loads of 0.5, 0.7 and 0.9 for each algorithm: a total of 24 test runs. While the arrival and service times were set to the appropriate values for a given applied load, the system utilization (load) was also measured and it was the measured value that was used in compiling the results.

4.2.1. Procedure.

For each test run, parameter and random number seed files were distributed to all test machines. If a different algorithm was being run than was run in the previous test, the appropriate executables (*jobsrc* and *xlbd*) were built and distributed as well. These executables were then started up via remote login from ma. At this point, the test machines were examined for external activity. When it was determined that no other external processes were active, and when no other active users were logged into ma, the experiment was then started by running xstrt on one of the machines, again via *rlogin* from ma. The *rlogin* session to the test machine and the login session on ma were then terminated.

After waiting a sufficient time to ensure that the experiment had completed, *xpoll* was run to verify the experiment had ended and *xstop* was run to halt all the daemons. The report generator *xrepn* was also run on each test machine to produce some preliminary reports. These reports and all other files relevant to that experiment were then copied to ma. All files were later copied to a local UNIX PC for more complete data reduction.

4.2.2. Test Observations.

Because of UDP's relatively high packet loss rate on the 3b1's, synchronizing the start of the test required sending 2 GO messages. Since the time of the send was included with the message, a relatively accurate (within about 0.2 seconds) synchronization of experiment events among the nodes was still possible. The only place this was important was in the creation of an events list, where job arrivals and departures were expected to occur before and after incrementing and decrementing the load, respectively. Here, when a finish time later than the load decrement time was detected, the finish time was simply adjusted to equal the load decrement time with the adjustment being noted. When the response time was calculated, this adjustment was just subtracted off again.

Many of the jobs from the no load balancing runs showed response times less than the minimum previously calculated (min_RT). Since the min_RT function is intended to reflect the minimum response time possible for any given job length (that is, its expected service time), this required that min_RT be adjusted to accommodate the new data. The new graph is given below.

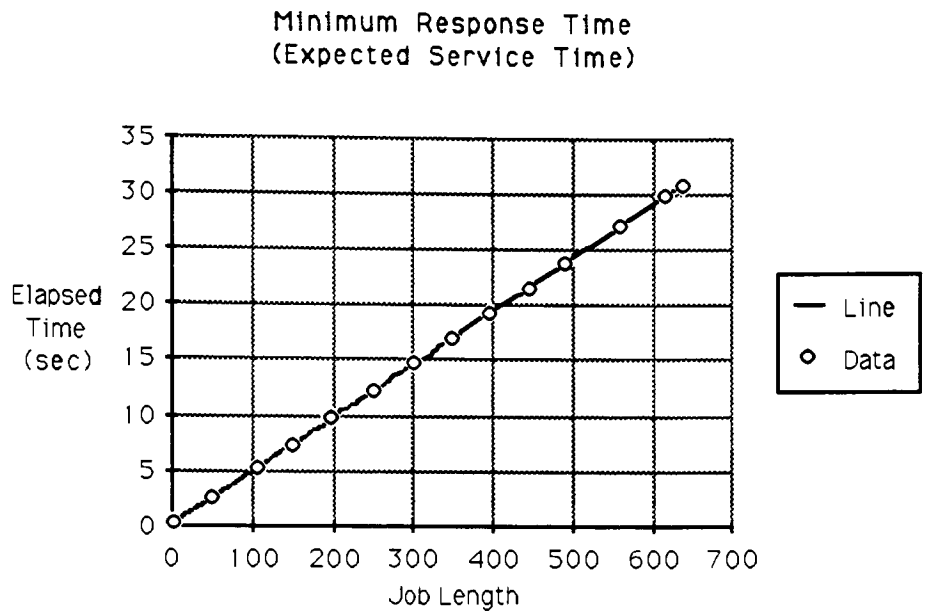


Figure 4.6. Modified min_RT function.

Several problems with the software were detected in running the tests. The most severe was one which fortunately occurred only once. This was caused by a TCP failure which occurred during the remote exec simulation and resulted in the load being calculated improperly on the node running that job. The test run showing this error was discarded.

Also, several abort files (lbdmp*) were the wrong size. This discrepancy has not been resolved, but is not believed to have effected the test results since all jobs were accounted for. Finally, two or three PC crashes caused the loss of a few statistics files. Where required for at least one datum at a given utilization, tests were re-run.

5. Results

The results of the experimentation are shown in the two contexts described in the Introduction: the complexity analysis provided by [Eager 86] and the startup criteria analysis described in section 2.2. The method by which each of the results was calculated is described in section 3.5.1 on data reduction.

5.1. Complexity

The primary results are shown in figure 5-1 below which is a graph of average response versus system utilization. That graph is quite similar to its analog in [Eager 86] (figure 2 in that paper); the similarities and discrepancies will be discussed later in this section. The M/M/1 and M/M/5 lines are from the formulas described in section 2.1.4. The parameters in the title of the graph are transfer cost, threshold, probing limit and transfer limit, respectively. These were set to the same values as in [Eager 86].

Note that there is a smaller range of possible improvement between the M/M/1 curve and the M/M/5 curve than between the M/M/1 and M/M/20 curves as given in [Eager 86]. Even so, a marked improvement is observed for all three algorithms over both M/M/1 and the no load balancing control test run. Also, all three algorithms show very similar performance benefits up to a load of about 0.7. These results are very similar to the results in [Eager 86].

Note also that the data points at loads 0.82 and 0.91 for the control no load balancing case ("No lb") fall below the M/M/1 queue line. This should not be possible since the test case includes costs for cpu scheduling and any external activity occurring during the experiment which are not considered in the simple M/M/1 model. However, figure 5-2 shows the frequency of fatal errors[1] which occurred in each of the tests. There appears to be a correlation between these errors and the discrepancy seen in the control case.

The load balancing cases exhibit a downward or rightward shift at loads above 0.8 as well, especially the threshold and shortest results at 0.97 which are below the optimum M/M/5 line. In these cases, a large amount of probing (see figure 5-3) and a reduced number of transfers (see figure 5-5) were observed. Failures in probing and transferring appeared to be roughly proportional to the frequency of each (see figures 5-4 and 5-6). For the threshold and shortest results at 0.97 utilization, a large number of fatal errors was also observed (see figure 5-2).

---

1 A fatal error is defined as one which precludes the completion of the job. All fork() system call failures were fatal errors in the experiments and were the only fatal errors.

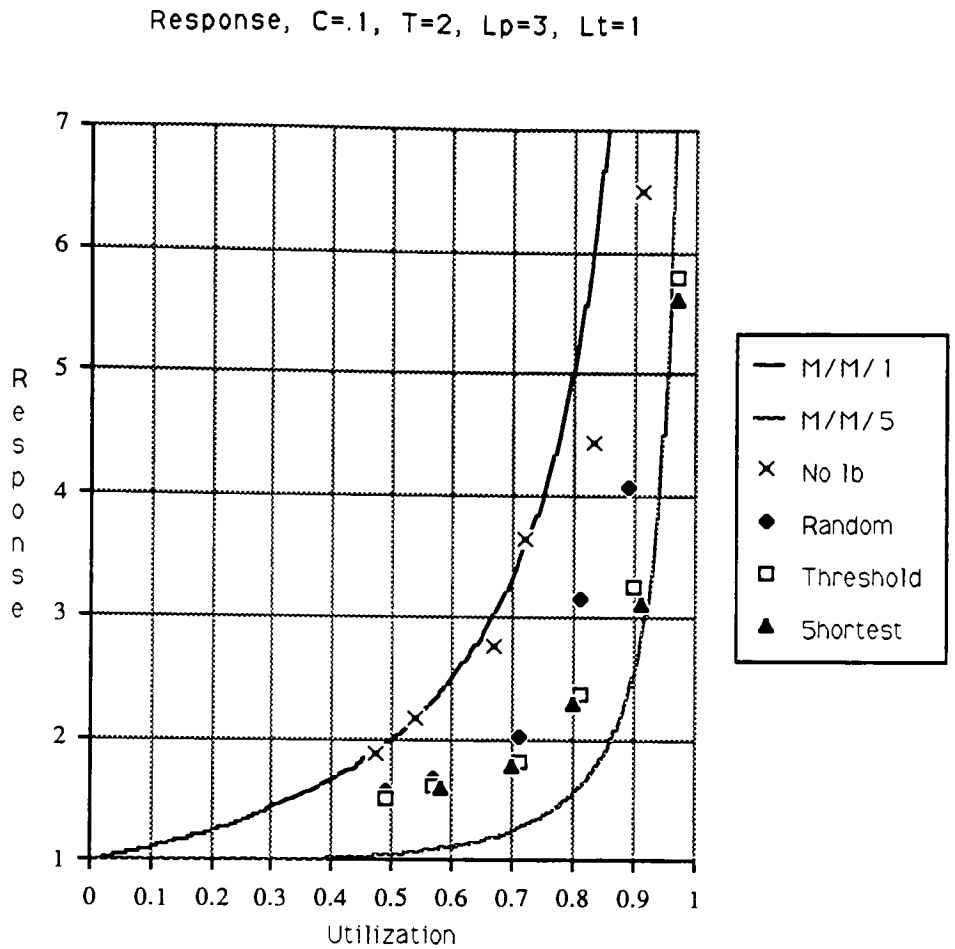


Figure 5-1. Complexity Response Results.

Also, the stability graphs in Appendix D show a much higher variation in arrival-to-departure ratio at an applied load of 0.9 than they do at 0.5 or 0.7. This may show, in part, that the sample space of jobs may have been too small for accurate statistics at the higher load. However, it also exhibits the effects of increased fatal errors. That is, jobs arriving in the window which would have normally departed outside the window, now depart in the window because of a fork() failure. This may also show that the input arrival and service rates were inaccurate enough that the effective arrival rate within the window was greater than the effective service rate, causing an unstable situation.

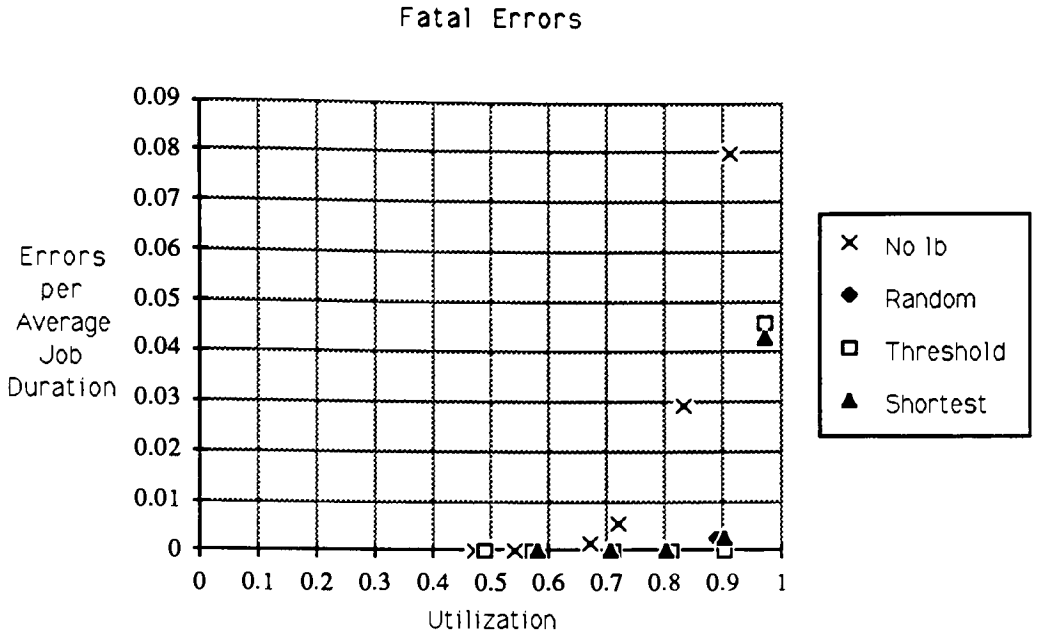


Figure 5-2. Frequency of fatal errors.

Finally, the probe and transfer rate graphs (figures 5-3 and 5-5) are also very similar to the analogous graphs in [Eager 86]. The graphs here do, however, show slightly lower values. This may be because the time of the entire experiment is considered in calculating these values, so edge effects near the beginning and end of the experiment tend to produce lower values than would normally be expected. Note that the graphs in [Eager 86] are for an average job service time of 1 second. In order to get analogous numbers here, it was necessary to normalize the rate to the average job duration. In other words, instead of transfers per second, here we have transfers per $1/\mu$. This is given by total transfers (or probes or errors ...) times the utilization divided by the number of jobs.

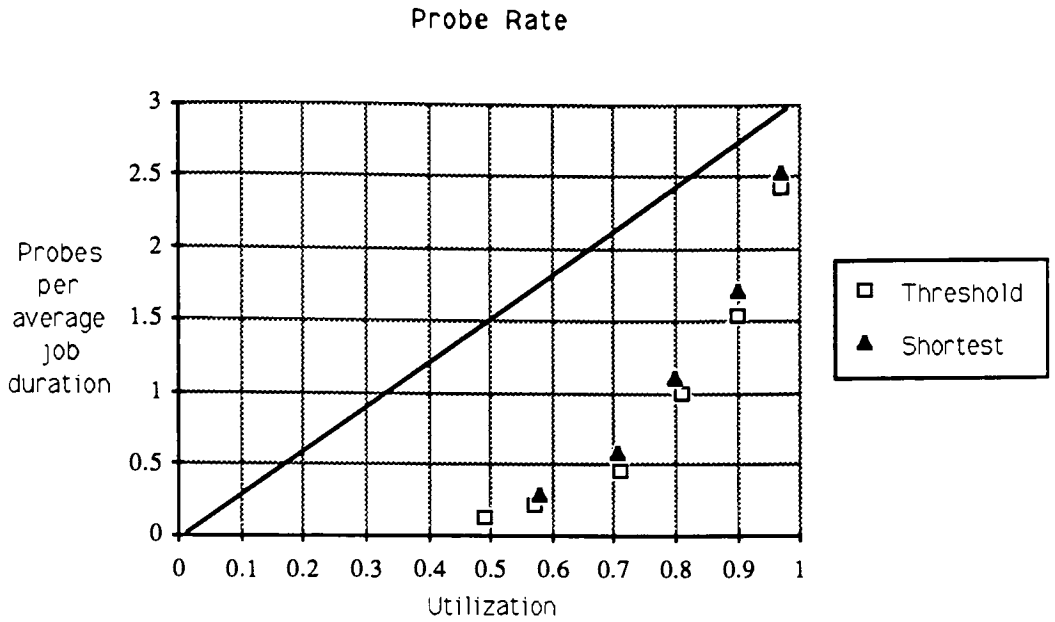


Figure 5-3. Frequency of probes for load info.

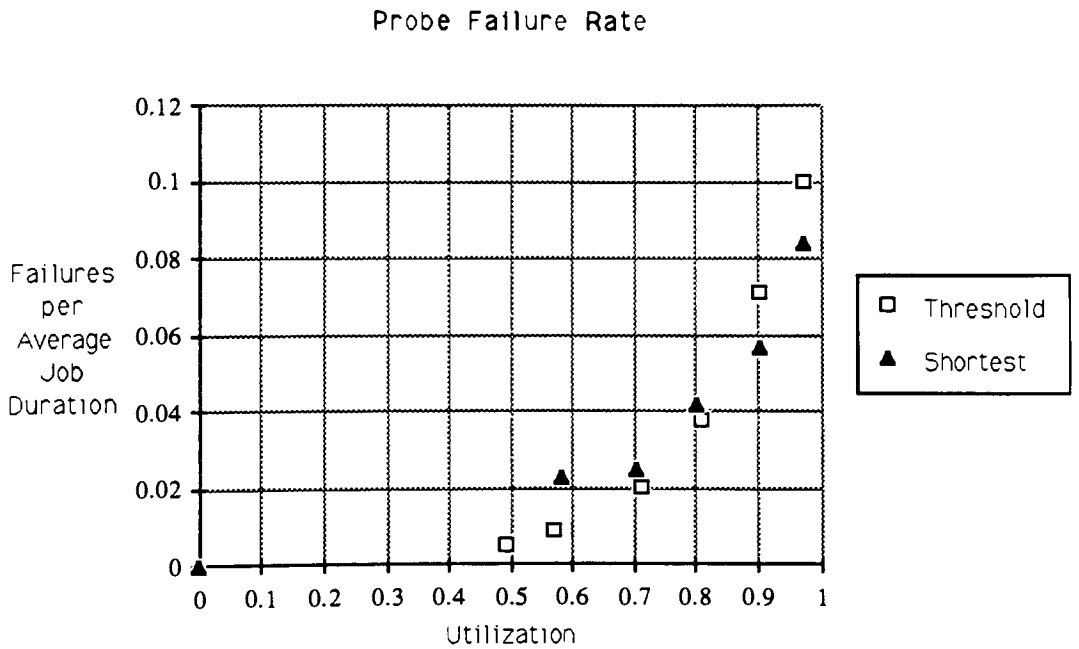


Figure 5-4. Frequency of probe failures.

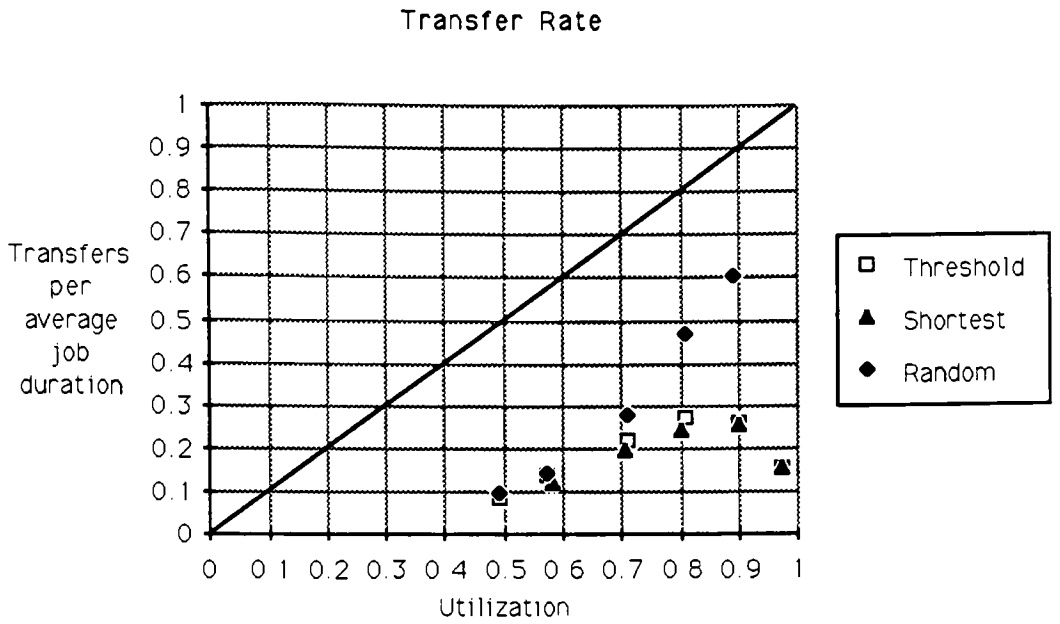


Figure 5-5. Frequency of job transfers.

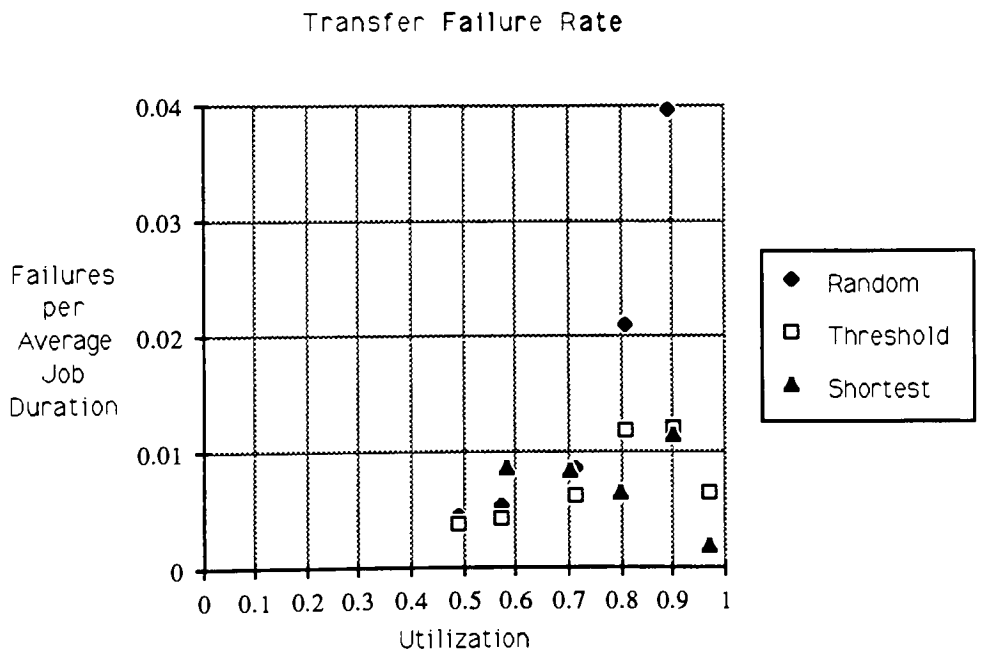


Figure 5-6. Job transfer failures.

5.2. Startup Criteria

This experiment and the results in [Eager 86] clearly indicate that load balancing should be more effective at a load of 0.7 than at a load of 0.5, for example. In fact, by looking at the response graphs, one might estimate roughly 3 times more improvement at 0.7 than at 0.5. The graph in figure 5-7 plots all three predictor formulae calculated in the proposal; they happen to lie one on top of the other for the experimental parameters used in the tests here (5 nodes, a quantum of 1/60Hz and a job arrival rate of about 1/24 jobs/second).

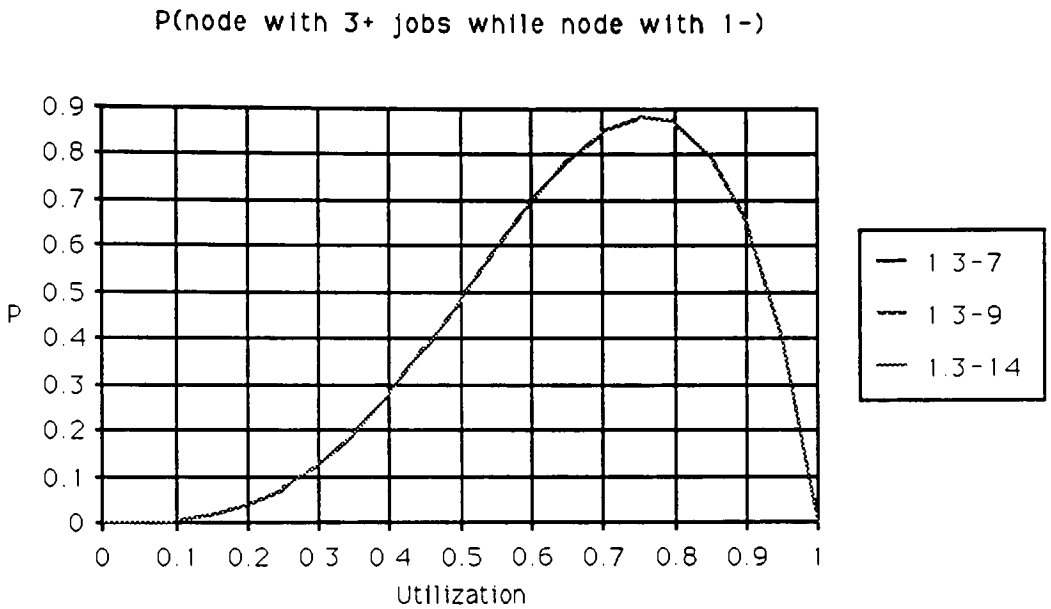


Figure 5-7. Startup criteria formulae evaluations.

One would expect these formulae to have the same general shape as the transfer rate plots for the threshold and shortest algorithms in figure 5-5 above, since one expects many transfers when the probability of unbalanced nodes is high. The graphs are quite similar in shape, although the startup criteria curve peaks earlier than the transfer rate curves.

6. Conclusions

6.1. Complexity

For utilizations at or below 0.7, the results obtained here tend to support the conclusions in [Eager 86] that substantial performance improvement is achievable with little or no system state information. The one questionable data point for the random algorithm at a utilization of 0.81 also lends some support to the contention that at least some system state information (threshold-type) substantially increases performance at higher utilizations. Comparing the experimental results at utilization 0.9 for the load balancing algorithms to the no load balancing case also provides significant evidence that limited state information greatly improves performance at high loads.

The lack of any substantial difference between the performances of the Shortest and Threshold algorithms at any load also supports their conclusion that using "threshold-type" information "obtains essentially all of the benefit available through [load sharing policies that employ threshold transfer policies]"[1]. This remains true even though this implementation included the optimization for the Shortest algorithm given in [Eager 86].

To the extent that the results of this work support the above two conclusions, it also extends them to a network of as few as five nodes.

In addition to the response results, the similarities between the Probe and Transfer Rate graphs presented here (figures 4.9 and 4.11) and those in [Eager 86] (figures 7 and 8) suggest that this implementation of the policies spelled out in that paper is behaving in a substantially correct manner.

Finally, the relatively high variation in the stability graphs in Appendix D at high applied loads as well as the dubious response results at high calculated loads casts doubt as to whether enough samples were taken at these loads. Recall, however, that the system load was calculated by dividing the time interval between the first arrival in the window and the first arrival after the window by the window size minus all zero load times and by the number of nodes (see the discussion in section 3.4.1). It is possible that in attempting to achieve an applied load of 0.9, the parameters were set such that the arrival rate could be higher than the service rate for substantial periods of time. Such an unstable situation could cause the total absence of zero load times, in which case the utilization calculation could conceivably exceed 1. This suggests that this method of calculating the utilization is flawed for very high loads and hence could be the cause of the apparently misplaced data points at the high end of the response graph.

6.2. Startup Criteria

All three predictor formulae evaluate to the same plot in figure 4.13 and all seem to predict fairly well when load balancing would be effective in the sense that the utilizations for which high probabilities of load imbalance are predicted are the same ones in which many jobs are transferred in the actual system. Given the availability of an ongoing measure of system load, using any of the formulae on that load to determine when to turn on load balancing should be effective.

---

1 Page 667 in [Eager 86]. Recall the transfer policy decides whether a job should be transferred (is the local load over threshold?)

References

[Barak 85a] A. Barak and A. Litman, "MOS: A multicomputer distributed operating system", *Soft. Pract. Exp.*, vol. 15, pp. 725-737, Aug. 1985.

[Barak 85b] A. Barak and A. Shiloh, "A distributed load-balancing policy for a multicomputer", *Soft. Pract. Exp.*, vol. 15, pp. 901-913, Sept. 1985.

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

[Chou 82] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems", *IEEE Trans. Softw. Eng.*, vol. SE-8, pp. 401-412, July 1982.

[Chow 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. 354-361, May 1979.

[Chu 80] W. W. Chu, et. al., "Task allocation in distributed data processing", *Computer*, vol. 13, pp. 57-69, Nov. 1980.

[Eager 86] D.L. Eager, et. al., "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Trans. Softw. Eng.*, vol. SE-12, pp. 662-675, May 1986.

[Efe 82] K. Efe, "Heuristic models of task assignment scheduling in distributed systems", *Computer*, vol. 15, pp.50-56, June 1982.

[Freid 86] C. B. Freidlander and H. F. Wedde, "Distributed processing under the Dragon Slayer operating system", *Proc. 1986 Int. Conf. Parallel Proc.*, pp. 250-257.

[Ghafo 86] A. Ghafoor and R. Inamdar, "A dynamic task scheduling algorithm for symmetric and homogeneous distributed systems", *Proc. 10th COMPSAC*, pp. 50-56, 1986.

[Hwang 82] K. Hwang, et. al., "A Unix-based local computer network with load balancing", *Computer*, vol. 15, pp. 55-66, April 1982.

[Jin 87] Yudan Jin, "Load balancing issues in distributed operating systems", Class paper in ICSS 811, Rochester Institute of Technology, Summer 1987.

[Jones 79] A. K. Jones, et. al., "StarOS, a multiprocessor operating system for the support of task forces", Proc. 7th Symp. Oper. Syst. Princ., pp. 117-127, Dec. 1979.

[Juang 86] J.-Y. Juang and B. W. Wah, "Global state identification for load balancing in a computer system with multiple contention buses", *Proc. 10th COMPSAC*, pp. 36-42, 1986.

[Klein 64] L. Kleinrock, *Communication Nets: Stochastic Message Flow and Delay.* McGraw-Hill, 1964. Pp. 84-91.

[Livny 82] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems", *Proc. Model. Perf. Eval.*, pp. 47-55, 1982.

[Ma 82] P.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems", *IEEE Trans. Comput.*, vol. C-31, pp. 41-47, Jan. 1982.

[Mirch 86] R. Mirchandaney and J. A. Stankovic, "Using stochastic learning automata for job scheduling in distributed processing systems", *J. Parallel & Distrib. Comput.*, vol. 3, pp. 527-552, Dec. 1986.

[Ouste 80] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: an experiment in distributed operating system structure", *Comm. ACM*, vol. 23, pp. 92-105, Feb. 1980.

[Ross 72] S. M. Ross, *Introduction to Probability Models.* Academic Press, 1972.

[Stank 85] J. A. Stankovic, "An application of Bayesian decision theory to decentralized control of Job Scheduling", *IEEE Trans. Comput.*, vol. C-34, pp.117-130, February 1985.

[Stank 84a] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms", *Computer Networks*, vol. 8, pp. 199-217, June 1984.

[Stank 84b] J. A. Stankovic and I. S. Sidhu, "An adaptive bidding algorithm for processes, clusters and distributed groups", *Proc. 4th Int. Conf. Distrib. Comput. Syst.*, pp. 49-59, May 1984.

[Tijms 86] H. C. Tijms, Stochastic Modelling and Analysis: A Computational Approach. John Wiley & Sons, 1986.

[Wah 85] B. W. Wah and J.-Y. Juang, "Resource scheduling for local computer system with a multiaccess network", *IEEE Trans. Comput.*, vol. C-34, pp.1144-1157, Dec. 1985.

[Witti 80] L. D. Wittie and A. M. van Tilborg, "MICROS, a distributed operating system for MICRONET, a reconfigurable network computer", *IEEE Trans. Comput.*, vol. C-29, pp. 1133-1144, Dec. 1980.

Appendix A. Idle Resource Calculation

The first formula presented in [Livny 82] suffers from a typographical error. We calculate the correct formula here and use it in sections 1.2 and 1.3. It is charted for various values of n in Figure 1-7.

That formula in [Livny 82] starts with

$$P_{wi} = \sum_{i=1}^{n-1} \binom{n}{i} Q_i H_{n-i}$$

where $Q_i = P_o^i$

and $H_i = (1-P_o)^i - (P_o(1-P_o))^i$

Note that we can take the product $Q_i H_{n-i}$ and split it into the difference of two products, so that P_{wi} can be expressed as the difference of 2 sums.

$$P_{wi} = \sum_{i=1}^{n-1} \binom{n}{i} P_o^i (1-P_o)^{n-i} \quad \sum_{i=1}^{n-1} \binom{n}{i} P_o^i (P_o(1-P_o))^{n-i}$$

Now we use the binomial expansion theorem on both sums and subtract the i=0 terms to get

$$P_{wi} = (P_o+1-P_o)^n \quad (1-P_o)^n \quad [P_o+ P_o(1-P_o)]^n + P_o^n(1-P_o)^n$$

which reduces to

$$\boxed{P_{wi} = 1 - (1-P_o)^n(1-P_o^n) - [P_o(2-P_o)]^n}$$

This is the formula which is used in Figure 1-7. It matches Figure 1 of [Livny 82] much more closely than a similar chart of the formula given in that paper. You can see this by comparing Figure A-1 with the Figure 1 in [Livney 82]. The corrected formulae produce the same curves as shown in Figure 1 while the formulae as given in that paper are off for smaller ρ. (The corrected formulae curves are those whose markers are black.)

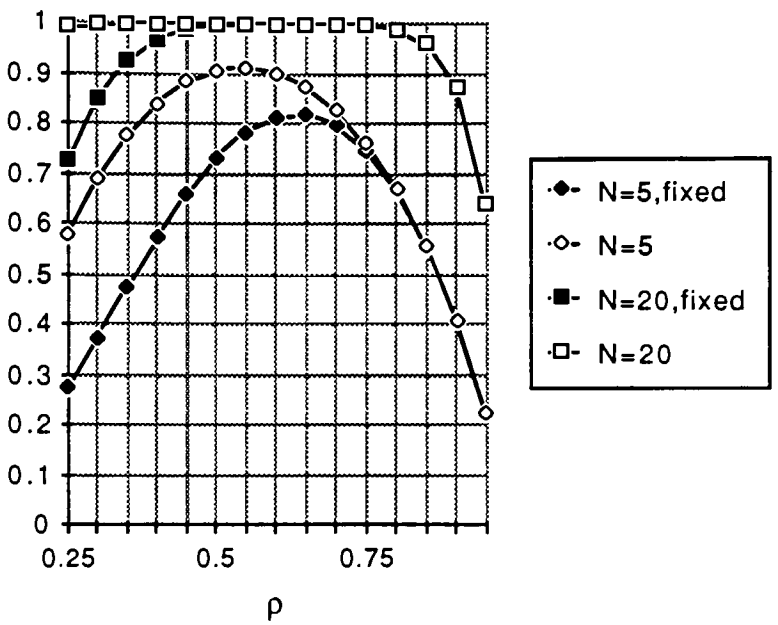


Figure A-1. Correction of Formula in [Livney 82].

Appendix B. "Adaptive Load Sharing in Homogeneous Distributed Systems"

Adaptive Load Sharing in Homogeneous Distributed Systems

DEREK L. EAGER, EDWARD D. LAZOWSKA, AND JOHN ZAHORJAN

Abstract—In most current locally distributed systems, the work generated at a node is processed there; little sharing of computational resources is provided. In such systems it is possible for some nodes to be heavily loaded while others are lightly loaded, resulting in poor overall system performance. The purpose of *load sharing* is to improve performance by redistributing the workload among the nodes.

The load sharing policies with the greatest potential benefit are *adaptive* in the sense that they react to changes in the system state. Adaptive policies can range from simple to complex in their acquisition and use of system state information. The potential advantage of a complex policy is the possibility that such a scheme can take full advantage of the processing power of the system. The potential disadvantages are the overhead cost, and the possibility that a highly tuned policy will behave in an unpredictable manner in the face of the inaccurate information with which it inevitably will be confronted.

The goal of this paper is not to propose a specific load sharing policy for implementation, but rather to address the more fundamental question of the appropriate level of complexity for load sharing policies. We show that extremely simple adaptive load sharing policies, which collect very small amounts of system state information and which use this information in very simple ways, yield dramatic performance improvements. These policies in fact yield performance close to that expected from more complex policies whose viability is questionable. We conclude that simple policies offer the greatest promise in practice, because of their combination of nearly optimal performance and inherent stability.

Index Terms—Design, load sharing, local area networks, performance, queueing models, threshold policies.

I. Introduction

LOAD SHARING attempts to improve the performance of a distributed system by using the processing power of the entire system to "smooth out" periods of high congestion at individual nodes. This is done by transferring some of the workload of a congested node to other nodes for processing. The potential attractiveness of load sharing is enhanced by factors such as the increasing size of locally distributed systems, the use of shared file servers, the presence of pools of computation servers, and the development of streamlined communication protocols.

Manuscript received October 31, 1984; revised June 28, 1985. This work was supported by the National Science Foundation under Grants MCS-8302383 and DCR-8352098, and by the Natural Sciences and Engineering Research Council of Canada. Part of this work was conducted while E. D. Lazowska was on leave at Digital Equipment Corporation's Systems Research Center.

D. L. Eager is with the Department of Computational Science, University of Saskatchewan, Sask. S7N 0W0, Canada.

E. D. Lazowska and J. Zahorjan are with the Department of Computer Science, University of Washington, Seattle, WA 98195.

IEEE Log Number 8607941.

Two important components of a load sharing policy are the *transfer* policy, which determines whether to process a task locally or remotely, and the *location* policy, which determines to which node a task selected for transfer should be sent. Policies that use only information about the average behavior of the system, ignoring the current state, are termed *static* policies. Static policies may be either *deterministic* or *probabilistic*. Policies that react to the system state are termed *adaptive* policies.

Numerous static load sharing policies have been proposed. In the earliest formulations of the problem it was assumed that information about the average execution times and intercommunication requirements of all tasks were known. Typically the goal was to find a technique to deterministically allocate tasks to nodes so that the total time to process all tasks was minimized; for example [2], [13], [14]. More recently, Tantawi and Towsley [15] developed a technique to find the optimal probabilistic assignment.

Adaptive load sharing policies have received less attention. Livny and Melman [11] showed that in a network of autonomous nodes there is a large probability that at least one node is idle while tasks are queued at some other node, over a wide range of network sizes and average node utilizations. This is a key result because it clearly indicates the potential benefit of adaptive load sharing. Livny and Melman also developed a taxonomy of load sharing policies, and used simulation to evaluate a number of them. Bryant and Finkel [3] proposed a specific adaptive load sharing policy, and analyzed its performance using simulation. They also explored techniques for estimating the remaining service time of a task already being processed, a quantity of interest in deciding which task to transfer from a congested node. Krueger and Finkel [7] also used simulation to evaluate the performance of a specific policy. Barak and Shiloh [1] used limited experimentation with synthetic workloads to investigate a policy distinguished by the technique used to maintain system state information. They showed that if the workload remained constant, their policy converged to a load distribution that was near optimal.

Static load sharing policies are attractive because of their simplicity: "transfer all compilations originating at node X to computation server Y" or "$\cdots$ to computation servers Y and Z with probabilities 0.8 and 0.2, respectively." It is clear, though, that the potential of static policies is limited by the fact that they do not react to the

curren. system state: at the time a particular compilation originates at node X, computation server Y may be so heavily loaded that Z is a much superior choice, or both Y and Z may be so congested that processing the task locally is preferable, even considering the impact of this decision on other tasks originating at node X. The attraction of adaptive policies is that they do respond to system state, and so are better able to avoid those states with unnecessarily poor performance. However, since adaptive policies must collect and react to system state information, they are necessarily more complex than static policies. The adaptive policies that have been examined in the literature collect considerable state information and attempt to make the "best" choice possible based on that information. For example, the policy proposed by Krueger and Finkel [7] attempts to keep the queue length at each node near the system average queue length.[1]

From a practical point of view, such complexity raises a number of concerns. The first concern is the effect of overhead. The value of a policy depends critically on the overhead required to administer it, which may vary considerably depending on system characteristics. Excessive overhead may negate the benefits of an improved workload distribution.

The second concern is the effect of the occasional poor decisions that inevitably will be made. Complex policies rely on detailed information about the system state and the behavior of the workload. Not only is this information expensive to gather, but some quantities, such as the expected congestion at nodes in the near future or the amount of processing that a particular task requires to complete, cannot be known precisely regardless of the effort expended. Because of this, a decision that a complex load sharing policy expects to be near optimal may in fact be quite poor.

The final concern is the potential for instability. In attempting to fully exploit system processing power, a complex load sharing policy must make decisions based on subtle apparent misallocations of load. This requirement to react to small distinctions means that the inherent inaccuracy and rapidly changing nature of system state information may cause the policy to react in an unstable manner [6]. At the extreme, a form of *processor thrashing* can occur, in which all of the nodes are spending all of their time transferring tasks. Less complex policies, because they tend to react more slowly to changes in the system state, are inherently less susceptible to such instability.

Motivated by these concerns, in this paper we ask a fundamental question concerning adaptive load sharing policies in general: what is an appropriate level of complexity for such policies? We show that:

- Extremely simple adaptive load sharing policies—policies that collect a very small amount of state information and that use this information in very simple ways—yield dramatic performance improvements relative to the no load sharing case.
- These extremely simple policies in fact yield performance close to that which can be expected from complex policies that collect large amounts of information and that attempt to make the "best" choice given this information—policies whose viability is questionable.
- These results are valid over a wide range of system parameters.

We conclude that simple adaptive load sharing is of considerable practical value, and that there is no firm evidence that the potential costs of collecting and using extensive state information are justified by the potential benefits.

II. Policies and Models

In studying the appropriate level of complexity for adaptive load sharing policies, we consider a set of abstract policies that represent only the essential aspects of load sharing, and we investigate these policies using simple analytic models. Our objective is not to determine the absolute performance of particular load sharing policies, but rather to assess the relative advantages of varying degrees of sophistication. By representing only the essential aspects of load sharing and eliminating secondary details, we are better able to interpret the results of our comparative analysis and so build our intuition.

An obvious concern is that this approach may ignore "details" with significant practical implications—the issues noted in Section I, such as the actual cost of collecting and reacting to state information, the behavior of policies when this information is unavailable or out-of-date, etc. If the conclusion of our study were that increasing sophistication yielded substantial benefit, then these concerns would have to be addressed, because failure to properly account for these characteristics will tend to overstate the performance of complex policies relative to the performance of simple ones. However, the conclusion of our study is quite the opposite, despite giving the "benefit of the doubt" to complex policies.

A. System Model

We represent distributed systems as collections of identical nodes, each consisting of a single processor. The nodes are connected by a local area broadcast channel (e.g., an Ethernet). All nodes are subjected to the same average arrival rate of tasks, which are of a single type.

In contrast to previous papers on load sharing, we represent the cost of task transfer as a processor cost rather than as a communication network cost. It is clear from measurement and analysis [9] that the processor costs of packaging data for transmission and unpackaging it upon reception far outweigh the communication network costs of transmitting the data. Further, network delays are small, and are almost entirely overlapped with processing

[1] An implicit assumption of most proposed schemes is that it is desirable to attempt to balance the queue lengths at the processors. In fact, such balancing is not required. All that is necessary for optimal performance (in the standard homogeneous model) is that all processors be busy if any task is waiting. Thus in this paper we purposefully adopt the terminology "load *sharing*" rather than "load *balancing*."

related to use of the network. Representing the network cost in addition to the processor cost would not affect the tractability of our models, but nor would it affect our results. Thus, for simplicity, it is omitted. (In Section III-E we will show that, under reasonable assumptions, the total communication network load imposed by adaptive load sharing is negligible.)

Our homogeneity assumptions—that nodes are identical and are subjected to the same average arrival rate of tasks—also are made principally to simplify the presentation, and do not undermine the applicability of the results. Node homogeneity is a reasonable assumption when considering load sharing among clusters of workstations or clusters of computation servers. Arrival homogeneity merely implies that *over the long term* the external load imposed on each node is the same. Over the short term, these loads may vary considerably. The entire objective of adaptive load sharing is to respond to such variations. Even if homogeneity does not hold (the system consists of a mix of nodes of different types, or there are differences in external loads), models that consider this case (but that are not considered here) indicate the suitability of simple policies. These simple policies are similar to those for homogeneous systems, but they additionally utilize the relatively static information specifying the system inhomogeneities.

B. Load Sharing Policies

We will study three abstract load sharing policies, comparing their performance to each other and to two "bounding" cases: no load sharing, and perfect load sharing at zero cost. As noted in Section I, a load sharing policy has two components: a *transfer* policy that determines whether to process a task locally or remotely, and a *location* policy that determines to which node a task selected for transfer should be sent. Each of these subpolicies might be expected to employ system state information. The three load sharing policies that we consider have identical transfer policies, but differ in their location policies.

The transfer policy that we have selected is a *threshold* policy: a distributed, adaptive policy in which each node uses only local state information. *No exchange of state information among the nodes is required in deciding whether to transfer a task.* A task originating at a node is accepted for processing there if and only if the number of tasks already in service or waiting for service (the node *queue length*) is less than some threshold T. Otherwise, an attempt is made to transfer that task to another node. Note that only newly received tasks are eligible for transfer. Transferring an executing task poses considerable difficulties in most systems [12].

The three location policies that we examine for use in conjunction with this extremely simple transfer policy are referred to as *Random*, *Threshold*, and *Shortest*. They are discussed in the subsections that follow.

1) Random: The simplest location policy is one that uses no information at all. With the Random policy a destination node is selected at random and the task is transferred to that node. *No exchange of state information among the nodes is required in deciding where to transfer a task.*

A question that arises in considering the behavior of the random policy is how the destination node should treat an arriving transferred task. The obvious answer is that it should treat it just as a task originating at the node: if the local queue length is below threshold the task is accepted for processing; otherwise it is transferred to some other node selected at random. As shown in Appendix A, this choice has the unfortunate property of causing instability: no matter what the average load, it is guaranteed that eventually the system will enter a state in which the nodes are devoting all of their time to transferring tasks and none of their time to processing them. This instability is analogous to that arising in the infinite population ALOHA system [5]; repeated task transfers in load sharing systems play a similar role with respect to stability as do message collisions in ALOHA.

Instability can be overcome by the use of an appropriate control policy. Such control policies have been developed for a number of multiple access systems [8], [16]. The simple control policy that we adopt here is to restrict the number of times that a task can be transferred using a static *transfer limit*, L_t. The destination node of the L_tth transfer of a task must process that task regardless of its state.

A key result of this paper is that, in many situations, this extremely simple combination of a threshold transfer policy and a random location policy with a static transfer limit dramatically improves system response time relative to no load sharing. Since this policy uses no system state information at all, this is an indication that very simple schemes can yield significant benefits.

2) Threshold: Threshold is a location policy that acquires and uses a small amount of information about potential destination nodes. Under this policy a node is selected at random and *probed* to determine whether the transfer of a task to that node would place it above threshold. If not, then the task is transferred; the destination node must process the task regardless of its state when the task actually arrives. If so, then another node is selected at random and probed in the same manner. This continues until either a suitable destination node is found, or the number of probes exceeds a static *probe limit*, L_p. In the latter case, the originating node must process the task.

The objective of the Threshold policy is to avoid "useless" task transfers (those to nodes already at or above their threshold), although, like Random, it makes no attempt to choose the "best" destination node for a task. The use of probing with a fixed limit, rather than broadcast, ensures that the cost of executing the load sharing policy will not be prohibitive even in large networks. As will be discussed in Section III-D, the performance of this policy is surprisingly insensitive to the choice of probe limit. In other words, the performance with a small (and

economical) probe limit, e.g., 3 or 5, is almost as good as the performance with a large probe limit, e.g., 20.

A key result of this paper is that the Threshold policy provides substantial performance improvement relative to the Random policy for a wide range of system parameters. This indicates that the use of a small amount of state information in a simple (and computationally inexpensive) way is likely to more than compensate for the additional cost.

3) Shortest: This location policy acquires additional system state information and attempts to make the "best" choice given this information. L_p distinct nodes are chosen at random, and each is polled in turn to determine its queue length. The task is transferred to a node with the shortest queue length, unless that queue length is greater than or equal to the threshold, in which case the originating node must process the task. The destination node must process the task regardless of its state at the time the task actually arrives. (A simple improvement to Shortest is to discontinue probing whenever a node with queue length of zero is encountered, since that node is guaranteed to be an acceptable destination.)

The Shortest policy uses more state information, in a more complex manner, than does the Threshold policy. A key result of this paper is that the performance of Shortest is not significantly better than that of the simpler Threshold policy. This suggests that state information beyond that used by Threshold, or a more complex usage of state information, is of little benefit.

C. Analytic Model Structure and Solution

The three policies introduced in the previous section have similar analytic models.

Each node is modeled as a queueing center [4]. New tasks arrive at each node at average rate λ. The average task service time (processing cost) is S. We define the *load factor* ρ of each node to be the ratio of offered load to service capacity (i.e., $\rho = \lambda S$). Because of the cost of task transfer, the average utilization of the nodes may be significantly greater than ρ.

The cost of transferring a task from one node to another is represented by a processing cost at the sending node whose average value is denoted by C. This cost is a key parameter. (The processing cost of receiving a task is included in the service time of the task, S.) As discussed earlier, communication network costs are assumed to be negligible (relative to other costs). In addition, the cost of probing a node is assumed to be negligible. These assumptions are examined in Section III-E.

At each node, the transferring of tasks is given preemptive priority over the processing of tasks. In the processing of tasks, any service discipline that selects tasks in a way that is independent of their actual service time (e.g., First-Come-First-Served, Processor Sharing) is allowed. All of the performance measures that will be considered here are independent of the actual discipline used.

Under the assumptions stated above, a Markov model of a distributed system under each of the load sharing policies can be constructed. The model has a very large state space, with complex structure. To simplify the analysis we decompose the model, by assuming that the state of each node is stochastically independent of the state of any other node. Each node can then be analyzed in isolation. The effect of the remainder of the system on an individual node is represented by an arrival process of transferred tasks. Because the network is homogeneous, system performance measures can be obtained by analyzing a model of any individual node.

This decomposition approach is asymptotically exact as the number of nodes in the system increases, since the queue lengths of the nodes are asymptotically independent. For systems of finite size the analysis is an approximation. Results obtained from simulation indicate that this approximation, which also has been used in modeling multiple access protocols such as ALOHA, introduces negligible errors even for relatively small numbers of nodes. In particular, the major numerical results used in our study have been validated through simulation for networks of 20 nodes (and thus certainly for greater numbers of nodes, although not necessarily for smaller numbers). A sample of our simulation results is contained in Appendix C.

All of the quantities needed to determine the state transition rates of the model of an individual node are input parameters, with the exception of a description of the arrival process of transferred tasks. The nature of this arrival process depends on the load sharing policy. For the Random policy, the arrival rate of transferred tasks is independent of the current queue length (i.e., state) of the node, since Random utilizes no information about the state of potential destination nodes. For the Threshold policy, arrivals of transferred tasks are constrained to those states in which the node is below its threshold. For the Shortest policy, the arrival rate of transferred tasks decreases as the queue length at the node increases.

The assumption of homogeneous nodes makes it possible to determine the arrival rate of transferred tasks: the overall arrival rate must equal the overall rate at which the node transfers tasks to other nodes, and the equilibrium state probabilities of potential destination nodes, as "observed" when probing, for example, are identical to those of the node itself. These quantities are model outputs. For the Random and Threshold policies this dependence of model inputs on outputs yields a single equation in a single unknown, which is solved numerically. For the Shortest policy, the dependence is sufficiently complex that an iterative numerical technique is required.

Equations relating the variables of the model are developed by considering the node to be in one of two phases: "processing" (when the node queue length is less than or equal to the threshold value), or "transferring" (when the node queue length is greater than the threshold value). During a processing phase the node is either idle or is processing tasks. During a transferring phase the node is busy, either transferring tasks or processing tasks

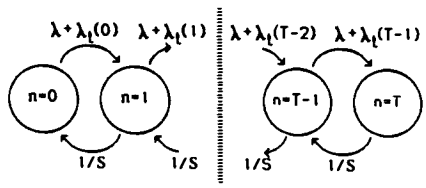


Fig. 1. Processing phase birth-death model.

that could not be transferred because of a restriction imposed by the location policy.

Fig. 1 shows the birth-death model corresponding to the processing phase. In each state the arrival rate of tasks is the sum of the rate of arrival of new tasks (λ) and the rate of arrival of tasks transferred to this node by the remainder of the system ($\lambda_t(n)$). This latter term is in general dependent on the queue length n at the node and the load sharing policy being modelled. This submodel can be analyzed using standard methods.

A transferring phase is identical in behavior to a busy period of a two class, preemptive priority HOL M/M/1 queue [4], where the classes are tasks that are processed and tasks that are transferred. The total arrival rate at the node is $(\lambda + \lambda_t(T^+))$, where $\lambda_t(T^+)$ denotes the arrival rate of tasks transferred to the node conditioned on the node being in a transferring phase. The proportion of this total arrival rate consisting of tasks that will be processed and the proportion consisting of tasks that will be transferred depends on the probability of a task not being transferred because of a location policy restriction.

The analyses of the birth-death model corresponding to the processing phase and of the HOL priority model corresponding to the transferring phase yield conditional state probabilities and performance measures. These are combined using weights representing the proportion of time the system spends in each phase to determine overall performance. The performance measures that can be obtained include average response times, utilizations, queue lengths, transfer rates, and probe rates. Details on the analyses of the two phases and the calculation of performance measures are given in Appendix B.

III. Performance Comparisons

Our objective is to compare the performance of three abstract load sharing policies—Random, Threshold, and Shortest—to each other and to two "bounding" cases: no load sharing (represented by K independent M/M/1 queues, where K is the number of nodes), and perfect load sharing at zero cost (represented by an M/M/K queue). Our measure of performance is mean response time as a function of system load.

This comparison is potentially difficult because of the large number of parameters involved: the average task service time S, the average cost of task transfer C, the threshold T, the probe limit for the Threshold and Shortest policies L_p, the transfer limit for the Random policy L_t, and the number of nodes K. Fortunately, the results are robust in the sense that the intuition gained from studying

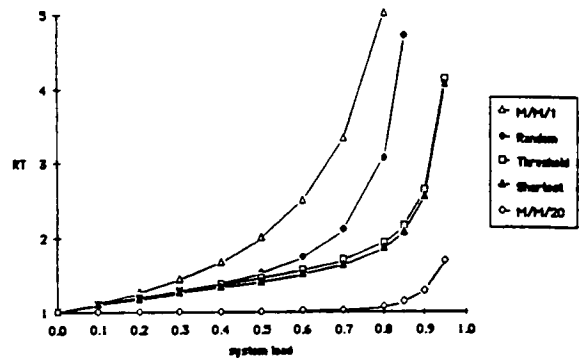


Fig. 2. Principal performance comparison: response time versus load ρ. S(task service time) = 1. C(cost of task transfer) = 0.1. T(threshold) = 2. L_p(probe limit for Threshold and Shortest) = 3. L_t(transfer limit for Random) = 1.

performance for a "representative" set of parameter values is valid over a wide range of parameter values. The structure of our presentation exploits this fact: Section III-A contains a thorough discussion of response time versus system load for a particular choice of parameter values, while Sections III-B–III-F explore the sensitivity of these results to the various parameters.

A. Principal Performance Comparison

Fig. 2 is a graph of average response time versus load for each of the five policies under consideration. For convenience, S is fixed at 1 throughout our analysis so that response times may be considered to be reported in units of the task service time.

We will first discuss the figure, and then the choice of parameter values indicated in the text accompanying the figure. The key observations concerning the figure are as follows.

- The Random policy yields substantial performance improvement over no load sharing. The degree of the improvement is surprising since the Random policy is so simple.
- The Threshold policy yields substantial further performance improvement for system loads greater than 0.5. This shows the value of the small amount of additional information utilized by Threshold.
- The Shortest policy yields negligible further performance improvement over the Threshold policy. Again this is somewhat surprising, since Shortest acquires considerably more information than Threshold, and attempts to make the "best" decision based on that information.

If factors such as the actual cost of collecting and reacting to state information, the behavior of policies when this information is unavailable or out-of-date, etc., are ignored, then intuitively Shortest should have the best performance among all load sharing policies that employ threshold transfer policies. Thus, based on the comparison of Threshold and Shortest, we can conclude (subject to verification that our results are robust with respect to the choice of parameter values) that relatively simple information concerning potential destination nodes is suffi-

cient to obtain essentially all of the benefit available through this class of policies. This conclusion is reinforced by the fact that our analysis indeed gives the "benefit of the doubt" to complex policies by ignoring the issues just noted, which clearly are more significant for complex policies such as Shortest than for simpler policies such as Threshold.

In Fig. 2 there is significant room for improvement between the performance of the Shortest policy and the bound established by the M/M/K analysis. This might suggest that our conclusion with respect to the information required by location policies does not hold for transfer policies: perhaps significantly improved performance can be obtained by using a transfer policy that employs more than local threshold information. However, there are reasons (in addition to the obvious pragmatic ones) to believe that simple transfer policies are as relatively advantageous as simple location policies. The M/M/K analysis does not provide a tight bound: it assumes perfect load sharing at zero cost, when in fact an "optimal" policy would require a significant rate of task transfers, each of which has a nonnegligible cost. Further, the parameter values used in Fig. 2 are conservative, rather than being advantageous to the policies under consideration. We will discuss these parameter values now, and return to the question of an appropriate optimistic bound on achievable performance in Section III-F.

In Fig. 2, the average cost of task transfer C was 0.1, that is, 10 percent of the average task service time. We believe this to be a conservative (overly high) choice; our reasoning, as well as the sensitivity of the results to the cost of task transfer, is explored in Section III-B.

The threshold T was 2. That is, a node would attempt to transfer a task that arrived when two (or more) tasks already were present. The sensitivity of the results to the choice of threshold is explored in Section III-C.

The probe limit for the Threshold and Shortest policies L_p was 3. The sensitivity of the results to the choice of probe limit is explored in Section III-D. The rates of probing in the Threshold and Shortest policies are compared in Section III-E.

The transfer limit for the Random policy L_t is set to 1. The implications of this will be discussed in Section III-B. The rate of task transfers for all policies, and its impact on network congestion, is discussed in Section III-E.

As noted in Section II, the number of nodes K is not a parameter of our analysis of Random, Threshold, and Shortest. The analysis is asymptotically exact as the number of nodes increases. Our major results have been validated through simulation for networks of 20 nodes, implying that the performance of the policies quickly becomes insensitive to the number of nodes as the number of nodes increases.

B. Sensitivity to Transfer Cost

We believe that the average cost of task transfer C, although nonnegligible, can be expected to be quite low relative to the average cost of task processing S; the range 1–10 percent seems to include the cases of greatest interest. (We mean this to be interpreted as an average across many tasks; we are not asserting a relationship between processing cost and transfer cost.)

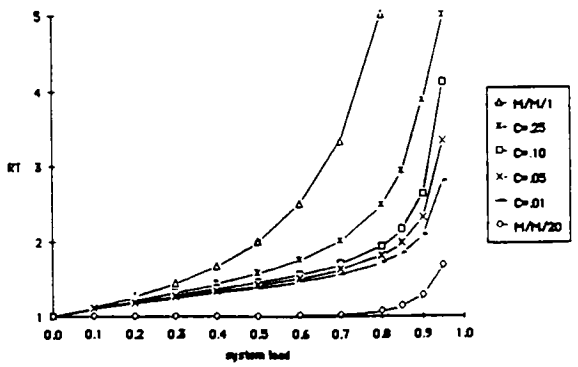


Fig. 3. Response time versus load ρ for various transfer costs C(Threshold policy). S(task service time) = 1. T(threshold) = 2. L_p(probe limit) = 3.

Transfer costs higher than 10 percent, although certainly possible, would likely be infrequent. On current systems not designed to facilitate load sharing (e.g., 4.2 BSD Unix running on Vaxes connected by Ethernet, using FTP on top of TCP/IP for task transfer), the transfer costs for relatively small compilations and formatting runs are a few percent of the processing costs. We would expect that any practical implementation of load sharing would attempt to select tasks such as these for migration—tasks with a relatively high ratio of processing cost to transfer cost. One also can easily imagine more efficient protocols. The advent of systems based on file servers and database servers will further decrease the cost of task transfer: only a descriptor will be shipped.

At the other end of the spectrum, performance is insensitive to transfer cost for costs of 1 percent or less.

Fig. 3 shows average response time versus system load for the Threshold policy for four different average transfer costs C: 0.01 (1 percent of the processing cost), 0.05 (5 percent), 0.10 (10 percent as shown in Fig. 2) and 0.25 (25 percent). The other parameters (e.g., threshold, probe limit) are fixed as in Fig. 2. Note that in practice the average transfer cost would be a factor considered in selecting the value of the threshold, whereas a fixed threshold of 2 was used for each transfer cost in Fig. 3.

Fig. 4 shows average response time versus average transfer cost C for all policies, for a fixed system load of 0.7. (Note that a log scale is used for the transfer cost axis.) Again, the other parameters (e.g., threshold, probe limit) are fixed as in Fig. 2. The performance of Threshold and Shortest relative to one another is insensitive to transfer cost. Their performance relative to the extremes of the M/M/1 and M/M/K analyses is insensitive to transfer cost for values below 0.05 (5 percent of processing cost), but degrades rapidly as transfer costs exceed 0.25 (25 percent). The Random policy performs relatively better at low transfer costs than at high ones. In fact, our

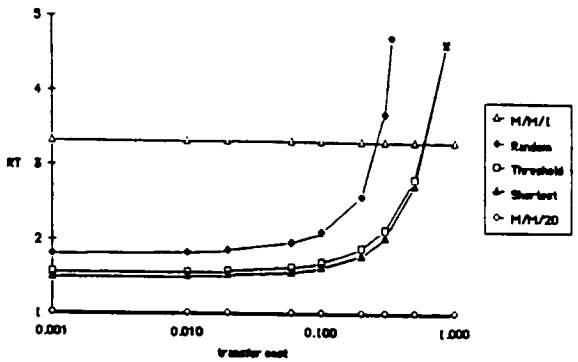


Fig. 4. Response time versus transfer cost C at fixed load ρ. S(task service time) = 1. ρ(system load, λS) = 0.7. T(threshold) = 2. L_p(probe limit for Threshold and Shortest) = 3. L_t(transfer limit for Random) = 1.

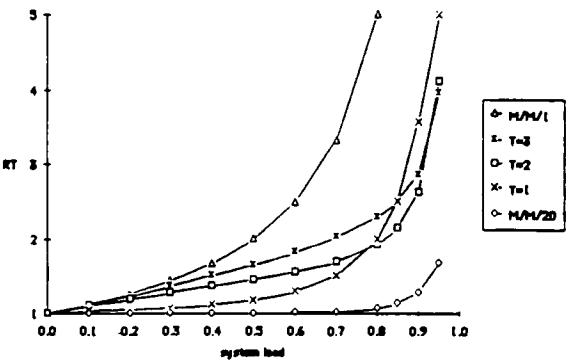


Fig. 5. Response time versus load ρ for three thresholds T (Threshold policy). S(task service time) = 1. C(cost of task transfer) = 0.1. L_p(probe limit) = 3.

analysis does not do justice to Random at low transfer costs. Reasonable performance at relatively high transfer costs requires a transmission limit L_t of 1, the value used throughout our analysis. However, at relatively low transfer costs a higher transmission limit yields substantially better performance, since tasks can be transferred multiple times (at low cost) in search of a suitable node. This yields behavior similar to that of the Threshold policy, except that the task itself is sent, rather than a probe.

C. Choice of Threshold

The threshold T is a fundamental parameter: for each of the three load sharing policies it determines when a task transfer will be attempted (through the transfer policy); for the Threshold and Shortest policies it determines whether the transfer will be allowed (through the location policy).

Clearly the "best" threshold depends on the system load and the transfer cost. At low loads a low threshold is appropriate because many nodes are idle, whereas at high loads a high threshold is appropriate because most nodes have significant queue lengths. Low thresholds are appropriate for low transfer costs, since smaller differences in node queue lengths can be exploited; high costs demand higher thresholds.

One might imagine that a complex adaptive threshold selection strategy would be required to obtain reasonable performance. Figs. 2–4, which used a fixed threshold of 2, indicate that this is not the case. To explore this point further, Fig. 5 shows average response time versus system load for the Threshold policy for three thresholds: 1, 2 (as shown in Fig. 2) and 3. (The corresponding graph for Shortest is essentially indistinguishable.) The other parameters are fixed as in Fig. 2. We see that 1 is the optimal threshold for system loads below 0.8. 2 is the optimal threshold for loads between 0.8 and 0.9, and thresholds greater than 2 are advantageous at (unreasonably high) system loads above 0.95. (The Random policy exhibits greater sensitivity to choice of threshold, but the optimal threshold still is 1 over a wide range of system load.)

These results suggest that the optimal threshold is not very sensitive to system load. Thus, a simple adaptive policy that selects among two or three threshold values, perhaps based on information acquired while probing, offers potential benefit at low cost and risk. Such policies are an area of current research.

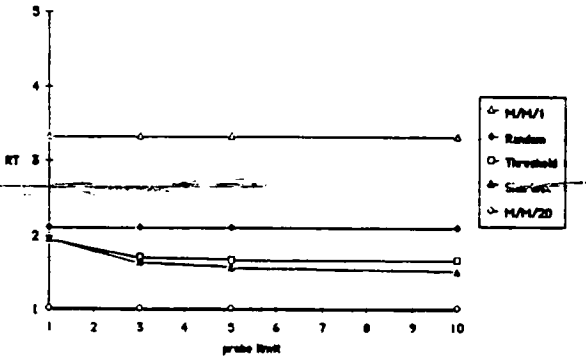


Fig. 6. Response time versus probe limit L_p at fixed load ρ. S(task service time) = 1. C(cost of task transfer) = 0.1. ρ(system load, λS) = 0.7. T(threshold) = 2. L_t(transfer limit for Random) = 1.

D. Choice of Probe Limit

Fig. 6 shows average response time versus probe limit for all policies, for a fixed system load of 0.7. (Random has no probe limit; it is included along with M/M/1 and M/M/K for comparison purposes.)

In the case of Threshold, the rapid decrease in the marginal benefit of increasing the probe limit is easy to explain. The purpose of probing in this policy is to locate a node that is below threshold. If p is the probability that a particular node is below threshold, then (because the nodes are assumed to be independent) the probability that a node below threshold is first encountered on the ith probe is $p(1-p)^{i-1}$. For large p, this quantity decreases rapidly: the probability of succeeding on the first few probes is high. For small p, the quantity decreases more slowly. However, since most nodes are busy, the improvement in system-wide response time that will result from locating a node below threshold is small, so abandoning the search after the first few probes does not carry a substantial penalty. It is clear that small probe limits are appropriate.

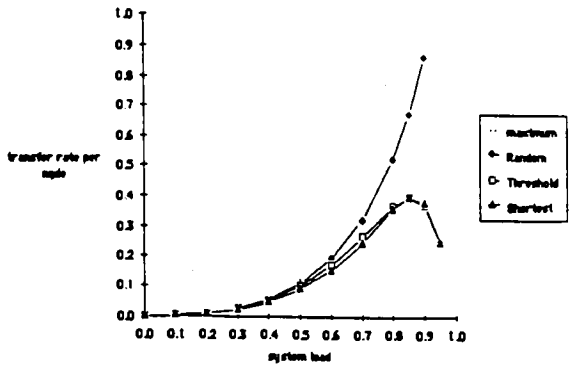


Fig. 7. Task transfer rate per node versus load ρ. S(task service time) = 1. C(cost of task transfer) = 0.1. T(threshold) = 2. L_p(probe limit for Threshold and Shortest) = 3. L_r(transfer limit for Random) = 1.

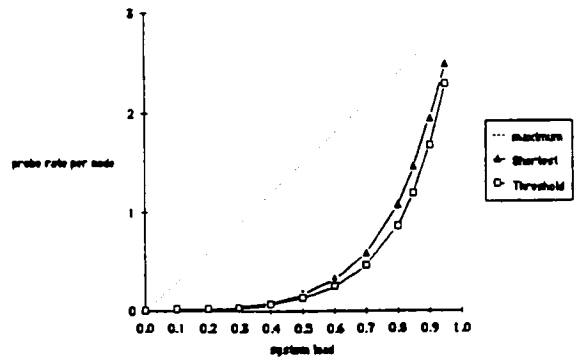


Fig. 8. Rate of probes per node versus load ρ. S(task service time) = 1. C(cost of task transfer) = 0.1. T(threshold) = 2. L_p(probe limit) = 3.

In the case of Shortest, the situation is somewhat more complex. There may be some marginal benefit even to very large probe limits. Fig. 6 shows this, but understates the effect for two reasons. First, this benefit is greatest at high system loads; the load of 0.7, selected for consistency with other figures and a "reasonable" high load for illustrative purposes, is not high enough to fully display the effect. Second, at a threshold value of 2, Shortest cannot find a node with a queue length more than one task shorter than that of the first acceptable destination found, since the maximum acceptable destination queue length is 1. If the threshold were higher (because of a higher system load, for example), there would be more room for improvement. However, it still is the case that relatively small probe limits are appropriate for Shortest. The marginal benefit of increasing the probe limit does decrease (although this decrease is not as rapid as for Threshold), and, as the probe limit increases, the rate and hence the cost of probing increases (this increase is actually greater for Shortest than for Threshold). (The latter effect is not shown in the figures, since the cost of probing is omitted from our analysis.)

E. Transfer and Probing Traffic

Here we consider the network traffic due to task transfers (for all three policies) and probes (for Threshold and Shortest).

Threshold and Shortest each will transfer an individual task at most one time. This also is true of Random with the transfer limit of 1 that we have been using in our examples. This implies that the transfer rate *per node* can be no greater than λ, and that the task transfer rate over the entire system can be no greater than $K\lambda$.

As illustrated in Fig. 7, the actual task transfer rates for Threshold and Shortest are extremely similar and are considerably less than this maximum value, while the rate for Random approaches this maximum only for relatively high system loads. (The unit of time in the figure is the task processing time S, which is equal to 1.) It is impossible to translate these results into network utilization without making rather arbitrary assumptions, but for the sake of illustration, suppose that the processing cost of tasks is related to their size in the ratio of 1 second per 1 kbytes (e.g., a 100K task would process for 100 seconds), and that we are considering a 10 Mbit network (i.e., each 1K transferred requires 0.0008 seconds of network time). Then, an upper bound network utilization due to task transfers under the Threshold or Shortest policies would be $0.4 \times 0.0008 \times K$ (since these policies never exceed a transfer rate of 0.4), yielding a network utilization of 3 percent in a system of 100 nodes. If we envision a system based on file servers, then the network cost of load sharing over and above the inherent cost of remote file access is insignificant regardless of the particular assumptions that are made.

Note, incidentally, that the decrease in transfer rate at high loads for Threshold and Shortest is exactly what one should expect: in this situation many nodes are over threshold, so there is an increasing probability that a transfer attempt will fail, i.e., that no suitable destination node will be found during the probe phase.

Fig. 8 shows the rate of probes per node for Threshold and Shortest. Because the probe limit is 3, the maximum probe rate per node can be no greater than 3λ. The figure shows that the two policies behave similarly, with Threshold requiring marginally fewer probes than Shortest. The difference is maximized at approximately the point corresponding to the maximum transfer rate. (The difference between the two policies can be much larger for larger probe limits and/or thresholds.) As the system load increases beyond 0.7, the probe rate for each policy begins to increase substantially. Still, given the maximum rate per node of $L_p\lambda$, the network load and processor load due to probing will be negligible. (Note that at most L_p probes can be performed per processed task.) Probing could be implemented, for example, using a single remote procedure cell with a return value that is binary (in the case of Threshold) or integer (in the case of Shortest).

F. An Optimistic Evaluation

We noted in Section III-A that our evaluation was conservative in two respects: the M/M/K analysis does not provide a tight bound, and the choice of parameter values is not advantageous to the policies under consideration.

In this section we briefly consider the Threshold scheme from a more optimistic point of view:

- The threshold T, rather than being fixed, is set to either 1 or 2 depending on the system load. Fig. 5 suggests the improvement that this offers.
- The probe limit is increased from 3 to 5. Fig. 6 suggests the improvement that this offers.
- We consider an average transfer cost of 0.01, in addition to 0.10. (The average task processing time remains fixed at 1.) Fig. 3 suggests the improvement that this offers.
- We compare performance to a plausible lower bound that includes the average transfer cost, obtained as follows:

An M/M/K queueing system can be viewed as a model of perfect load sharing among K nodes: no node will ever be idle when more than a single task is present at some other node. The results of an M/M/K analysis are "too optimistic," though, because the cost of the task transfers required to achieve this perfect load sharing is ignored.

Livny and Melman [11] calculate a lower bound on the number of task transfers required to ensure that no node will ever be idle when another node has a queue length greater than 1. They note that a task must be transferred when one of the following events occurs: an arrival occurs at a busy node when there are less than K tasks in the system, or a completion occurs at a node with only one task present when there are more than K tasks in the system. Thus, the minimum rate of task transfers $\bar{\lambda}_T$ can be expressed as

$$\bar{\lambda}_T = \sum_{i=1}^{K-1} \left\{ \lambda i P[i] + \frac{1}{S}(K - i)\, P[K + i] \right\}$$

where $P[j]$ is the probability that there are j tasks in an M/M/K queueing system with arrival rate $K\lambda$ and service rate per server $1/S$.

This expression can be used to increase the average task service time S by the transfer cost C multiplied by the probability that a task requires a transfer, $\bar{\lambda}_T/\lambda$. Since the use of these increased service times in the M/M/K analysis results in a new set of state probabilities (implying a new rate of task transfers), an iteration is used.

Fig. 9 shows a comparison of M/M/1, Threshold with transfer costs of 0.1 and 0.01, the modified M/M/K analysis just described (labeled "Mod. M/M/20") with a transfer cost of 0.1, and the traditional M/M/20 analysis. The important observations are:

- With a variable threshold and a probe limit of 5, the performance of Threshold with a transfer cost of 0.1 is noticeably improved over that shown in Fig. 2, i.e., noticeably further from the performance of the M/M/1 system, and noticeably closer to the performance of the M/M/20 system.
- Viewing the modified M/M/K analysis as a plausible lower bound, there is little room for improvement beyond the performance of Threshold.

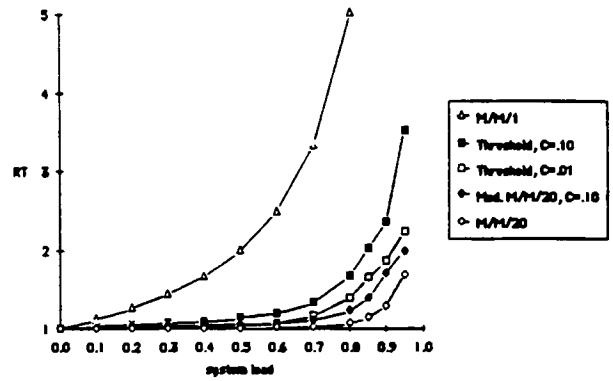


Fig. 9. Optimistic evaluation of the Threshold policy. S(task service time) = 1. T(threshold) = *variable*. L_p(probe limit) = 5.

- When the transfer cost drops to 0.01, the performance of Threshold is such that there is very little room for improvement relative to the absolute bound established by the M/M/20 analysis. (The performance of the modified M/M/20 system with a transfer cost of 0.01 is indistinguishable from the performance of the traditional M/M/20 system.)

IV. Summary

We have explored the use of system state information in adaptive load sharing policies for locally distributed systems, with the goal of determining an appropriate level of policy complexity. Our investigations have been based on the use of simple analytic models of load sharing policies. Simulation results have indicated the validity of these models.

Our results suggest that extremely simple load sharing policies using small amounts of information perform quite well—dramatically better than when no load sharing is performed, and nearly as well as more complex policies that utilize more information. This provides convincing evidence that the potential benefits of adaptive load sharing can in fact be realized in practice.

Our original intent in considering the class of threshold policies in general, and the Threshold policy in particular, was to establish a plausible bound on the performance of realistic load sharing schemes by considering a policy so simple that one expects to be able to do better in practice. However, the results of our analysis indicate that fairly direct derivatives of Threshold are plausible candidates for implementation. In particular, our results have shown the benefit of "threshold-type" information, as opposed to no information at one extreme or to "complete" information at the other.

Appendix A
Instability of Random with No Transmission Limit

This Appendix considers a variation of the Random policy in which there is no transfer limit ($L_t = \infty$): transferred tasks are treated exactly as new tasks when applying the transfer policy. We refer to this policy as *Uncontrolled Random*. Uncontrolled Random is *unstable* for a

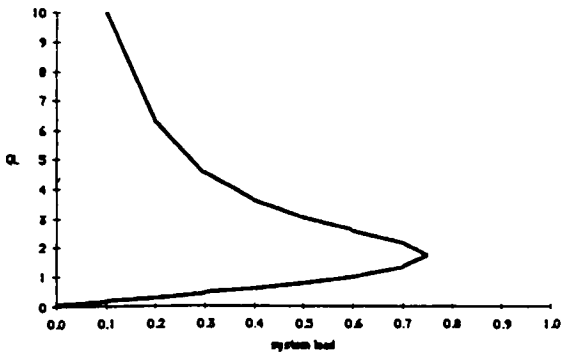


Fig. 10. Uncontrolled Random: queue length equilibrium contour. S(task service time) = 1. C(cost of task transfer) = 0.1. T(threshold) = 2.

nonzero transfer cost. No matter what the average load ρ, the system does not approach an equilibrium behavior: the expected backlog of work increases monotonically with time. Intuitively, this behavior occurs since there is a positive probability that all nodes will be in a transferring phase simultaneously. In Uncontrolled Random, each node would then begin to receive transferred tasks at rate $1/C$, and would try to retransfer these tasks at the same rate. Since no useful work is being done, the queue size at each node would increase at rate λ.

The instability of Uncontrolled Random is analogous to that of the infinite population ALOHA system [5]. Much of the terminology regarding stability that has been developed for use in the ALOHA context will be used here.

Fig. 10 is a result of the analysis that follows in this Appendix. The figure shows the mean node queue length *equilibrium contour* as a function of the system load, for Uncontrolled Random with a threshold of 2 and a transfer cost of 0.1. The average task service time is fixed at one. The equilibrium contour is composed of *system equilibrium points*, at which the output traffic intensity of the system (defined as the throughput multiplied by the average task service time) equals the input traffic intensity (or loading factor). A *load line* is defined as a vertical line corresponding to a particular value of input traffic intensity. There may be none, one, or two points of intersection of an equilibrium contour with a particular load line. At the input traffic intensity ρ_{max} equal to the maximum possible output traffic intensity, there is only one intersection point. For values of ρ greater than ρ_{max}, the system is overloaded and queue lengths grow without bound; in this case there are no intersection points. For values of ρ less then ρ_{max}, there are two intersection points; the lower is termed the *system operating point* and the upper the *system saturation point*. Below the system saturation point, the tendency of the system is to return to the system operating point. However, once the system saturation point is exceeded (which occurs eventually due to random fluctuations), the performance of the system degrades rapidly.

The remainder of this Appendix consists of the analysis of the Uncontrolled Random policy for system equilibrium points. Here $p_n (0 \le n \le T)$ and $\bar{n}_p$ denote the conditional probability of queue length n, and the mean queue length, respectively, given that the node is in a processing phase. λ_t and $\lambda_t(n)$ denote unconditioned and conditioned arrival rates of transferred tasks, respectively.

In Uncontrolled Random, the arrival rate of transferred tasks is independent of the node state. Therefore, for all n:

$$\lambda_t(n) = \lambda_t. \tag{UR1}$$

The arrival rate of transferred tasks must equal the rate of task transfers. Also, since no tasks are processed in the transferring phase, the probability of being in this phase is equal to that portion of the node utilization that is due to transferring tasks, or $C\lambda_t$. Since only those tasks that arrive while the queue length is greater than or equal to T are transferred,

$$\lambda_t = [p_T(1 - C\lambda_t) + C\lambda_t](\lambda + \lambda_t).$$

Solving for p_T yields

$$p_T = \frac{\dfrac{\lambda_t}{\lambda + \lambda_t} - C\lambda_t}{1 - C\lambda_t} \tag{UR2}$$

Consider now the birth–death model of Fig. 1. The conditional probability p_0 is given by

$$p_0 = \frac{1 - \rho - C\lambda_t}{1 - C\lambda_t}.$$

Using this expression, (UR1), and the formula for the solution of a birth–death model [4] yields, for $0 \le n \le T$,

$$p_n = \frac{1 - \rho - C\lambda_t}{1 - C\lambda_t}[S(\lambda + \lambda_t)]^n. \tag{UR3}$$

Equating the right-hand side of (UR3) for $n = T$ with the right-hand side of (UR2) gives

$$\frac{\lambda_t}{\lambda + \lambda_t} - C\lambda_t = (1 - \rho - C\lambda_t)[S(\lambda + \lambda_t)]^T. \tag{UR4}$$

Equation (UR4) has the solution $\lambda_t = (1/S) - \lambda$ for all $C \ge 0$. However, for each S in the region of interest ($C < S < (1/\lambda)$), this solution does not result in conditional state probabilities that sum to one, except for a special case value of λ that depends on S. This special value can be found by substituting this solution for λ_t into equation (UR3). Noting that the right-hand side is then independent of n, and that all of the conditional probabilities must therefore be equal, the conditional probabilities sum to one if and only if $p_n = 1/(T + 1)$. Making this substitution and solving for λ yields as the special case value:

$$\lambda = \frac{1 - \dfrac{C}{S}}{\dfrac{T + 1}{T} S - C}.$$

If $C = 0$ (and $\rho < 1$), (UR4) has exactly one valid so-

lution. Therefore, there is only one equilibrium point, and the system is stable. If $C > 0$, (UR4) has exactly two valid solutions for all positive values of ρ less than some value $\rho_{max} < 1$, one valid solution for $\rho = \rho_{max}$, and no valid solutions for $\rho > \rho_{max}$. The value ρ_{max} defines the maximum possible throughput of the system. For $\rho \geq \rho_{max}$ the system is overloaded. For $0 < \rho < \rho_{max}$ the system is unstable.

The solutions of equation (UR4) may be found numerically by any appropriate method. (The method used to obtain the numerical results of Fig. 10 is based on the bisection method of finding roots.) For each valid solution, the conditional state probabilities p_n are given by (UR3). The conditional mean queue length $\bar{n}_p$ is given by

$$\bar{n}_p = S(\lambda + \lambda_t)\left[\frac{1 - \rho - C\lambda_t}{1 - C\lambda_t}\right]\left[\frac{1 - [S(\lambda + \lambda_t)]^T}{[1 - S(\lambda + \lambda_t)]^2} - \frac{T[S(\lambda + \lambda_t)]^T}{1 - S(\lambda + \lambda_t)}\right]. \quad \text{(UR5)}$$

From this solution all of the performance measures of interest can be derived. For example, the mean response time R of tasks is given by

$$R = \frac{\bar{n}_p(1 - C\lambda_t) + TC\lambda_t}{\lambda} + \frac{\lambda_t}{\lambda}\left[\frac{C}{1 - C(\lambda + \lambda_t)}\right]. \quad \text{(UR6)}$$

The first term in (UR6) is derived by applying Little's equation [10], using expressions for the throughput and mean queue length of locally processed tasks. The throughput of processed tasks is given by λ. The mean queue length of tasks to be processed at the node is given by the mean queue length during a processing phase, multiplied by the probability of being in a processing phase, plus the queue length during a transferring phase, multiplied by the probability of being in a transferring phase. The second term in (UR6) is just the mean number of times a task must be transferred, multiplied by the mean delay experienced each time.

Appendix B
Solution of Models

This Appendix completes the analysis of the load sharing models introduced in Section II. As in Appendix A, $p_n(0 \leq n \leq T)$ and $\bar{n}_p$ denote the conditional probability of queue length n and the mean queue length, respectively, given that the node is in a processing phase; λ_t and $\lambda_t(n)$ denote unconditioned and conditioned arrival rates of transferred tasks, respectively. In addition, p^a_{T+} denotes the absolute probability of being in a transferring phase.

A. Random Policy

In the Random policy, the arrival rate of transferred tasks is independent of the node state. Therefore, for all n,

$$\lambda_t(n) = \lambda_t. \quad \text{(R1)}$$

The arrival rate of transferred taks must equal the rate of task transfers. Since only those tasks that arrive while the queue length is greater than or equal to T are transferred, under the constraint of the transfer limit L_t,

$$\lambda_t = \lambda\left[\sum_{l=1}^{L} [p_T(1 - p^a_{T+}) + p^a_{T+}]^l\right].$$

Note, in the above equation, that λ multiplied by the lth term in the summation gives the rate at which tasks that have already been transferred $l - 1$ times are transferred once more. Performing the summation yields

$$\lambda_t = [p_T(1 - p^a_{T+}) + p^a_{T+}]\,\lambda\left[\frac{1 - [p_T(1 - p^a_{T+}) + p^a_{T+}]^L}{1 - [p_T(1 - p^a_{T+}) + p^a_{T+}]}\right]. \quad \text{(R2)}$$

The probability of being in a transferring phase is just that portion of the node utilization due to performing task transfers and processing tasks that could not be transferred because of the transfer limit. Therefore,

$$p^a_{T+} = C\lambda_t + S\lambda[p_T(1 - p^a_{T+}) + p^a_{T+}]^{L+1}.$$

Using (R2) to substitute for the exponentiated term and solving for p^a_{T+} yields

$$p^a_{T+} = \frac{p_T S(\lambda + \lambda_t) - (S - C)\lambda_t}{1 - (1 - p_T)S(\lambda + \lambda_t)}. \quad \text{(R3)}$$

Consider now the birth-death model of Fig. 1. Using the formula for the solution of a birth-death model along with equation (R1) yields

$$p_T = \frac{[S(\lambda + \lambda_t)]^T[1 - S(\lambda + \lambda_t)]}{1 - [S(\lambda + \lambda_t)]^{T+1}} \quad \text{(R4)}$$

Equation (R4) can be used to substitute for p_T in (R2) and (R3). Equation (R3) can then be used to substitute for p^a_{T+} in (R2), yielding a nonlinear equation in the single unknown λ_t. The solution of this equation may be found numerically by any appropriate method. (The method used to obtain numerical results is based on the bisection method of finding roots.) Once λ_t has been found, p_T is given by (R4), and p^a_{T+} is then given by (R3). From the formula for the solution of a birth-death model, the conditional mean queue length $\bar{n}_p$ is given by

$$\bar{n}_p = S(\lambda + \lambda_t)\left[\frac{1 - \rho - C\lambda_t}{1 - p^a_{T+}}\right]\left[\frac{1 - [S(\lambda + \lambda_t)]^T}{[1 - S(\lambda + \lambda_t)]^2} - \frac{T[S(\lambda + \lambda_t)]^T}{1 - S(\lambda + \lambda_t)}\right]. \quad \text{(R5)}$$

All of the performance measures of interest can now be computed. In particular, the mean response time R of tasks is given by:

$$R = \frac{\bar{n}_p(1 - p^a_{T^+}) + Tp^a_{T^+} + \lambda[p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L+1} R_{T^+p}}{\lambda} + \frac{\lambda_t}{\lambda} R_{T^+t} \quad \text{(R6)}$$

where R_{T^+t} denotes the mean delay experienced in being transferred, and R_{T^+p} denotes the processing delay for a task that arrives at the node when the node is at or over threshold, and yet is not transferred due to the transfer limit. The first term in (R6) is derived by applying Little's equation, using expressions for the throughput and mean queue length of locally processed tasks. The throughput of processed tasks is given by λ. The mean queue length of tasks to be processed at the node is given by the mean queue length during a processing phase, multiplied by the probability of being in a processing phase, plus the threshold multiplied by the probability of being in a transferring phase, plus the mean queue length of tasks that are processed locally only because of the transfer limit (which is given by the arrival rate of such tasks multiplied by their mean delay). The second term in (R6) is just the mean number of times a task must be transferred, multiplied by the mean delay experienced each time.

Expressions for R_{T^+t} and R_{T^+p} are derived from the solution of a preemptive priority HOL M/M/1 queue [4]. R_{T^+t} is given by

$$R_{T^+t} = \frac{C}{1 - C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}}} \quad \text{(R7)}$$

R_{T^+p} is given by

$$R_{T^+p} = \frac{S - (S - C)\,C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}}}{\left[1 - C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}}\right]\left[1 - C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}} - S\lambda[p_T(1 - p^a_{T^+}) + p^a_{T^+}]^L\right]}. \quad \text{(R8)}$$

B. Threshold Policy

In the model of the Threshold policy, all transferred tasks arrive when the node queue length is less than the threshold T. Therefore, $\lambda_t(T)$ and $\lambda_t(T^+)$ are both zero. When the node queue length is less than T, the arrival rate of transferred tasks is independent of the node state. Since the probability that the node queue length is less than T is $(1 - p_T)(1 - p^a_{T^+})$, it must be the case that, for $0 \le n \le T - 1$,

$$\lambda_t(n) = \lambda_t^* \quad \text{(T1)}$$

where λ_t^* is defined by

$$\lambda_t^* = \frac{\lambda_t}{(1 - p_T)(1 - p^a_{T^+})}. \quad \text{(T2)}$$

The arrival rate of transferred tasks must equal the rate of task transfers. Since only those tasks that arrive while the queue length is greater than or equal to the threshold T are transferred, under the constraint of a probe limit of L_p,

$$\lambda_t = [p_T(1 - p^a_{T^+}) + p^a_{T^+}]\,\lambda \cdot [1 - [p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p}].$$

This gives

$$\lambda_t^* = \frac{[p_T(1 - p^a_{T^+}) + p^a_{T^+}]}{(1 - p_T)(1 - p^a_{T^+})}\lambda \cdot [1 - [p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p}]. \quad \text{(T3)}$$

The probability of being in a transferring phase is just that portion of the node utilization due to performing task transfers and processing tasks that could not be transferred because of a failure to find a suitable destination. Therefore,

$$p^a_{T^+} = C\lambda_t + S\lambda[p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p+1}$$

Using (T3) to substitute for the exponentiated term and solving for $p^a_{T^+}$ yields

$$p^a_{T^+} = \frac{p_T S\lambda - (1 - p_T)(S - C)\,\lambda_t^*}{1 - (1 - p_T)(S\lambda + (S - C)\,\lambda_t^*)}. \quad \text{(T4)}$$

Consider now the birth–death model of Fig. 1. Using the formula for the solution of a birth–death model along with (T1) yields

$$p_T = \frac{[S(\lambda + \lambda_t^*)]^T(1 - S(\lambda + \lambda_t^*))}{1 - [S(\lambda + \lambda_t^*)]^{T+1}}. \quad \text{(T5)}$$

Equation (T5) can be used to substitute for p_T in (T3) and (T4). Equation (T4) then can be used to substitute for $p^a_{T^+}$ in (T3), yielding a nonlinear equation in the single unknown λ_t^*. The solution of this equation may be found numerically by any appropriate method. (The method used to obtain numerical results is based on the bisection method of finding roots.) Once λ_t^* has been found, p_T is given by (T5), $p^a_{T^+}$ is then given by (T4), and λ_t is then given by (T2).

From the formula for the solution of a birth–death model, the conditional mean queue length $\bar{n}_p$ is given by

$$\bar{n}_p = S(\lambda + \lambda_t^*)\left[\frac{1 - \rho - C\lambda_t}{1 - p^a_{T^+}}\right]\left[\frac{1 - [S(\lambda + \lambda_t^*)]^T}{[1 - S(\lambda + \lambda_t^*)]^2} - \frac{T[S(\lambda + \lambda_t^*)]^T}{1 - S(\lambda + \lambda_t^*)}\right]. \quad \text{(T6)}$$

The mean response time R of tasks is given by

$$R = \frac{\bar{n}_p(1 - p^a_{T^+}) + Tp^a_{T^+} + \lambda[p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p+1} R_{T^+p}}{\lambda} + \frac{\lambda_t}{\lambda} R_{T^+t} \quad \text{(T7)}$$

where R_{T^+t} denotes the mean delay experienced in being transferred, and R_{T^+p} denotes the processing delay for a task that arrives at the node when the node is at or over threshold, and yet is not transferred since a suitable destination has not been found after the maximum number of probes. This equation is quite similar to that for the Random policy, and is derived in a similar manner.

Expressions for R_{T^+t} and R_{T^+p} are derived from the solution of a preemptive priority HOL M/M/1 queue. R_{T^+t} is given by exactly the same equation as for the Random policy

$$R_{T^+t} = \frac{C}{1 - C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}}} \quad \text{(T8)}$$

R_{T^+p} is given by

$$R_{T^+p} = \frac{S - (S - C)\,C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}}}{\left[1 - C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}}\right]\left[1 - C\dfrac{\lambda_t}{p_T(1 - p^a_{T^+}) + p^a_{T^+}} - S\lambda[p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p}\right]}. \quad \text{(T9)}$$

C. Shortest Policy

Iteration is used to evaluate the model of the Shortest policy. In a typical step, a model solution is used to derive new values for the arrival rates of transferred tasks, and a new solution is computed. In the following description of the iteration equations, $pshort_n$ denotes the probability that a node with queue length n is selected when attempting to find a suitable node to which to transfer a task. The following equation gives $pshort_n$, $0 \le n \le T - 1$, in terms of the probe limit L_p, the conditional state probabilities p_n, and the probability of being in a transferring phase $p^a_{T^+}$:

$$pshort_n = \left[1 - \left(\sum_{m=0}^{n-1} p_m\right)(1 - p^a_{T^+})\right]^{L_p} - \left[1 - \left(\sum_{m=0}^{n} p_m\right)(1 - p^a_{T^+})\right]^{L_p} \quad \text{(S1)}$$

The first term in (S1) gives the probability that all of the L_p randomly chosen nodes have queue length greater than or equal to n; the second term gives the probability that all of the L_p randomly chosen nodes have queue length greater than or equal to $n + 1$. The difference of the two terms provides the required probability. Noting that only those tasks that arrive when a node is at or over threshold can be transferred, under the constraint of a probe limit of L_p, and that the arrival rate of transferred tasks must equal the rate of task transfers,

$$\lambda_t = [p_T(1 - p^a_{T^+}) + p^a_{T^+}]\,\lambda \cdot [1 - [p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p}]. \quad \text{(S2)}$$

The probability of being in a transferring phase is just that portion of the node utilization due to performing task transfers and processing tasks that could not be transferred because of a failure to find a suitable destination. Therefore:

$$p^a_{T^+} = C\lambda_t + S\lambda[p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p+1}. \quad \text{(S3)}$$

The arrival rates $\lambda_t(n)$, for $0 \le n \le T - 1$, are then given by

$$\lambda_t(n) = \frac{\lambda_t}{p_n(1 - p^a_{T^+})} \frac{pshort_n}{1 - [p_T(1 - p^a_{T^+}) + p^a_{T^+}]^{L_p+1}}. \quad \text{(S4)}$$

The formula for the solution of a birth–death model yields, for $1 \le n \le T$,

$$p_n = p_{n-1}\left[1 + \frac{\lambda_t(n - 1)}{\lambda}\right]\rho. \quad \text{(S5)}$$

Finally, p_0 is given by

$$p_0 = \frac{1 - \rho - C\lambda_t}{1 - p^a_{T^+}} \quad \text{(S6)}$$

In one iteration of the method used to compute numerical results, (S1)–(S6) are applied to a model solution in order, yielding a new model solution. The conditional state probabilities of the new solution then are normalized to sum to one by scaling the probabilities p_k for $k > 0$. The iteration stopping criterion is based on comparing old and new conditional mean queue length values, as obtained from the conditional state probabilities. Empirically, the iteration is insensitive to the initializations used. Once the solutions of (S1)–(S6) have been obtained, the mean response time R of tasks is given by the same equations as in the analysis of the Threshold policy.

Appendix C
Simulation Results

Experimentation with an event-driven simulation program has provided validation of the decomposition approximation utilized in our analytic models. The simulation program uses the same system model as do the analytic models, but does not make the decomposition approximation.

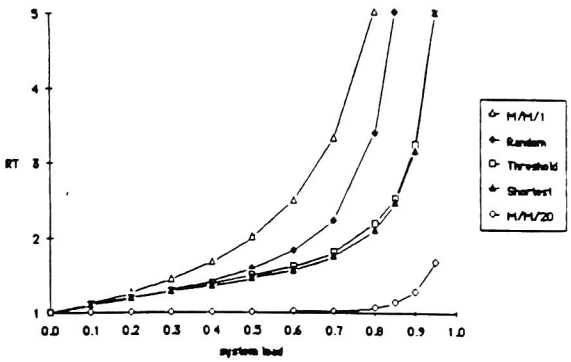


Fig. 11. Simulation results: response time versus load ρ. S(task service time) = 1. C(cost of task transfer) = 0.1. T(threshold) = 2. L_p(probe limit for Threshold and Shortest) = 3. L_t(transfer limit for Random) = 1.

In Fig. 11 we present a small sample of the results of our simulation experiments. A system with 20 nodes has been simulated. Fig. 11 should be compared to Fig. 2; for the Random, Threshold, and Shortest policies the former figure shows the simulation results that correspond to the analytic results of the latter figure. (Fig. 11 also includes the analytic results for M/M/1 and M/M/20 for comparison purposes.) Note the close correspondence between the two figures, both with respect to the absolute values of the performance measures (particularly at low to moderate loadings), and with respect to the indicated relative performance of the three load sharing policies.

Acknowledgment

D. Towsley of the University of Massachusetts discussed these issues extensively with us. K. Sevcik of the University of Toronto provided comments on an earlier draft.

References

[1] A. Barak and A. Shiloh, "A distributed load balancing policy for a multicomputer," Dep. Comput. Sci., Hebrew Univ. of Jerusalem, Jerusalem, Israel, 1984.

[2] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 341–349, July 1979.

[3] R. Bryant and R. A. Finkel, "A stable distributed scheduling algorithm," in *Proc. 2nd Int. Conf. Distributed Comput. Syst.*, 1981, pp. 314–323.

[4] L. Kleinrock, *Queueing Systems: Volume I—Theory*. New York: Wiley, 1976.

[5] L. Kleinrock and S. S. Lam, "Packet switching in a multiaccess broadcast channel: Performance evaluation," *IEEE Trans. Commun.*, vol. COM-23, pp. 410–423, Apr. 1975.

[6] A. Kratzer and D. Hammerstrom, "A study of load levelling," in *Proc. IEEE Fall COMPCON*, 1980, pp. 647–654.

[7] P. Krueger and R. A. Finkel, "An adaptive load balancing algorithm for a multicomputer," Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep. 539, Apr. 1984.

[8] S. S. Lam and L. Kleinrock, "Packet switching in a multiaccess broadcast channel: Dynamic control procedures," *IEEE Trans. Commun.*, vol. COM-23, pp. 891–904, Sept. 1975.

[9] E. D. Lazowska, J. Zahorjan, D. R. Cheriton, and W. Zwaenepoel, "File access performance of diskless workstations," Dep. Comput. Sci., Univ. Washington, Seattle, Tech. Rep. 84-06-01, June 1984.

[10] J. D. C. Little, "A proof of the queueing formula $L = \lambda W$," *Oper. Res.*, vol. 9, pp. 383–387, May 1961.

[11] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," in *Proc. ACM Comput. Network Performance Symp.*, 1982, pp. 47–55.

[12] M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," in *Proc. 9th ACM Symp. Operat. Syst. Principles*, 1983, pp. 110–119.

[13] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 85–93, Jan. 1977.

[14] ——, "Critical load factors in two processor distributed systems," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 254–258, May 1978.

[15] A. N. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *J. ACM*, vol. 32, pp. 445–465, Apr. 1985.

[16] F. A. Tobagi and L. Kleinrock, "Packet switching in radio channels: Part IV—Stability considerations and dynamic control in carrier sense multiple access," *IEEE Trans. Commun.*, vol. COM-25, pp. 1103–1119, Oct. 1977.

Derek L. Eager received the B.Sc. degree in computer science from the University of Regina, Regina, Sask., Canada, in 1979, and the M.Sc. and Ph.D. degrees in computer science from the University of Toronto, Toronto, Ont., Canada, in 1981 and 1984, respectively.

He is currently an Assistant Professor in the Department of Computational Science at the University of Saskatchewan, Saskatoon. His research interests are in the areas of performance modeling and distributed systems.

Edward D. Lazowska received the A.B. degree from Brown University, Providence, RI, in 1972 and the Ph.D. degree in computer science from the University of Toronto, Toronto, Ont., Canada, in 1977.

Since 1977 he has been on the faculty of the Department of Computer Science at the University of Washington, Seattle, where he recently returned after a sabbatical leave at Digital Equipment Corporation's Systems Research Center. His research interests fall within the general area of computer systems: modeling and analysis, design and implementation, and distributed systems.

Dr. Lazowska is the Chairman of SIGMETRICS, the Association for Computing Machinery's Special Interest Group concerned with computer system performance.

John Zahorjan received the Sc.B. degree in applied mathematics from Brown University, Providence, RI, in 1975, and the M.Sc. and Ph.D degrees in computer science from the University of Toronto, Toronto, Ont., Canada, in 1976 and 1980, respectively.

Presently, he is an Associate Professor of Computer Science at the University of Washington, Seattle. His active research interests include performance modeling of computer systems, load sharing policies in distributed systems, and issues in naming in distributed systems. He is an author of papers in these areas, and is coauthor of a recent book on performance modeling of computer systems using queueing network models.

Appendix C. "Load Balancing in Homogeneous Broadcast Distributed Systems"

Load Balancing in Homogeneous Broadcast Distributed Systems

by
Miron Livny and Myron Melman
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

ABSTRACT

Three different load balancing algorithms for distributed systems that consist of a number of identical processors and a CSMA communication system are presented in this paper. Some of the properties of a multi-resource system and the balancing process are demonstrated by an analytic model. Simulation is used as a mean for studying the interdependency between the parameters of the distributed system and the behaviour of the balancing algorithm. The results of this study shed light on the characteristics of the load balancing process.

INTRODUCTION

Distributed processing systems are characterized by resource multiplicity and system transparency [1]. Every distributed system consists of a number of autonomous resources that interact through a communication system. From the user's point of view this set of resources acts like a 'single virtual system'. As he submits a task for execution he does not and should not consider either the internal structure or the instantaneous load of the system. It is the duty of the system's load balancing algorithm to control the assignment of resources to tasks and to route the tasks according to these assignments.

The stochastic properties of the tasks - arrival and execution times cause resource contentions that lead to the establishment of queues. The existence of queues of waiting tasks demands dynamic reconsideration of previous assignments. The assignment algorithm is motivated by the desire to achieve better overall performance relative to some selected metric of system performance. The strategy of the load balancing algorithm has a strong effect on the utilization of the system resources and determines its overall performance. The purpose of this paper is to investigate the behaviour of the load balancing process in broadcast distributed systems.

The problem of resource allocation in an environment of cooperating autonomous resources and its relationship to system performance is a major issue associated with the design of distributed systems [2]. A number of studies of this issue have been reported [3] [4] [5] [6]. Most of these studies deal with distributed systems that utilize central elements, such as a job dispatcher, a shared memory , a main processor, or with systems that consist only of two processors. This paper deals with distributed load balancing algorithms for homogeneous distributed systems whose communication system consists of a broadcast medium. There are no central elements in the system and the balancing algorithm is distributed among the resources. The policy of the algorithm is to minimize the expected turnaround time of the tasks.

Initially a simple analytic model is used for demonstrating some of the properties of a multi-resource system and the balancing process. Then three different load balancing algorithms for broadcast distributed systems are defined and discussed. The last part of the paper presents results of the simulation study. In the study, the three algorithms were simulated under various operating conditions. The results demonstrate the interdependency between the parameters of the distributed system and the behaviour of the balancing algorithm.

1982 ACM 0-89791-069-9/82/0400-0047 $00.75

LOAD BALANCING

In a distributed system it might happen that a task waits for service at the queue of one resource while at the same time another resource which is capable of serving the task is idle. A load balancing algorithm whose goal is to minimize the expected turnaround time of the tasks will tend to prevent the system from reaching such a state.

Assume a system of N identical[1] and independent M/M/1 queueing systems [7]. Let P_{wi} be the probability that the system is in a state in which at least one customer waits for service and at least one server is idle then

$$P_{wi} = \sum_{i=1}^{N} \binom{N}{i} Q_i H_{N-i} = (1-Po^N)(1-Po^N-(1-Po)^N)$$

where

$Q_i = Po^i$ is the probability that a given set of i servers are idle

$H_i = (1-Po)^i-(Po(1-Po))^i$ is the probability that a given set of i servers is not idle and at one or more of them a task waits for service

$Po = 1-\frac{\lambda}{\mu}$ is the probability that a server is idle.

Fig. 1 shows the value of P_{wi} for various values of server utilizations, 1-Po, and number of servers N. The curves of the figure indicate that for practical values of ρ, P_{wi} is remarkably high and that in systems with more than ten servers almost all the time a customer is waiting for service and another server is idling.

The high value of P_{wi} indicates that by balancing the instantaneous load of the multi-resource system their performance can be considerably improved. Note that the average load of a server is the same for all servers. The shape of the curves shows that for a given number of servers P_{wi} reaches its maximum value when the servers are utilized during 65% of the time. As the utilization of the servers increases past the level of 65% P_{wi} decreases. This property of P_{wi} indicates that a 'good' load balancing algorithm should work less when the system is heavily utilized. It is clear that the same thing is true for systems that are idle most of the time.

A reduction in P_{wi} of a multi-resource system will cause an improvement of the expected turnaround time, W, of the tasks. If the servers are interconnected by a communication system P_{wi} can be

[1] All the systems have the same arrival, λ, and service, μ, rates.

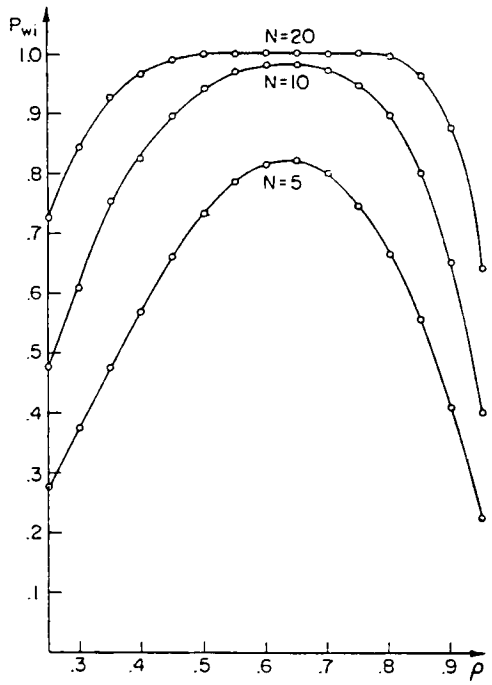


figure: 1 P_{wi} as a function of ρ

reduced by transfering tasks from one queue to another. These transfers affect the utilization and consequently the performance of the communication system and can be considered as the price paid for the reduction of W.

The expected turnaround time of the above multi-resource system will be minimal if P_{wi} will be zero. In such a case the system will behave like an M/M/N (single queue N servers) system [7]. P_{wi} can be reduced to zero only if the servers are inter-connected by a communication system whose task transfer rate is much higher than the service rate of the servers. In a system where P_{wi} is zero a task will be transferred from one queue to another when one of the following events occurs:

1. A task arrives at a busy server and there are less than N tasks in the system.
2. A server completes the service of a task, no other tasks are waiting in its queue and there are more than N tasks in the entire system.

Therefore a lower bound to the rate of tasks transferred in order to minimize W is given by

$$LT = \sum_{i=1}^{N-1} (\lambda i P_i + \mu(N-1)P_{N+i})$$

where P_i is the probability of having i tasks in an M/M/N system [7]. The first element of the summation is the rate of transfers caused by the arriving tasks (the first event). The second element is part of the transfer rate caused by the departing tasks (the second event).

Fig. 2 gives the values of the lower bound LT as a function of the number of servers for various arrival rates, . Note that a considerable number of tasks has to be transferred in order to achieve the performance of a M/M/N system. For systems with more than ten servers almost one out of λ^{-1} tasks are transferred.

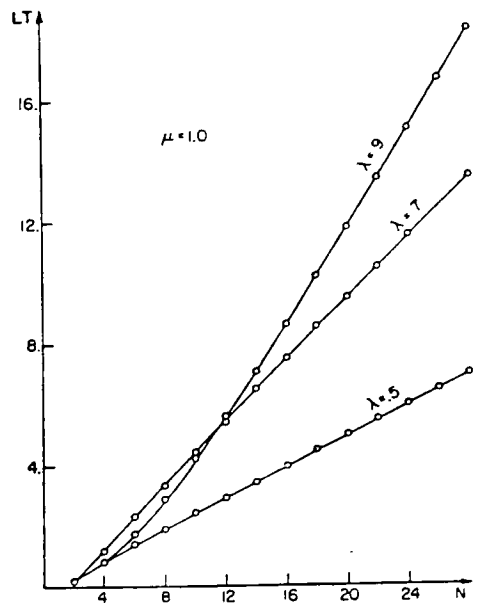


figure: 2 Lower Bound on task transfer rate in an M/M/N like system vs. nuber of servers

These results indicate that in systems where task transmission time is not negligable the load balancing process will utilize a large portion of the capacity of the communication system. The utilization of the communication system will determine the delays associated with the transmission of a task or any other message. These delays will cause an increase in P_{wi} and therefore an increase in W. The amount of traffic generated by the balancing algorithm has a major effect on its ability to improve the performance of the system. Fig. 2 shows that in order to achieve the optimal performance, $P_{wi} = \emptyset$, a large portion of the tasks have to be transferred.

THE DISTRIBUTED SYSTEM MODEL

The model describes a homogeneous N-server distributed system. The system consists of N identical nodes and a communication channel. Every node has a processor P, a communication processor CP and a queue, Fig. 3. The channel is a passive broadcast medium (radio or coxial/fiber cable) with a CSMA-CD (carrier sense multiple access collision detection) access method. The access to the channel and the transmission of messages is controlled by the CP according to the ETHERNET protocol [8] [9].

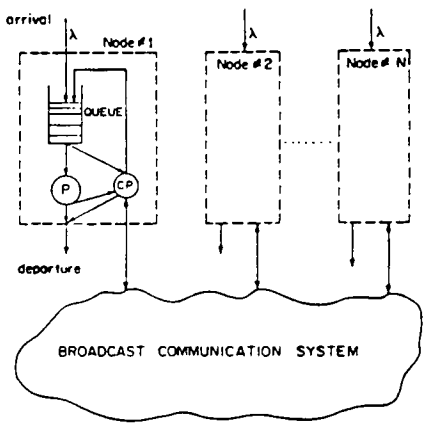


figure: 3 The Broadcast Distributed System

Tasks arrive independently at each node and join the queue. The queueing discipline at all the nodes is FIFO (first-in-first-out). The arrival rate of each stream of tasks is λ and the inter-arrival time has a negative exponential distribution. The task arrival process to the entire system consists of N identical independent poisson processes with a total rate of N .

The service time demand of the tasks has a negative exponential distribution and the mean service time is μ^{-1}. The tasks leave the system after being served, and depart from the same node at which they had entered the system. It is assumed that the system operates in steady-state conditions ($\lambda < \mu$). The utilization of the servers is $\rho = \frac{\lambda}{\mu}$.

The number of tasks at node i (waiting for service or being served) is denoted by N_i and $ST=(n_1,\ldots,n_N)$ describes the state of the system. A state of the system is defined as unbalanced if there are two servers i and j such that $n_i - n_j > 1$. The unbalance factor of a state ST is defined as

$$UBF = \begin{cases} \underset{\emptyset<i,j\leq N}{\text{MAXIMUM}}((n_i-n_j)n_j^{-1}) & \text{if ST is UNB} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that if the system is in an unbalanced state and one of the servers is idle the UBF of the state is infinite.

The purpose of the channel, Fig. 3, is to transfer tasks from one node to the other in order to improve the expected turnaround time of a task. The flow of tasks via the channel is governed by a distributed load balancing algorithm.

A node that wants to transfer one of its waiting tasks to another node will send it a message that describes the task. The message has to contain all the external data a server needs in order to identify and serve the task. In this model it is assumed that this amount of data, T, is fixed and the same amount of data is sent from the node that executed the task back to the entrance node of the task. Such a transmission takes place only when the task was not served by the node at which it entered the system. The balancing rate of the system, α, is defined as $\frac{n-1}{CT}$ where C is the capacity of the communication channel. The factor β expresses the ratio between the mean execution time of a task and the time needed to transfer a task from one node to another. Note that when β is zero the system becomes an N(M/M/1) queueing system and when β becomes very large the system behaves like an M/M/N system.

LOAD BALANCING ALGORITHMS

A distributed load balancing algorithm is composed of two main elements - the control law element and the information policy element. The control law determines when, from where and to whom to transfer a waiting task. The decision is made according to the current available information on the state of the system. It is the function of the information policy to collect data for the control element concerning the load of the system resources. Both elements use the communication system for carrying out their functions. The control element sends messages that describe tasks and the information element sends 'status messages' that contain data on the system's load.

The delays associated with the transmission of a message may lead to the execution of a wrong operation by the balancing algorithm. As a result of such an operation a task is placed in a queue that has more waiting tasks than the queue from which the task has been removed. The balancing process faces a 'transmission dilemma' because of the two opposing impacts the transmission of a message has on the overall performance of the system. On the one hand the transmission improves the ability of the algorithm to balance the load. On the other hand it raises the expected queueing time of messages because of the increase in the utilization of the channel. The net impact of a message transmission on the overall performance of the system depends on the balancing rate of the communication system, the number of nodes and the rate at which tasks arrive at the system.

Three different distributed load balancing algorithms for broadcast distributed system are defined in this study. From the load balancing point of view broadcast communication systems have two advantages:

1. Uniform distance - the expected time that is needed to transfer a message from one node to another is the same for all pairs of nodes. Therefore all the nodes are equal-priority candidates for receiving a waiting task. Only the relative load of the nodes has to be considered by the control law.
2. Messages broadcast the capability of the communication system to broadcast messages improves the ability of the algorithm to get a global and updated description of the system status.

The communication system consists of a single transmission resource and therefore it can not transfer a number of messages simultaneously. The high rate of message transfers generated by the balancing process (fig. 2) requires that the balancing rate of the system will be high.

The state broadcast algorithm - STB. The STB balancing algorithm utilizes both the broadcast and the uniform distance properties of the communication system. The information policy of the algorithm is based on status broadcast messages. Whenever the state of the node changes, because of the arrival or departure of a task, the node broadcasts a status message that describes its new state. This information policy enables each node to hold its own updated copy of the system state vector, SSV , and guarantees that all the copies are identical. The information contained in the $SSV=(s_1, .. ,s_N)$ gives the node a global and updated picture of the system state and enables the control law to base its decisions on the state of the whole system. Note that SSV may differ from ST due to transmission delays. The distributed control law of the STB algorithm will transfer a waiting task from node i to node j if the following conditions are fulfilled.

1. $s_i - s_j > 1 + (BT \cdot s_j)$ where BT is a parameter that controls the balancing threshold of the algorithm.

2. $((s_i > s_k)$ or $(s_i = s_k$ and $i \geq k))$ for all $k = 1,\ldots,N$.

3. $s_j \leq s_k$ for $k = 1,\ldots,N$.

When more than one node has a minimal number of waiting tasks the selection of the destination node is made randomly.

The broadcast idle algorithm - BID . The BID algorithm is based on a less liberal information policy. Under this policy a node broadcasts a status message when it enters an idle state. The message alerts all the other nodes and causes them to activate the control element of the algorithm. The control law of the BID algorithm consists of the following steps:

1. If $n_i > 1$ go to step 2, else terminate the algorithm.
2. Wait $D \cdot n_i^{-1}$ units of time. D is a parameter of the algorithm. Its value depends on the properties of the communication system.
3. Broadcast a reservation message if no other node has broadcasted such a message during the time-out period.[2] If another node has succeeded to broadcast a reservation message terminate the algorithm.
4. Wait for a reply message. The reply will be positive if the node that has broadcasted the idle message is still idle. The node will send a reply in any case.
5. If the reply is positive and $n_i > 1$ transfer a task to the idle node, else terminate the algorithm.

The purpose of the state-dependent time-out period is to give nodes with greater load a better chance to transfer a task to the idle node.

The poll when idle algorithm - PID. The information policy of both previous algorithms is based on broadcast messages. The information policy of the PID algorithm is based on polling . The node starts to poll a subset of the system nodes whenever it enters an idle state. The sequence of the polling operation of the PID algorithm is the following:

1. Randomly select a set of R nodes $(a_i, \ldots, a_R)$ and set $J = 1$. R is a parameter of the algorithm.
2. Send a message to node a_j and wait for a reply.
3. Receive the reply message. Node a_j will either send back one of its waiting tasks, if there are any, or an 'empty queue' reply.
4. If the node is still idle and $j < R$, increment j and go to step 2 else terminate the polling.

The STB algorithm attempts to prevent the system from being in a state in which the UBF is greater than BT whereas the two other algorithms decide to transfer a task only when the UBF of the state is infinite. The STB algorithm is motivated by the assumption that by keeping the UBF of the system below BT the probability that the system will be in a state with an infinite UBF will decrease. The IDB and PID algorithms assume that because of the 'transmission dilemma' it is more important to minimize the channel utilization than to keep the UBF below a finite level.

SIMULATION STUDY

All the above algorithms aspire to improve the performance of the distributed system by balancing the instantaneous load of the system resources, each one in its own way. In order to evaluate the algorithms their performance has to be predicted and the relation between their behaviour and the parameters of the system studied.

The balanced distributed system can be modeled as a queueing network. Because of the dynamic routing of the tasks the queueing model has no feasible numerical solution. Therefore simulation has to be used as a means to predict the performance of the model.

For this study three discrete time simulation models were written using SIMSCRIPT II.5. Each model describes a different algorithm. In all the models it was assumed that there are no delays associated with the control operations of the balancing algorithm. The only delays considered are communication delays. The communication is carried out according to the ETHERNET protocol and the effect of collisions is included in the simulation model. Table 1 lists the numerical values of the simulation parameters.

The expected turnaround time, W, of a task in an M/M/N queueing system with a task arrival rate of $N \cdot \lambda$ is a monotonic decreasing function of N [7]. Although the addition of another server increases the rate at which tasks arrive at the system the supplemental node decreases the expected queueing time of a task.

The effect of the number of nodes, N, on the W of the distributed system is demonstrated by Fig.

[2] If the transmission of the message is delayed because of collisions the same condition is tested before an attempt to retransmit the message is made.

STB - □ ; IDB - ▲ ; PID - ● ; M/M/N - Δ ; M/M/1 - o ;

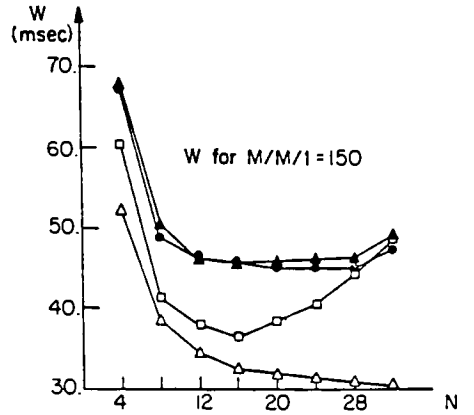


figure: 4 expected turnaround time vs. number of servers β=40

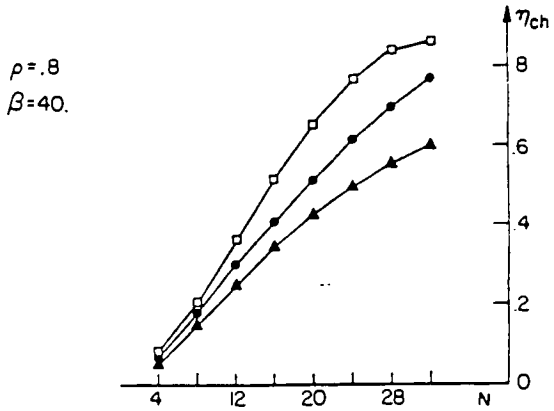


figure:4.a channel utilization vs. number of servers β =40

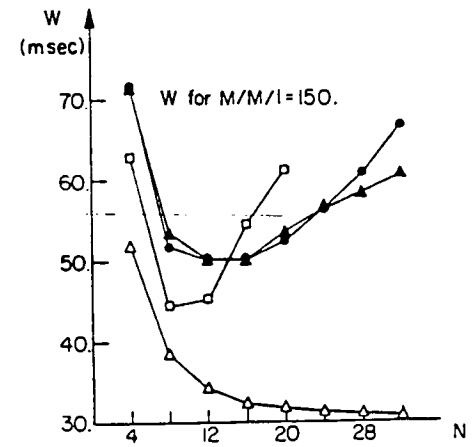


figure: 5 expected turnaround time vs. number of servers β=20

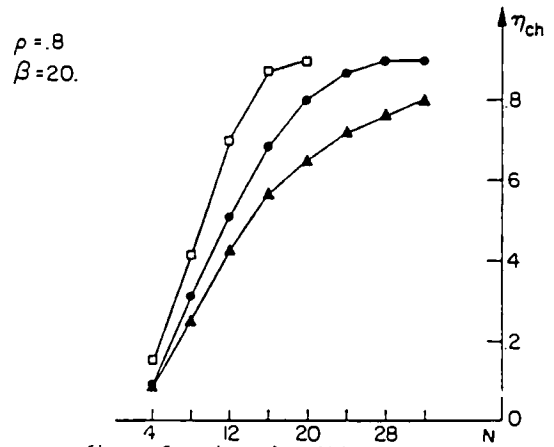


figure:5.a channel utilization vs. number of servers β =20

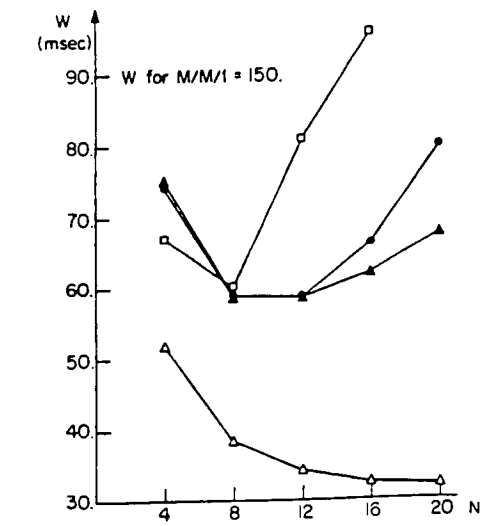


figure: 6 expected turnaround time vs. number of servers β =10

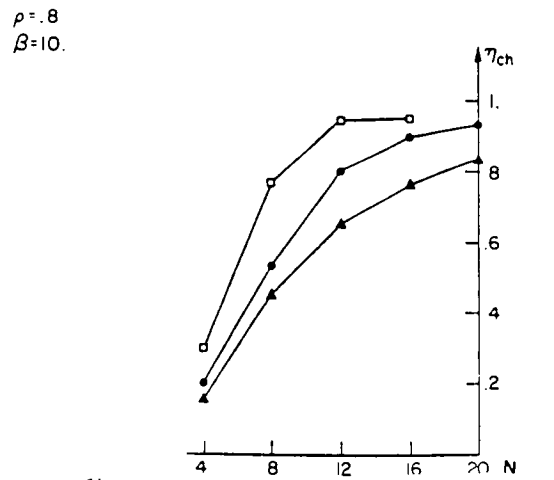


figure:6.a channel utilization vs. number of servers β =10

STB - □ ; IDB - ▲ ; PID - ● ; M/M/N - Δ ; M/M/1 ○ ;

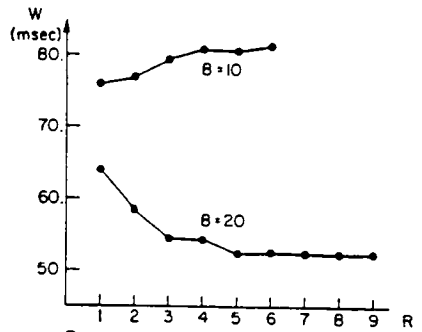


figure: 7 expected turnaround time vs. the R parameter

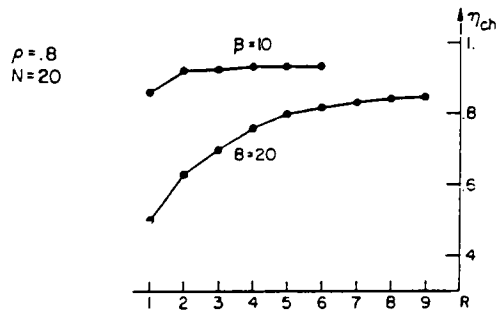


figure:7.a channel utilization vs. the R parameter

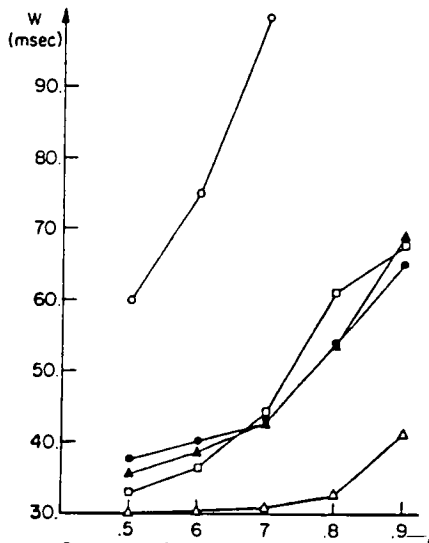


figure: 8 expected turnaround time vs. servers utilization

N=16
β=20.

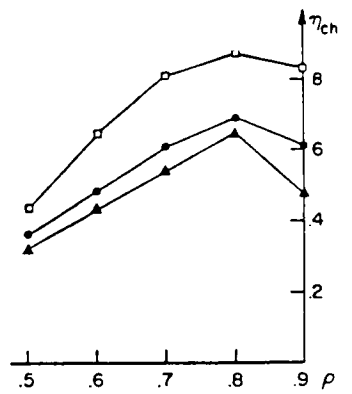


figure:8.a channel utilization vs. servers utilization

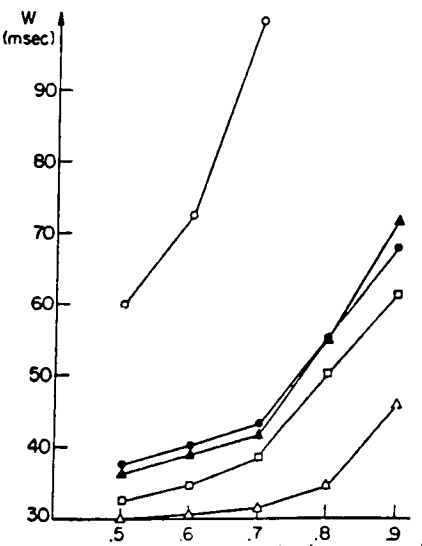


figure: 9 expected turnaround time vs. servers utilization

N=12
β=20.

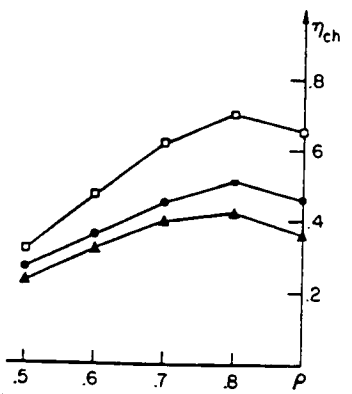


figure:9.a channel utilization vs. servers utilization

TABLE 1
Values of simulation parameters

channel transmission rate	3 Mbit/sec
slot length (see [8])	3.2 μ sec
retransmission delay uniformly distributed between 28. CN sec and 50. CN sec where CN is the collision counter (see [9])	
transmission time of status/ reservation/polling message	50 μ sec
expected task service time (μ^{-1})	30 msec
BT parameter of STB algorithm	1.9
D factor of IDB algorithm	1.0 msec
R parameter of PID algorithm	5
balancing rate β	10,20,40 (β=10 means T=1Kbyte)
simulation length · λ^{-1}	30.0 sec

4, 5, 6. The figures give the W of the three algorithms for three different balancing rates, β. In all the cases the balanced system has a considerably better W then the unbalanced system, M/M/1.

For a system with β=10 the expected waiting time of a task is decreased by at least 70%. The degree to which the balancing algorithm approaches the optimal W of an N server system (M/M/N) depends both on the balancing rate of the system and on the number of nodes. The turnaround time curves show that an increase in the number of nodes in a balanced distributed system has two counteracting effects. On the one hand it improves the probability that a waiting task will be transferred to an idle server, as in an M/M/N system. But on the other hand it raises the utilization of the communication channel, Fig. 4a, 5a, 6a. Higher channel utilization causes a slow-down in the balancing process resulting from an increase in message queueing delays. The net result of these two effects will determine whether the increase in N improves, does not affect, or deteriorates the expected turnaround time of a task. Every algorithm reaches a point, N_m, at which an addition of another server will cause an increase in W. The value of N_m depends on the algorithm and balancing rate of the system. Note that in all cases when N is less than the N_m of the STB algorithm the W of this algorithm is the smallest. After it reaches its minimal value the W of the STB algorithm increases in a steep slope until it becomes greater than the W of the other algorithms. The degradation in the performance of the STB algorithm is caused by the increase in transmission delays. The BID and STP algorithms are less sensitive to the utilization of the channel. Therefore there is a wide range of N values for which they have almost the same performance. The reservation mechanism of these algorithms helps them to prevent 'wrong operations'. On the other hand the two algorithms transfer tasks only when at least one of the servers is idle. Therfore an increase in transmission delays increases the P_{wi} of the system. The IDB and PID algorithms have almost the same W under all the conditions simulated.

The balancing process utilizes a large portion of the communication channel capacity, Fig. 4a, 5a, 6a. The STB algorithm has the highest channel utilization and the IDB the smallest. The communication activity of the PLI algorithm can be easily controlled by the value of the R parameter. Fig. 7, 7a show how both channel and W depend on the size of the polling set of the algorithm. Note that for β = 10 a decrease in R causes a reduction in both W and the channel utilization.

Fig. 8 and 9 show how the balancing process reacts to changes in the utilization of the servers, ρ. For all values of ρ that were simulated the balanced algorithms improve considerably the expected turnaround time of the tasks. Note that the relative performance of the algorithms depend on the utilization of the servers.

Fig. 8a, 9a show that when the system is heavily utilized, ρ>.8, an increase in the utilization of the system causes a decrease in the channel utilization. Although the throughput of the system increases, the amount of transfers needed to balance the system decreases.

CONCLUDING REMARKS

In the opening analysis it was shown that the expected queueing time of a task in a distributed system can be reduced by means of load balancing. The results obtained from the simulation studies give a quantitative description to this ability. The results presented demonstrate the strong dependency between the performance of the balancing algorithm and the system parameters.

The purpose of the study was to shed light on the load balancing process in homogeneous broadcast distributed systems. The three algorithms that were defined in the course of the study represent three different approaches to the distributed load balancing problem. The simulation results show

that each approach is the 'best' under certain conditions. The dependency between the behaviour of the algorithm and the parameters of the system deters from any attempt to select the ultimately 'best' algorithm. For these algorithms, as for other distributed control algorithms, there is no absolute answer to the question 'is algorithm A better then B' (see [10]). Therefore getting a better uderstanding of the processes involved in distributed load balancing has to be the aim of a study of this type of algorithms.

Three main conclusions can be derived from the simulation study:

1. Higher resource multiplicity does not necessarily result in better turnaround time. Every algorithm reaches a point at which an increase in the number of servers decreases the performance of the system. Therefore when a number of servers is given it might be better, from the W point of view, to assemble them into two or more systems than to integrate them into one system.
2. The balancing process has a high communication activity. This has been predicted by the analytic analysis and is demonstrated by the results of the simulation runs.
3. The selection of the control law and information policy should depend on the expected transmission delays of the balanced system. The 'transmission dilemma' is an important element of the balancing process.

This study is a part of an ongoing research in distributed load balancing systems. In the coming stages some of the restrictions of the model presented here will be released and distributed systems with other communication disciplines will be considered.

REFERENCES

(1) P. H. Enslow, Jr. "What is a Distributed Data Processing System ," Computer, Jenuary 1978, pp. 13-21.

(2) R. H. Echhouse, Jr., J. A. Stankovic, "Issues in Distributed Processing An Overview of Two Workshops," Computer, January 1978, pp. 22 26.

(3) H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Trans. of Software Engin., Vol. SE-3, No. 1, January 1977, pp. 85-93.

(4) A. Kratzer, D. Hammerstrom., "A Study of Load Leveling," Proceedings of COMPCON, Fall 1980, pp. 647-654.

(5) Y. C. Chow, W. H. Kohler, "Dynamic Load Balancing in Homogeneous Two-Processor Distributed Systems," International Symposium on Computer Performance, Modeling, Measurement and Evaluation, August 1977, pp. 49-52.

(6) Y. Eran, M. Livny, M. Melman, "Modeling and Evaluation of a Tree Structured Network: a Case Study," Proceedings of MELECON 1981.

(7) L. Kleinrock, "Queueing Systems Vol. 1: Theory," Wiley, New York, 1975.

(8) R. M. Metcalfe, D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Xerox Research Report, csl-80-2, 1980.

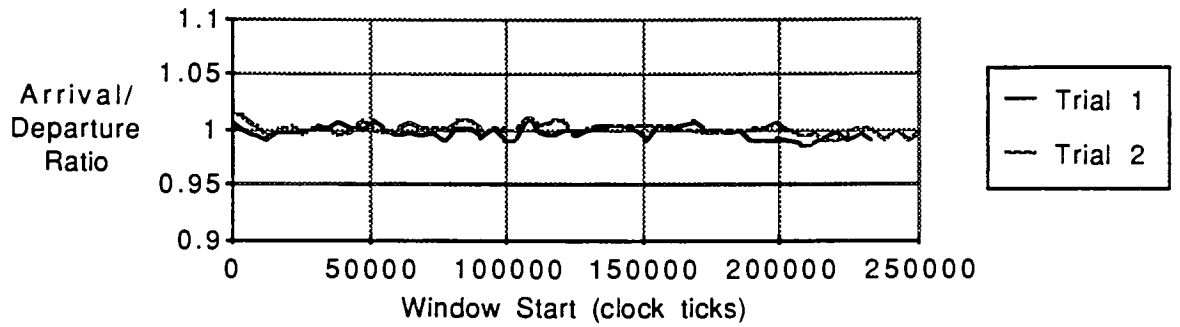
(9) A. K. Agrawala, R. M. Bryant, J. Agre, "Analysis of an Ether-Like Protocol," Computer Networking Symposium 1977, pp. 104-111.

(10) J. M. Mcquillan, I. Richer, E. C. Rosen, "The New Routing Algorithm for the ARPANET," IEEE Trans. on Communications, vol. COM-28, No. 5, May 1980, pp. 711-719.

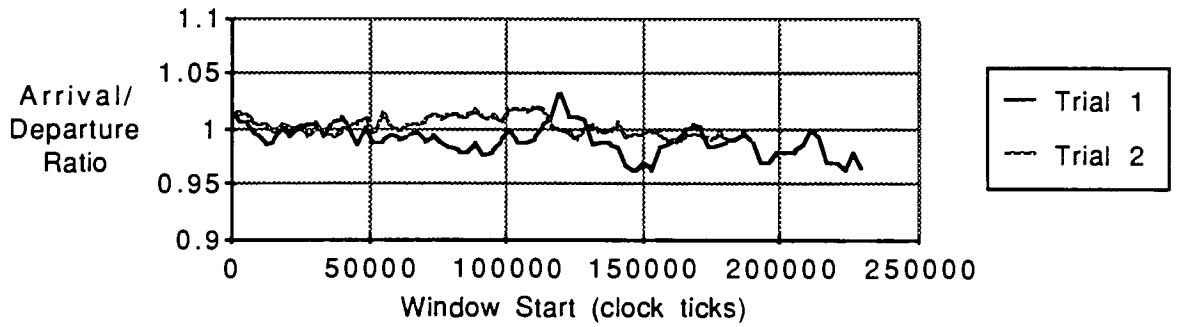
Appendix D. Stability Reports

The following charts are intended to show any abrupt changes in the flow of jobs caused by abnormal job terminations. The data represent the ratio of arrivals to departures within a time window of about half the duration of the given experiment. The window's position is given by the X axis. About fifty data points were calculated for each experiment.

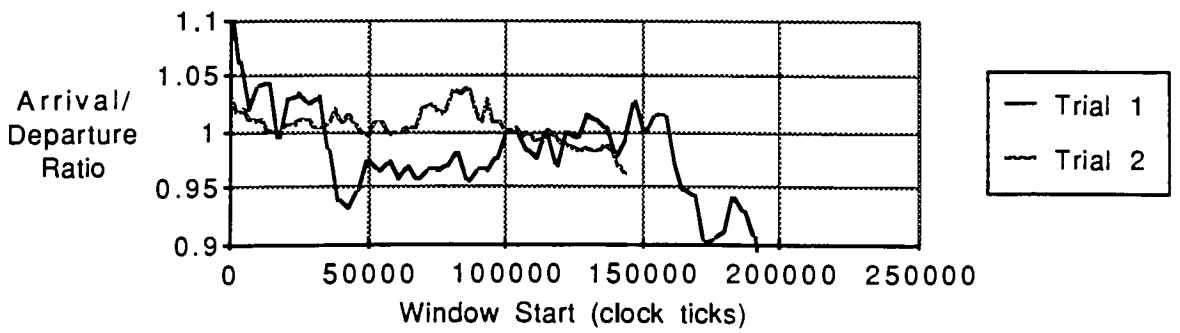
No Load Balancing, Load = .5



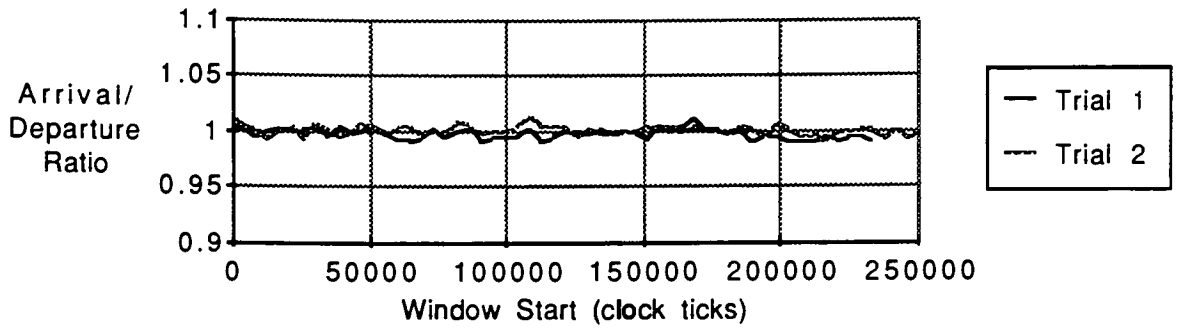
No Load Balancing, Load = .7



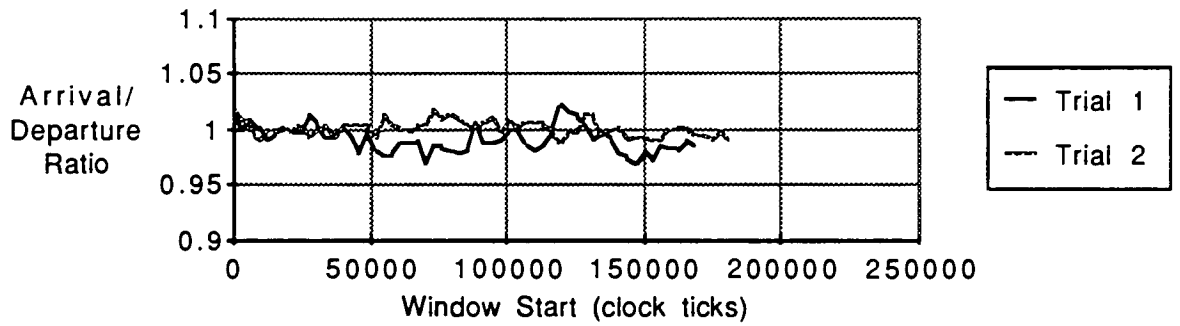
No Load Balancing, Load = .9



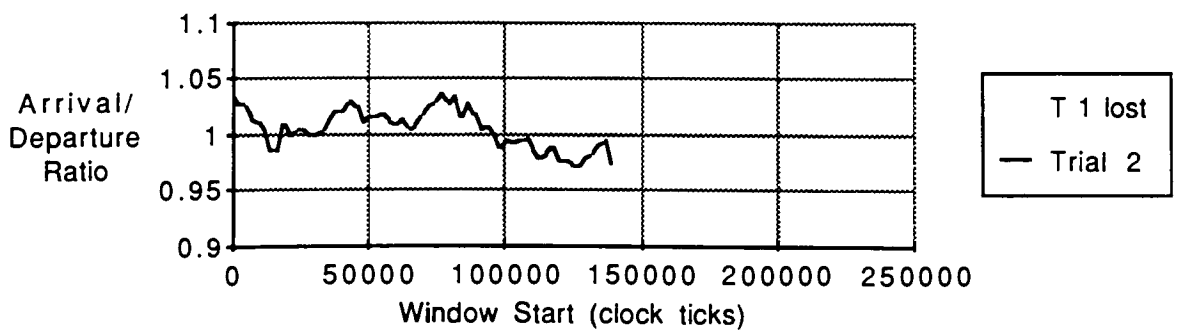
Random Algorithm, Load = .5

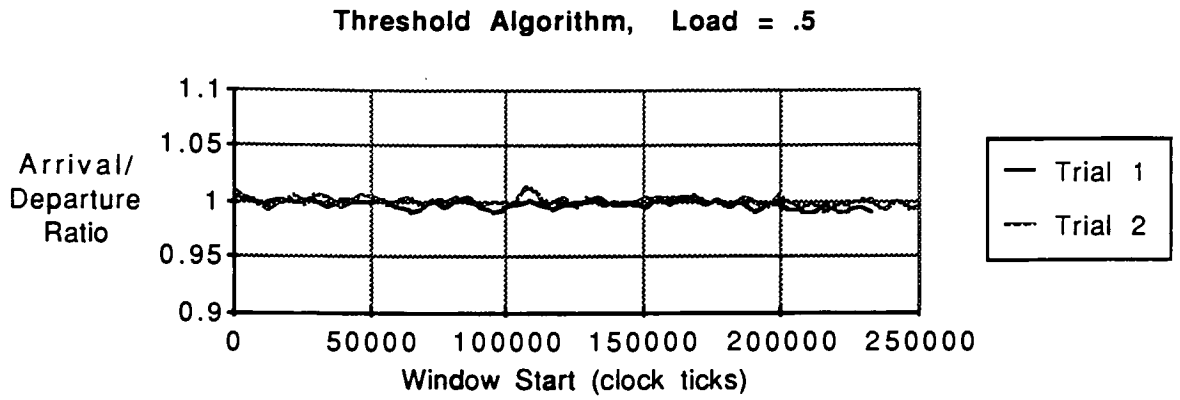


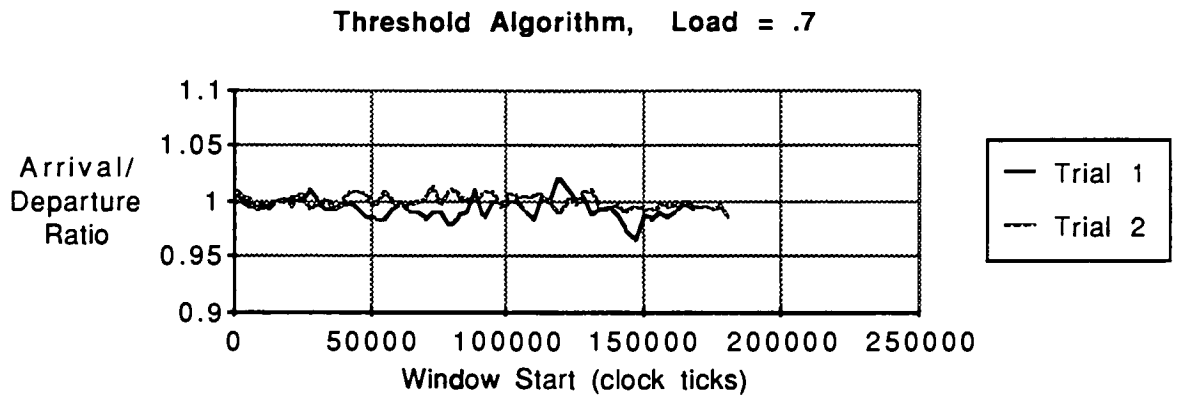
Random Algorithm, Load = .7

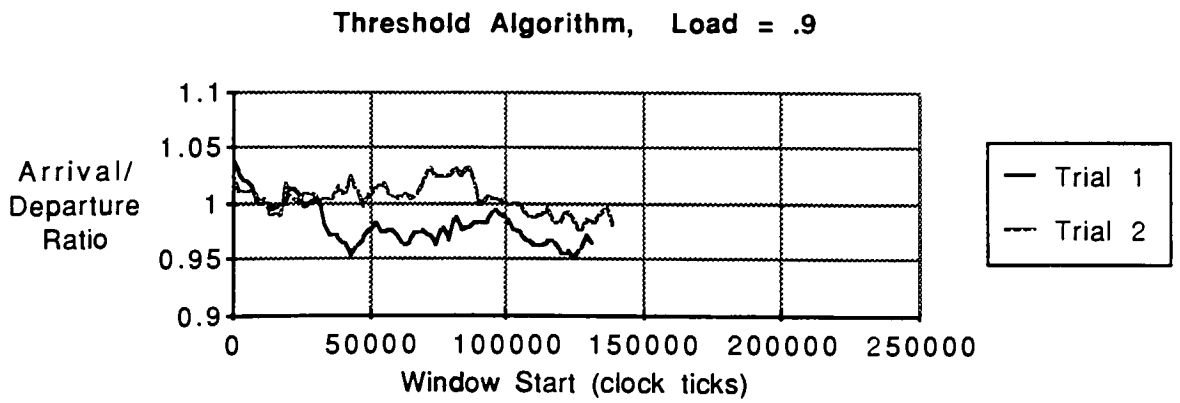


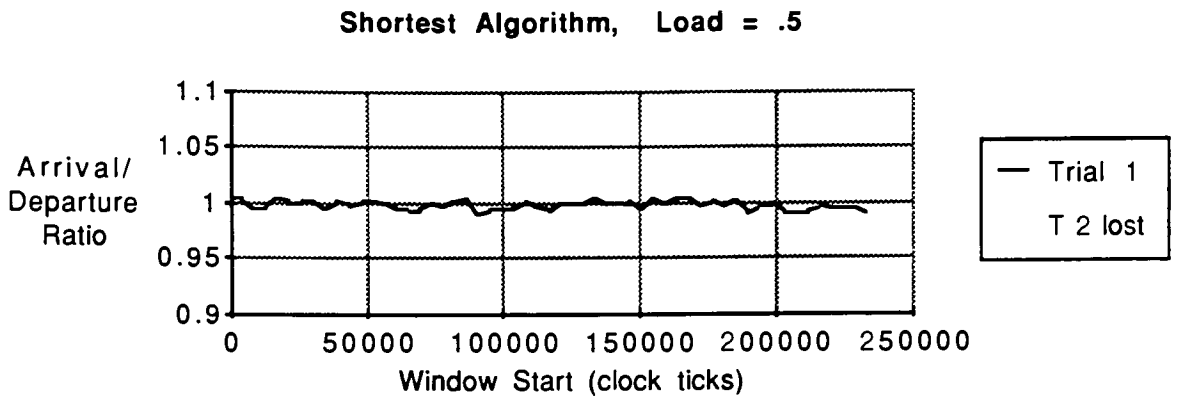
Random Algorithm, Load = .9

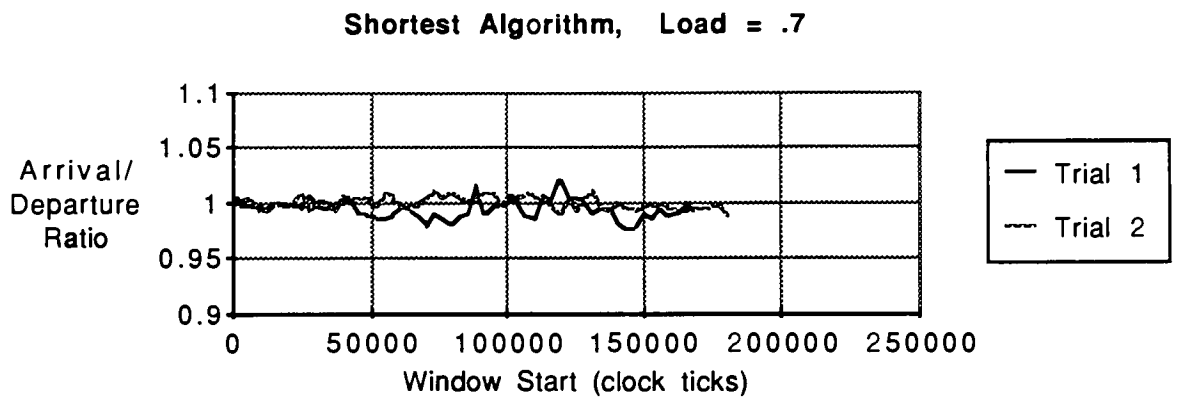


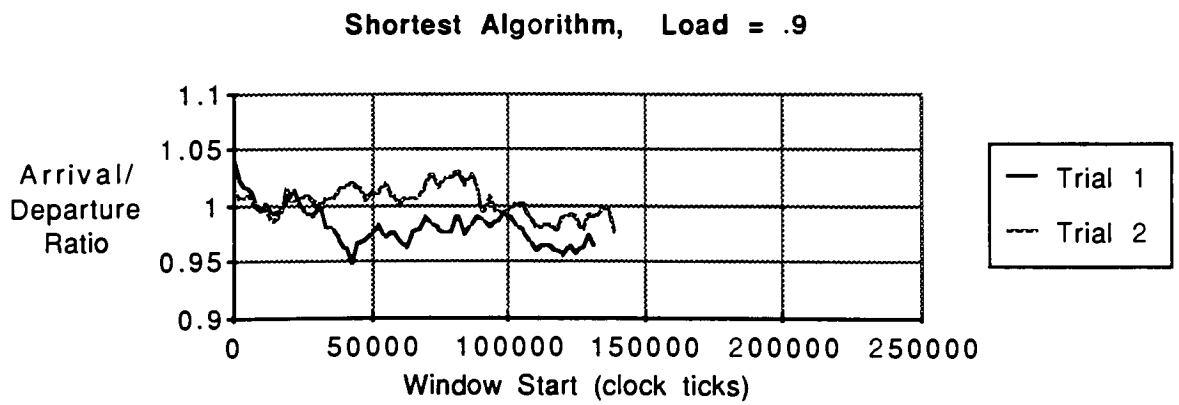

Threshold Algorithm, Load = .5
Arrival/ Departure Ratio
1.1
1.05
1
0.95
0.9
0
50000
100000
150000
200000
250000
Window Start (clock ticks)
Trial 1
Trial 2


Threshold Algorithm, Load = .7
Arrival/ Departure Ratio
1.1
1.05
1
0.95
0.9
0
50000
100000
150000
200000
250000
Window Start (clock ticks)
Trial 1
Trial 2


Threshold Algorithm, Load = .9
Arrival/ Departure Ratio
1.1
1.05
1
0.95
0.9
0
50000
100000
150000
200000
250000
Window Start (clock ticks)
Trial 1
Trial 2


Shortest Algorithm, Load = .5
Arrival/ Departure Ratio
1.1
1.05
1
0.95
0.9
0
50000
100000
150000
200000
250000
Window Start (clock ticks)
Trial 1
T 2 lost


Shortest Algorithm, Load = .7
Arrival/ Departure Ratio
1.1
1.05
1
0.95
0.9
0
50000
100000
150000
200000
250000
Window Start (clock ticks)
Trial 1
Trial 2


Shortest Algorithm, Load = .9
Arrival/ Departure Ratio
1.1
1.05
1
0.95
0.9
0
50000
100000
150000
200000
250000
Window Start (clock ticks)
Trial 1
Trial 2