THE EFFECTS OF LOAD BALANCING ON RESPONSE TIME FOR CSMA/CD NETWORKS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Katherine M. Baumgartner

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

August 1984

ACKNOWLEDGMENTS

I would like to give special thanks to my advisor, Professor Benjamin Wah, for his patience and guidance during the research of this thesis. I would also like to acknowledge the support of this research by the National Science Foundation under grant ECS-81-05968.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Katherine M. Baumgartner. MSEE. Purdue University. August 1984. The Effects of Load Balancing on Response Time for CSMA/CD Networks. Major Professor: Benjamin W. Wah.

Local Area Networks have been developed to aid communication between computer systems. CSMA/CD networks are a commonly used Local Area Network with a bus topology. Local Area Networks can also be used to support load balancing, which is the remote execution of jobs in a user transparent manner for improved system performance. Previous studies have shown that systems which use load balancing have improved performance over systems that do not. Load balancing on CSMA/CD networks will have its greatest effect when jobs are sent from the maximally loaded processor to the minimally loaded processor. The objective of this study is to determine the effects of load balancing on response time for different traffic intensities. This information can be used to more efficiently locate the maximally and minimally loaded processors, and to determine which traffic intensities are appropriate for load balancing. The results of the analysis and simulation show that the benefits of load balancing are maximum at low traffic intensities and load balancing intervals that are short compared to job service time. Load balancing at high traffic intensities is shown to be detrimental to system performance.

CHAPTER I
INTRODUCTION

1.1. Evolution of Computer Systems

Early computer systems had a centralized architecture which was characterized by control being concentrated at one location. System design was constrained by hardware expense: it was cost prohibitive to duplicate resources. In addition, other costs such as staffing were minimized by maintaining the computer in a central location. Furthermore, the centralized architecture supported efficient resource sharing and applications than required a high degree of interaction among users.

As computer technology progressed, hardware costs dropped dramatically and hardware capabilities increased. As users needs became diverse and specialized, it became cost effective to have more than one system at a single location. Computer systems became more distributed and resources were duplicated. These distributed systems consisted of autonomous systems dispersed about a site: control was not limited to one location. Distributed systems, however, lacked the ease of communication allowed by centralized systems. Since many applications are interaction intensive, this caused a significant problem.

Long distance communication between centralized systems was typically handled using telecommunications equipment. This type of communication was infrequent enough that low data transmission rates of 10^4 bits per second, and error rates of one bit per 10^5 bits were tolerable [Tan81]. This performance was not good enough for communication between systems at the same site. Local systems required frequent interaction; therefore data transmission and error rates should be comparable to those for memory access, that is, on the order of 10^6 bits per second with high reliability.

Alternatively, systems can be connected with cables in a point to point fashion. These connections allow data rates of 10^7 bits per second with error rates of one bit per 10^{12} bits [Sta84]. Clearly the performance of these connections is far superior to that of telecommunications. Maintenance and I/O hardware costs for point to point communications however, become prohibitive as the number of processors increases. Cost is proportional to the square of the number of devices: for n processors, $n(n-1)$ connections are required.

The communications needs of locally distributed systems are specific. Communications facilities need to handle varying traffic loads as they will handle communications between a variety of devices. Additional requirements are easy installation, reconfigurability, and low cost. Three classes of local networks have been developed in response to these requirements: the high speed local network (HSLN), the computerized branch exchange (CBX), and the local area network (LAN).

The HSLN is a specialized network that offers high data rates (about 50Mbps), with application for connecting mainframes and other high speed devices [Sta84]. HSLNs have a physical range limited to one kilometer, a bus topology, and can support tens of devices. They are packet switched meaning that information is sent in fixed size units or packets and physical connections between the source and destination are not established. HSLNs are not general purpose networks and they are considerably more expensive than CBXs or LANs.

CBXs are the least expensive type of local network [Sta84]. They are circuit switched networks meaning that a connection from source to destination is established for each transmission. CBXs have a star topology (Figure 1.1), and like the HSLN they have a physical range of about one kilometer. Data transmission rates are 9.6 - 64kbps. They can support hundreds to thousands of devices.

LANs are the midrange-cost local network [Sta84]. They can have bus, tree or ring topologies (Figures 1.2,1.3,1.4), with the greatest physical range of the local networks at 25 kilometers. The tree topology is a generalization of the bus topology. LANs are packet switched with transmission rates of 1 to 20Mbps. Like the CBX, they can support hundreds to thousands of devices. LANs are the most general purpose of the local networks. They can support a larger variety of devices and data rates than either the CBX or HSLN because they can support a variety of traffic loads.

As mentioned above, LANs can have either a bus, tree or ring topology. The operation of the LAN differs for the ring as opposed to the bus or tree topologies. A LAN connected with a ring topology physically consists of repeaters connected by unidirectional data paths (Figure 1.5). The repeaters are active devices that insert, remove, and receive packets. A station will transmit a packet to the network, and the packet will propagate around the network and be repeated at every station. At the packet's destination, it can

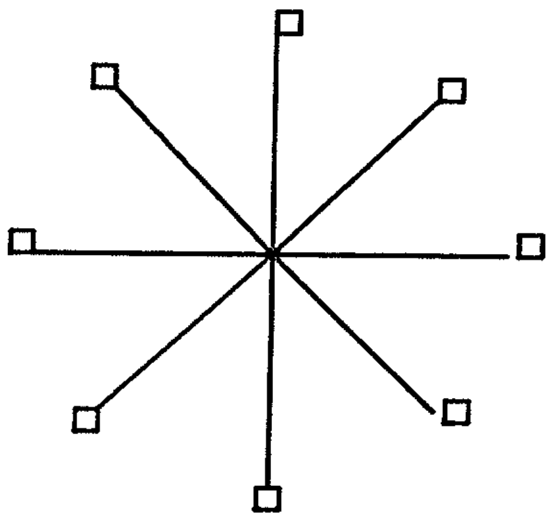

Figure 1.1 Star Network Topology

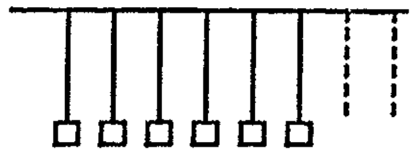

Figure 1.2 Bus Network Topology

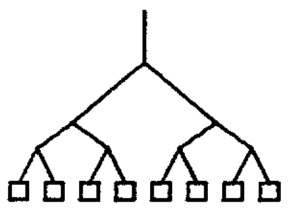

Figure 1.3 Tree Network Topology

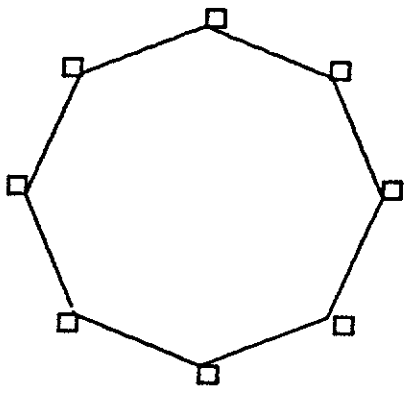

Figure 1.4 Ring Network Topology

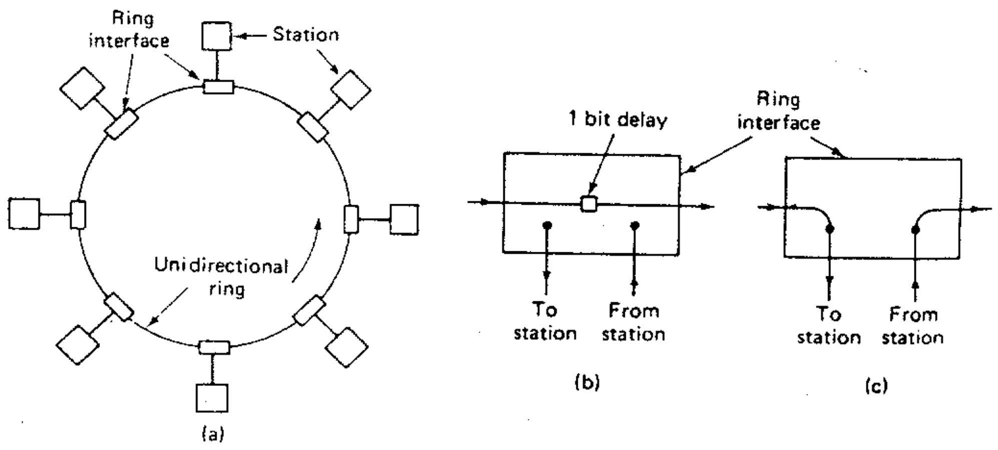

Figure 1.5 Ring Network [Tan81]

either be removed from the network, or allowed to continue the full revolution to its source to serve as an acknowledgement. The packet would be removed at that point.

Disadvantages of the ring LAN are first that a failure in a data path or repeater disables the entire network. Second, installation of a new station temporarily disables the entire network. Repeaters need to be synchronized with one another. An additional design constraint is that a bit must be able to travel around the loop in an incremental number of clock pulses.

The access points of the bus and tree topologies are passive in contrast with the ring. A station transmits a packet to the network and all stations can listen and copy packets addressed to them. Packets can be acknowledged by reserving the network for verification messages immediately following the packet's transmission. Packets are removed from the network by being absorbed at the terminal points. The bus and tree topologies are flexible and commonly used.

LANs with a bus or tree topology are most commonly implemented with a carrier-sense-multiple-access with collision-detection (CSMA/CD) network, or a token bus network. For the token bus, the stations are arranged in a logical ring and the physical arrangement is unrelated. Control of the network is granted to the station possessing the "token", and the station can transmit or poll other stations while in control. When the station is through or its control exceeds a time limit, it passes to token to the next station in the ring. The main advantages of the token bus is its high throughput, disadvantages are complexity and overhead.

CSMA/CD networks evolved from CSMA networks. A station wanting to transmit on a CSMA network first listens to see if the network is in use, then transmits if the network is idle. CSMA/CD networks have better performance because is the station listens before and during a transmission to determine if a collision occurs, and stops transmitting if that is the case. A more detailed description of CSMA networks is included in Chapter 2. CSMA/CD networks are designed for bursty traffic and low load where the overhead of the token bus make it less efficient.

1.2. Motivation

Local networks evolved from distributed systems need for a specific kind of communications facility. The direct benefit from local networks is ease of communications between devices. An indirect benefit arises when resources are shared using the network: the network can be used for load balancing. Load balancing uses communications facilities to support remote job execution in a user transparent fashion so utilization is improved through resource sharing. When jobs arrive at a processor, a decision to load balance is made if the job is likely to be finished sooner when executed remotely than when executed locally.

Only one job can be sent across a local network at a time. If load balancing is to have a maximum effect, the job must be sent from the maximally loaded processor to the minimally loaded processor. Response time is the amount of time elapsed from job submission to job completion and is an indication of the processor load. The maximally loaded and minimally loaded processors can be located efficiently with knowledge of the response time characteristics.

The behavior of networks of processors varies as load varies. The effects of load balancing will be different as job arrival and service characteristics change. The ratio of the job arrival rate to the job service rate is called the traffic intensity. As the traffic intensity changes, the decision to load balance must be adjusted.

The global objective of load balancing is to minimize average network response time. Load balancing requires that the maximally and minimally loaded processors be located which can be done efficiently with some knowledge of response time. Additionally, network characteristics such as traffic intensity affect network behavior. Knowledge of network behavior and response time at various traffic intensities can aid in making beneficial load balancing decisions.

1.3. Objective

The main objective of this study is to determine the effect of load balancing on the response time of CSMA/CD networks. The information gained about response time can be used to locate maximally and minimally loaded processors for more efficient load balancing. The study of network behavior at different traffic intensities when load balancing is applied can be used to determine when load balancing is appropriate. These objectives will be realized by analysis and simulation.

1.4. Assumptions

Assumptions were made about the load balancing operation and the characteristics of the network and stations (processors) for the analysis and simulations associated with this study. These assumptions were made to give consistency and reality to the study and to make the solution computationally feasible.

The load balancing operation is assumed to send a job from the maximally loaded processor to the minimally loaded processor. As mentioned above, this is necessary for effective load balancing. A load balancing decision considers the transmission of a single job. Jobs can be sent for remote execution only once.

The processors connected to the CSMA/CD network are assumed to be identical. This assures that results will be the same regardless of whether the job is executed remotely or locally. The processors are considered M/M/1 [Kle75] systems with a finite queue limit.

1.5. Organization

There are four chapters following this introduction. Chapter 2 provides background information. It includes a discussion of previous load balancing studies and describes the operation of CSMA/CD networks and network protocols in detail. There is an explanation of which protocols can be used with load balancing, and which protocol is assumed for this study.

Chapter 3 describes the analytical model of the network and load balancing operations. It includes a detailed description of the network model and the equations used to describe the processor behavior. Results of the analysis are included.

Chapter 4 details the simulations. The simulation language is described, and the static and dynamic portions of the simulation model are explained. The network behavior is observed for a range of traffic intensities.

Chapter 5 contains a summary of results and draws conclusions.

CHAPTER II
BACKGROUND

This chapter provides background information about the load balancing problem and CSMA/CD networks. First, a description of CSMA/CD networks and network protocols is given. Next, the load balancing problem is discussed. Points for consideration in this study relative to previous studies and the protocol assumed for this study are noted.

2.1. Description of CSMA/CD Networks

A CSMA/CD network is a carrier sense multiple access network with collision detection. CSMA/CD networks are characterized by a distributed control scheme with no central controller. A station wishing to use the network first senses it to determine if it is in use, that is, if a carrier is present. If the network is in use, the station waits, otherwise it attempts to obtain control of the network by transmitting a message. This is called contending for use of the network.

A station cannot be sure that it will have exclusive control of the network when it finds the network idle. It is possible that other stations will sense the network simultaneously and all conclude that the network idle. It is also possible that a station will sense the network and determine that it is idle when another station has already begun transmitting. This occurs if the transmitting station's signal did not have sufficient time to propagate across the entire network. This implies that a station at an extreme end of the network will not be sure that it has exclusive control of the network for a period of twice the end to end propagation delay of the network. This would be the time for the transmitting station's signal to reach the station furthest from it, and for any information about collision to propagate back to the transmitting station. This time interval is called the collision interval, collision window, or contention slot. If the packet length is small and the collision interval is large, then it is efficient to allow the stations to transmit when they find the network idle. If the collision window is small and the packet time is long, then it is not efficient to allow a station to transmit when it finds the network idle because the network is essentially disabled for the time required to send a packet if a collision occurs.

Contention protocols are designed to limit wasted time caused by collisions and situations where the network is idle but a station is ready to transmit. There are three types of protocols for contention resolution on CSMA/CD networks. The simplest type is the contention protocol. Examples of this type of protocol are ALOHA, slotted ALOHA, persistent CSMA, nonpersistent CSMA, p-persistent CSMA, and binary exponential backoff.

The ALOHA protocol is the least complex of the multiple access protocols [Abr70]. A station with a packet in its transmission buffer will transmit immediately without listening to determine if the network is in use. Collisions are detected by the lack of an acknowledgement within a prescribed amount of time and retransmission occurs a random amount of time after a collision has been detected. The maximum throughput for this protocol is $1 / 2e = 0.18$ [Tan81]. An improvement to ALOHA is the slotted ALOHA protocol. This protocol uses time slots to synchronize transmissions which results in improved performance (throughput of 0.36) [Tan81] because colliding packets will overlap completely.

The other contention protocols mentioned use time slots also. With the binary exponential backoff, persistent CSMA, and non-persistent CSMA protocols, a station wanting to transmit will assume control of the network if it senses that is idle. The p-persistent CSMA protocol causes the station wanting to transmit to assume control of the network with probability p and defer to the next time slot with probability $1-p$. If the network is idle during the next time slot, the station will once again transmit with probability p and defer with probability $1-p$. For all these protocols the station will stop transmitting if a collision is detected and repeat the procedure described. The same is true if the station senses that the network is in use initially. The persistent CSMA senses the network continually and begins transmitting immediately upon determining that the network is idle. The p-persistent and nonpersistent cause the stations to wait a random interval before transmission, but for the binary exponential backoff protocol, the number of slots deferred depends on the number of collisions experienced for the current packet. If there have been k collisions, then the station will defer sensing the carrier for 2^k slots.

The second type of contention resolution protocol is the collision free protocol. An extreme example of this type of protocol is time division multi-access (TDMA) [Mar70] where each station is given the opportunity to use the network in a sequential fashion. Other examples of this type of protocol are the bit map protocol, the broadcast recognition with alternating priorities protocol[Sch76], multi-level multi-access protocol[RoW77], and the binary countdown protocol [Tan81]. Collision free protocols require that each station have an address. The contention window then consists of some number of slots where stations wishing to transmit write their addresses to the network in some formatted manner. Before transmission begins an order of transmitting stations is determined. The significant characteristics of this type of protocol are that each station must have an address associated with it and that there is some amount of network overhead associated with every transmission even if a station is uncontested for use of the network.

Contention protocols perform well under conditions of low network load because there is little contention associated with each transmission. As network load increases, the time for contention resolution increases and network efficiency decreases. Of the collision protocols described, only the binary exponential backoff protocol adapts to the current network load. Nonadaptive protocols have been shown to be unstable because positive feedback after collisions can potentially cause endless collisions [Fay77]. The binary exponential backoff protocol reacts to the current network load, but must rediscover load information after every successful transmission.

Collision free protocols have greater efficiency than contention protocols at high network load, but larger delays when network load is light. An additional disadvantage of collision free protocols is that they may need to be adjusted when new stations are added to the network if the new address are not accommodated. A desirable alternative is a hybrid and such protocols are described below.

The third type of protocol for contention resolution is the limited contention protocol which adjusts its performance to the network load. This type of protocol has contention-protocol performance at low network load and collision-free-protocol-performance at high network load. Access to the network is granted based on the outcome of a contention period for this type of protocol. Stations ready to transmit broadcast a number called a contention parameter. The significance of the parameter is dependent on the specific protocol. As collisions occur during contention periods, the number of stations allowed to contend in the next period is decreased until only one station broadcasts. That station is granted control of the network to transmit its message. Contention parameters can be based on different information for example, the distance of a station from an origin, the time at which a packet became ready for transmission, or a random variable. Examples of limited-contention-protocols follow.

The Adaptive Tree Walk protocol [Cap77] and the Urn protocol [Gal78] have contention parameters based on a station logical distance from an origin. The Adaptive Tree Walk protocol is a tree traversal algorithm in which the stations are organized as the leaves of a binary tree. All stations are allowed to contend initially. If collisions occur, the set of stations is decreased to subtrees until one station remains. The contending parameter is the distance from the origin or root of the tree. The objective is to find the station ready to transmit with the minimum distance from the origin. The number of stations eligible to transmit is essentially halved after each contention period. This approach has been shown sub optimal [Hlu81].

The Urn protocol is an enhancement of the Adaptive Tree Walk Protocol. For this protocol, the stations can be thought of as being in an urn. A sample of stations is drawn from the urn and there is a collision if more than one processor in the sample is ready to transmit. The size of the sample drawn from the urn is found by estimating the number of ready stations and determining the optimal sample size to yield one ready station. The actual stations are chosen in a round robin fashion. The stations are arranged logically in a circle and the set of contending stations rotates around the circle. Once again, the contending parameter is the distance from an origin. This protocol adapts to heavy network load by adjusting the sample size.

The Arrival Time Window protocol [ToV82] bases contention parameters on the time that packets become ready for transmission. All stations consider a common interval. Stations that have packets that entered their transmission buffers during that interval are allowed to broadcast a contention parameter. As collisions occur during the contention period, the interval is decreased. Gallagher also proposed a protocol based on this criteria.

The Virtual Window protocol [JuW84] uses random numbers as contention parameters. The station broadcasting the maximum number will gain access to the network. The number is found by using a window search technique where the upper and lower bounds of the window are adjusted by considering the distribution of the random numbers. The Virtual Window protocol can be generalized for load balancing, resource sharing, query processing, and priority resolution by using different contention parameters. Such an extension is described in the next section.

2.2. Load Balancing Problem, Previous Studies

Load balancing was defined in Chapter 1 as the use on communications facilities to support remote job execution in a user transparent fashion so utilization is improved through resource sharing. The decision to perform a load balancing operation (sending a job for remote execution) can be based on many different criteria, for example the current state of an individual processor and its past experience with job arrival rate and service requirements. The load balancing would be based on local information in that case. Alternatively, the decision can be based on global information such as the state of all stations attached to the network, and the history of all stations arrivals and departures. The decision could also be based on a combination of global and local information.

Load balancing can be decided in a centralized or distributed manner. A centralized decision implies that information is collected and decisions to load balance are made at a one location. An example would be a system with a job scheduler where all jobs arrive at one location, and are then dispatched to stations for processing. A distributed decision occurs when the responsibility for decisions is not limited to one processor in the system.

The load balancing decision can also be classified as deterministic or probabilistic [ChK79]. A decision that is based on the current state of the system is deterministic. If a decision is based on physical characteristics of the network of stations such as the service rate of a station, then the decision is probabilistic. Obviously, a probabilistic decision does not adapt to the current state of the system.

Two theoretical studies associated with centralized load balancing studies are those by Chow and Kohler, and Ni and Hwang. Both of those studies consider multi-processor systems with job dispatchers or schedulers (Figure 2.1).

The study by Chow and Kohler involved both deterministic and probabilistic strategies. A probabilistic strategy was proposed for load balancing a single job class. In that strategy, jobs were dispatched by proportional number to each processor relative to that processor's service rate

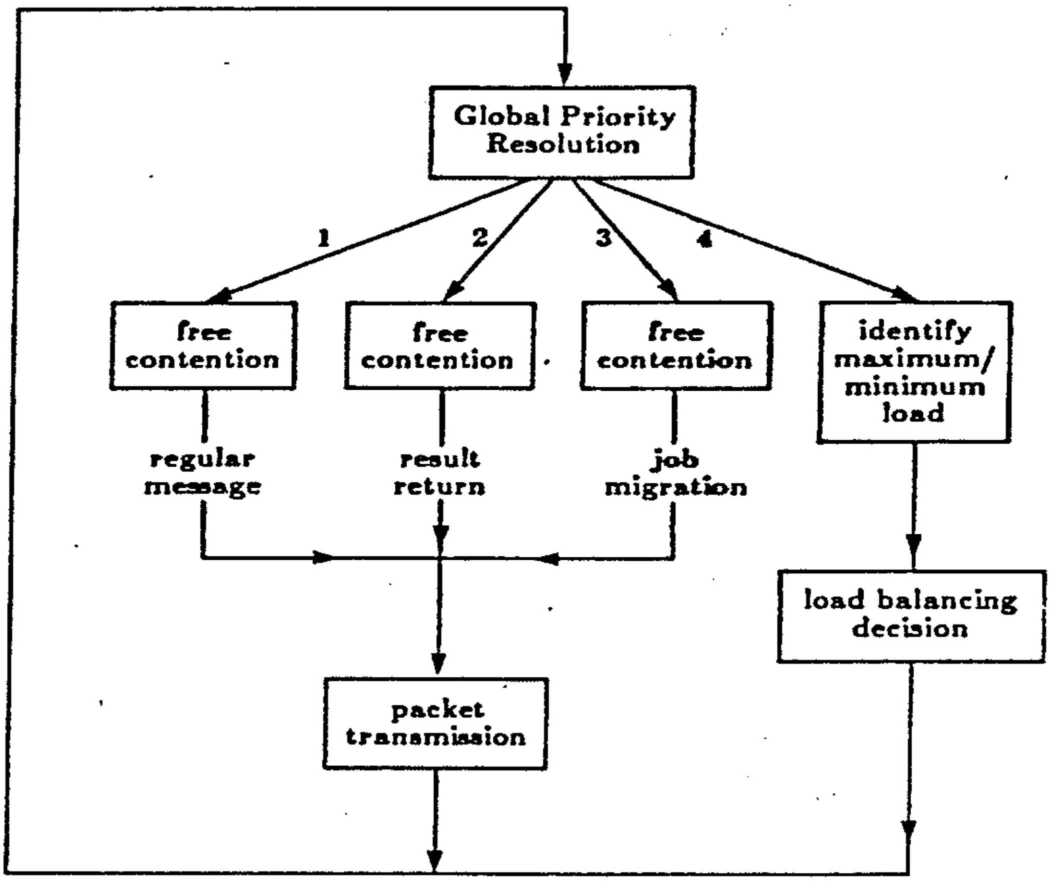

Figure 2.1 Summary of Load Balancing Protocol [WaJ83]

and the strategy is called proportional branching. A deterministic strategy was proposed for a system with multiple job classes. For that strategy, the decision where to send a job was based on either minimizing or maximizing a system parameter for the best system performance. The parameters used were response time, system time, and throughput. Maximizing throughput resulted in the best performance.

Ni and Hwang's study considered probabilistic strategies [NiH81]. The consideration was that although deterministic strategies result in better performance, the overhead associated with implementing them is higher than that associated with probabilistic strategies. The probabilistic strategy for a single job class yielded better performance than Chow's proportional branching strategy. An optimal probabilistic algorithm for multiple job classes was also proposed and was easier to implement than deterministic strategies.

Studies have been done concerning resources connected with interconnection networks. Wah and Hicks compared distributed versus centralized strategies for resource sharing on interconnection networks. If processors are considered as resources, then load balancing is actually a form of resource sharing. Crossbar interconnection networks (Figure 2.2) and interconnection networks with logarithmic delays (such as the cube [SiM81] depicted in Figure 2.3) were considered in the study. Crossbar networks have direct connections between all systems connected to the network and are more expensive than interconnection networks with logarithmic delays. Networks with logarithmic delays can form connections between all systems connected to the network but these connections are not permanent. Because of these characteristics, crossbar networks can use circuit switched connections but interconnection networks with logarithmic delays more efficiently use packet

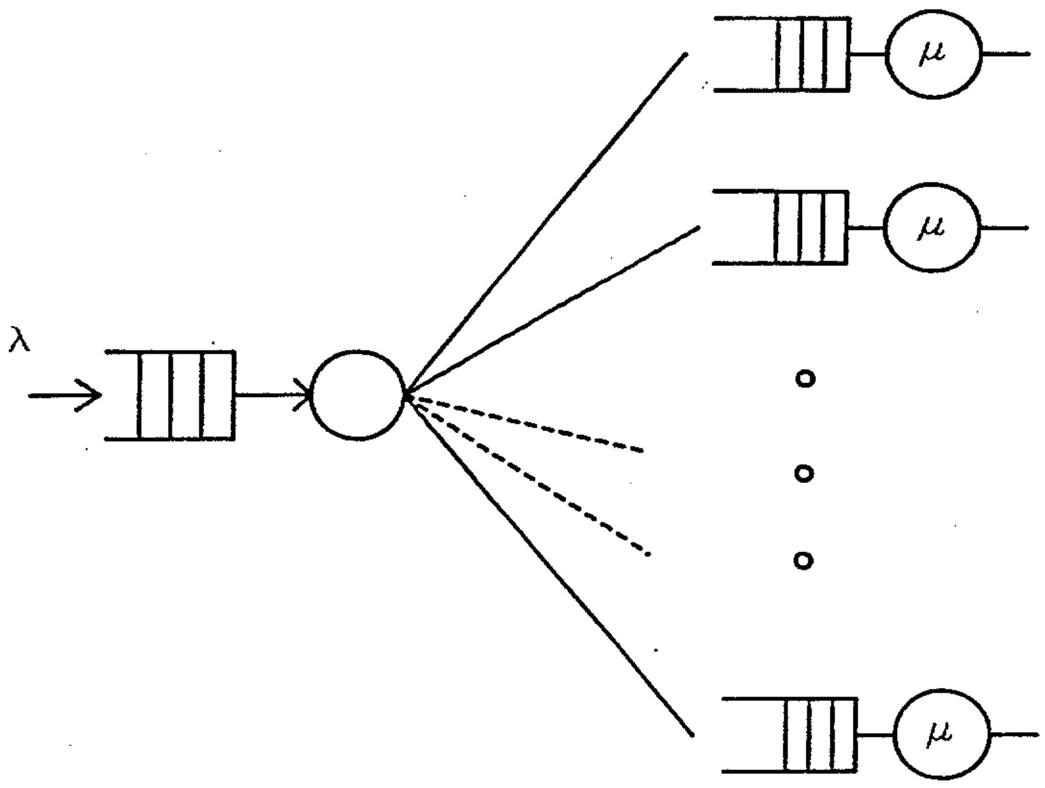

Figure 2.2 Multiprocessor System with Job Dispatcher

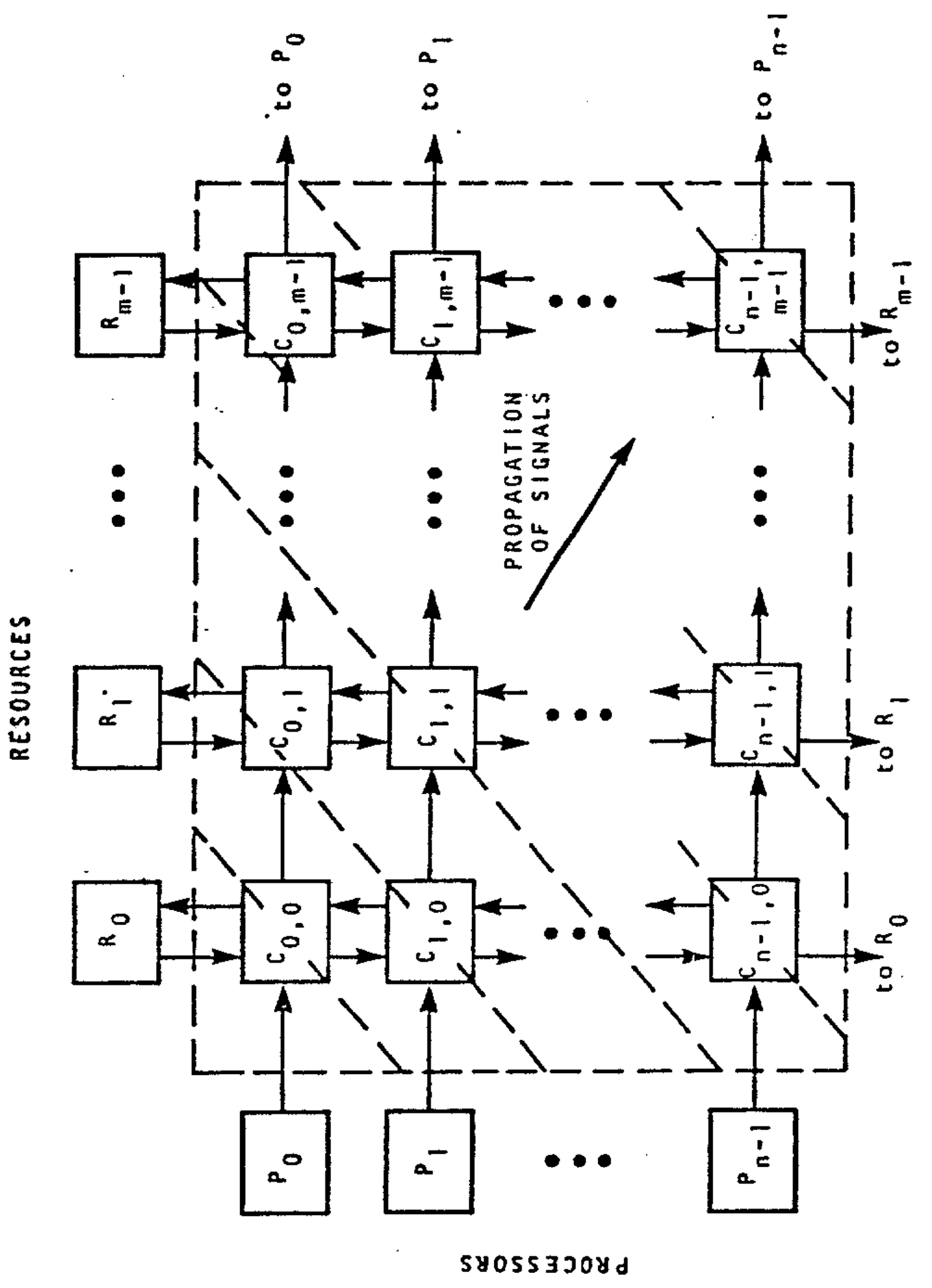

Figure 2.3 Crossbar Interconnection Network

switching. Distributed strategies for resource sharing on these networks were found to schedule jobs in a time proportional to network delays.

Load balancing is used on the Purdue Engineering Computer Network (ECN) [Hwa82]. ECN is a system of computers connected by a CSMA/CD network that uses the binary exponential backoff protocol for contention resolution. The load balancing decisions are distributed: each processor decides whether to send its jobs for remote execution. A processor will poll the other processor for information about their loads, decide which processor has the lowest load, and send the job for remote processing if there is a benefit.

The Virtual Window protocol (described above) has been extended to be specifically useful for load balancing by using a contending parameters that reflect the load at the stations to identify the maximally and minimally loaded processors [WaJ83]. This "load balancing protocol" is summarized in Figure 2.4. There are four priority levels of contention associated with this protocol:

(1) contention for regular message transfer;

(2) contention for result return;

(3) contention for job migration; and

(4) identification of maximally and minimally loaded processors.

(1) has the highest priority because regular message transfer is the primary purpose of the network. (2) is next because the completion of a current load balancing operation has priority over the initiation of new load balancing operations. For the same reason, the third highest level is (3), and the lowest is (4).

When regular messages are sent, there is one step in the protocol which is a contention period. If the highest priority packet ready for transmission is (2)

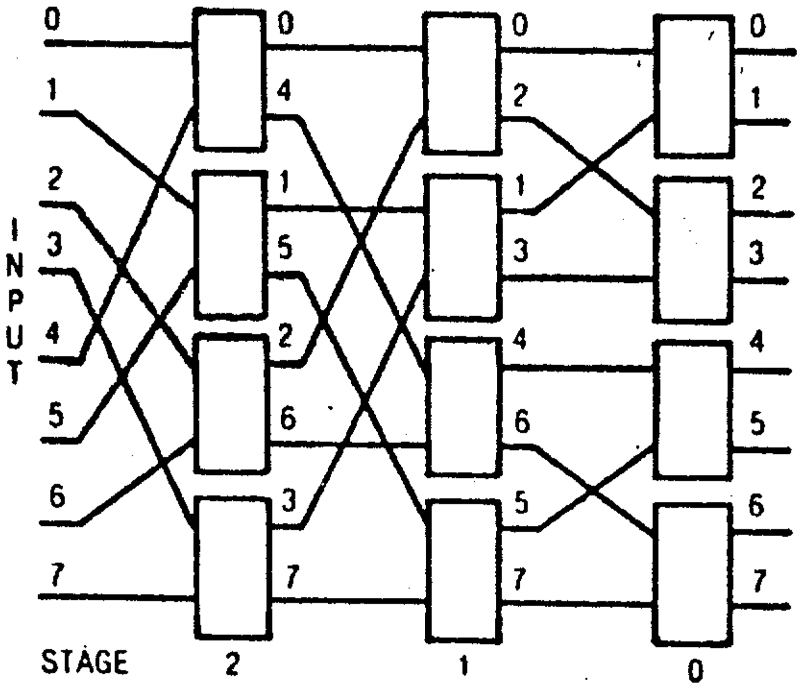

Figure 2.4 Cube Network [SiM81]

or (3), first the priority level is determined, then there is a contention period within that priority level. The identification of the maximally and minimally loaded processors is the possible initiation of a new load balancing operation. There are two contention intervals involved with identifying the maximally and minimally loaded processors and no packets are actually sent.

A common result of the previous studies on the load balancing problem is that a network with load balancing performs better than a network without load balancing. Load balancing, however, may cause the following problems.

(1) If status information is used in a deterministic decision it must be readily available, or the decision to load balance could be made for an a system status which is not current. This type of action can increase the response time.

(2) Load balancing causes network load which can impede message transmissions. The overhead due to the broadcast of status information and sending jobs should be small.

Three types of protocols for contention resolution on local computer networks have been described. Load balancing can be done with any of these types of protocols with varying degrees of efficiency. If the decision to load balance is to be made in a centralized fashion, then load average statistics must be collected at one of the n stations. This means at least $n - 1$ point-to-point communication operations will be needed. This is possible with all three types of contention-resolution protocols, but as noted previously, the limited contention protocols are efficient over a larger variety of network loads. If the load balancing decision is to be made in a distributed fashion, either the load statistics must be maintained at all the stations, or must be kept in a location that is accessible to all stations. In the case where statistics are maintained at all stations, n broadcasts are necessary regardless of the type of protocol used. If statistics are kept at a common location, the following are considerations. There is no central memory for the network of processors, and if statistics are kept at one processor, it is inefficient for other stations to access the statistics. The network itself, however, is common to all stations, and the load balancing protocol exploits this fact. For a distributed load balancing decision, the load balancing protocol yields no duplication of load average statistics and the fewest number of transmissions for the types of decisions considered.

2.3. Model for the Current Study

This study considers a CSMA/CD network with identical stations (or processors) connected to it. The system is modeled using the load balancing protocol suggested by Wah and Juang. If another protocol were used, or the decision made in a centralized manner, adjustments would need to be made in the time required for the load balancing operation. For most other protocols, this interval would be increased.

CHAPTER III
ANALYSIS OF RESPONSE TIME FOR A LOAD BALANCED NETWORK

This chapter contains a description of the model of the network of stations and an explanation of the equations used to find the average response time of the network. The states of a single station and of the network of stations are described first. The method used to find the average response time is then outlined. Results of the analysis are also included.

3.1. Description of a Network State

The system modeled is a group of independent computer systems, or stations, connected by a LAN. The assumptions made about the model are:

(1) the stations are modeled as M/M/1 systems [Kle75];

(2) the stations have a finite queue limit, l; and

(3) the time required for a load balancing operation (the load balancing interval) is constant.

The state of each station is specified by the number of jobs in its queue. The queueing structure for an M/M/1 system is shown in Figure 3.1. The queue limit imposed results in $l + 1$ possible states for a station since it can have from zero to l jobs queued. The network consists of n stations connected

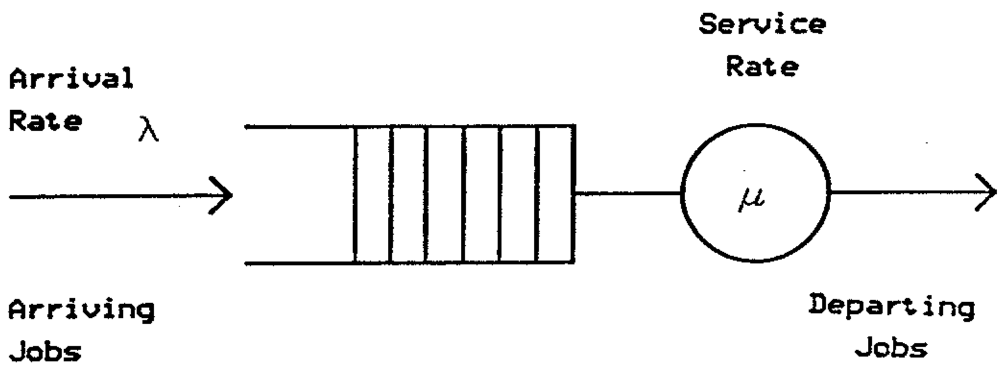

Figure 3.1 Queuing Structure of an M/M/1 System

to a CSMA/CD network with a bus topology (Figure 3.2). The network's state depends on the states of each station. The most obvious representation of the network's state would be an ordered sequence of n numbers where number i corresponds to the number of queued jobs at station i. If there were n stations, and the queue limit were l, there would be n^{l+1} possible states for the network. The number of states can be reduced considerably by considering the state of the network as the number of stations at each queue length. This is shown graphically in Figure 3.3 for a network with a queue limit of seven. Each x corresponds to a station. The numbers at the base of the columns are the queue lengths. The state of all the stations can be represented symbolically as $S = (q_0 , q_1 , \cdots , q_l)$, where q_i is the number of stations with i jobs queued. The state of the network shown in Figure 3.3 is (1 2 1 2 3 1 2 4). The number of network states, S_n, is the number of orderings of $l + 1$ numbers whose sum is n.

$$S_n = \left(\frac{(n + l)!}{n!} \right)$$

This results in fewer states than the previous representation. For a network with $n = 8$ and $q = 3$ there would be 4096 states using the first state representation, but only 165 using the second state representation.

3.2. Method

The method used to find the average response time has four steps. The first step is to find the $(l + 1)$-by-$(l + 1)$ state transition matrix for a single station. The state transition matrix is found for a interval which corresponds to the time for a complete load balancing operation. A load balancing operation requires the job be sent and returned, so the amount of time is

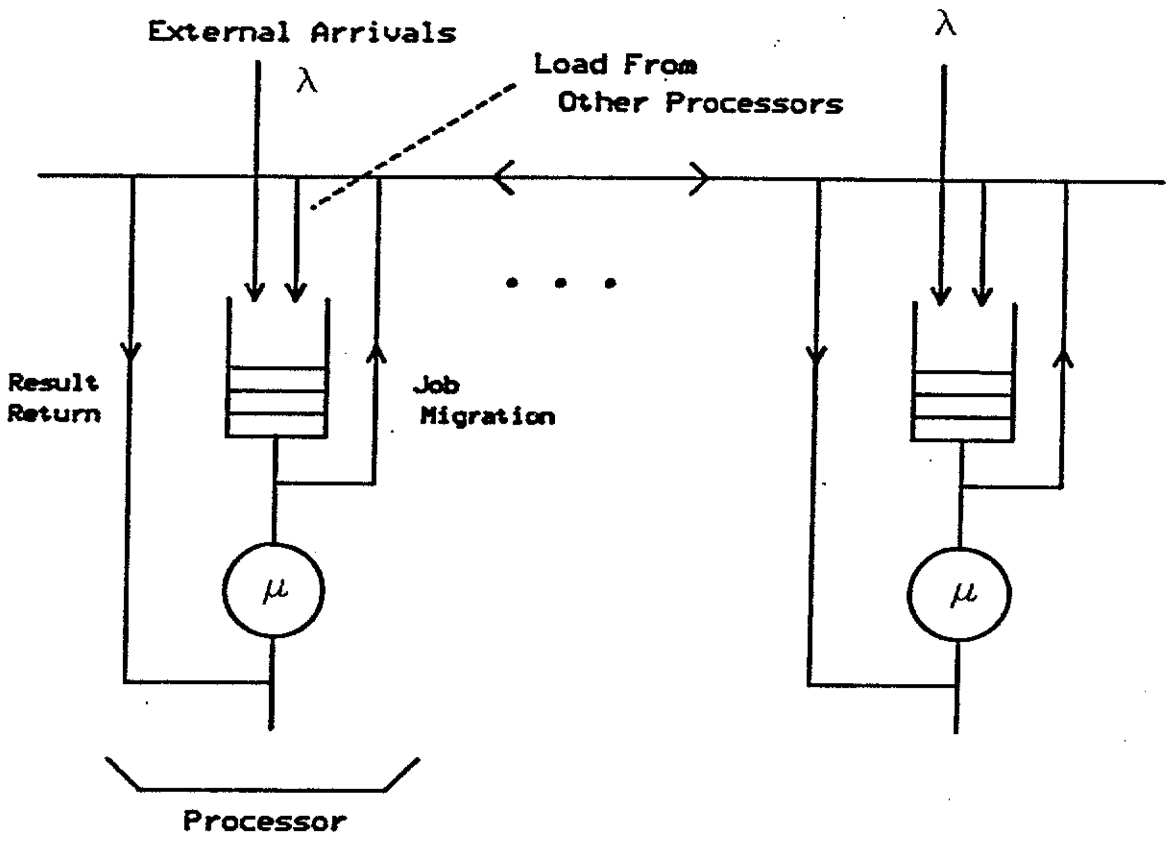

Figure 3.2 Queuing Model of CSMA/CD Network

```
                              x
                x             x
    x       x   x         x   x
x   x   x   x   x   x     x   x
0   1   2   3   4   5     6   7
```

Figure 3.3 Model of a Network State

proportional to twice that required to send a job across the network. Standard methods for determining steady state probabilities for an M/M/1 systems cannot be applied for this study because of this interval. The second step is to use the transition probabilities from the single station case to find the state transition matrix for all the stations. The method is described in Section 3.4. Next, the steady state probability vector, π, for the network of stations can be found using the state transition matrix, p, to solve $\pi = \pi\, p$. The average network response time is obtained using the steady state probabilities to compute a weighted average.

3.3. State Transition Matrix for a Single Processor

Each station is modeled as an M/M/1 system. M/M/1 systems are characterized by Markovian (or Poisson) arrival processes and service time distributions and one processor serving the queue. The probability of n arrivals over an interval of duration t for a Poisson arrival process with an average arrival rate λ is

$$P_n(t) = \frac{(\lambda t)^n e^{-\lambda t}}{n!} \,. \tag{3.1}$$

Likewise, the probability of n departures over an interval of duration t for a Poisson service time distribution with an average service rate μ is

$$P_n(t) = \frac{(\mu t)^n e^{-\mu t}}{n!} \,. \tag{3.2}$$

Considering a load balancing interval t, the probability of a transition from state i to state j is found by examining the discrete transitions. A transition can occur in a variable number of steps, so the probability of the transition is expressed as

$$Pr(i \rightarrow j) = \sum_k Pr(i \rightarrow j \mid k\ steps) \times Pr(k\ steps). \tag{3.3}$$

Consider the first term in the summation of Equation 3.3. If a station is in state n, it will go to state $n - 1$ if there is a departure, and to state $n + 1$ if there is an arrival. The state transition is a Markov chain with the following conditions: if $N(\ t\ )$ is the current state, the probability of the state increasing is

$$Pr\left\{N(t + \tau) = n + 1 \mid N(t) = n\right\} = \frac{\lambda}{\lambda + \mu}, \tag{3.4}$$

and the probability of the state decreasing is

$$Pr\left\{N(t + \tau) = n - 1 \mid N(t) = n\right\} = \frac{\mu}{\lambda + \mu}. \tag{3.5}$$

The matrix of transition probabilities for a queue length of three is

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{\mu}{\lambda + \mu} & 0 & \frac{\lambda}{\lambda + \mu} & 0 \\ 0 & \frac{\mu}{\lambda + \mu} & 0 & \frac{\lambda}{\lambda + \mu} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The row of an entry is the current state, the column is the next state. Element (i,j) is the probability of a transition from state i to state j in one step.

As the matrix is raised to successive powers, element (i,j) of the matrix raised to the k^{th} power is $Pr(i \rightarrow j \mid k\ steps)$. This chain is not regular meaning that steady state probabilities cannot be reached as the matrix is raised to higher powers because the zero entries will oscillate.

The second term in the summation of Equation 3.3 is $Pr(k\ steps)$. The probability of $(k\ steps)$ is found by considering the source and destination states as follows:

$$Pr(k\ steps) = \sum_{i \to j} Pr(k\ steps \mid i \to j) \times Pr(i \to j) \tag{3.6}$$

The first term of the summation of Equation 3.6 is the product of the probability of the number of arrivals and departures for a transition from state i to state j in k steps. For a transition in k steps, there would be a arrivals, and d departures where

$$a = \frac{k + j - i}{2} \tag{3.7a}$$

and

$$d = \frac{k - j + i}{2}\ . \tag{3.7b}$$

If either a or d is noninteger, then $Pr(k\ steps \mid i \to j) = 0$. The ordering of the arrivals and departures is significant. The number of orderings of arrivals and departures is r, where

$$r = \frac{k!}{a!d!}\ .$$

Using Equations 3.1, 3.2, 3.7a, and 3.7b, it follows that

$$Pr(k\ steps \mid i \to j) = \frac{(\lambda t)^a e^{-\lambda t}}{a!} \times \frac{(\mu t)^d e^{-\mu t}}{d!} \times r \tag{3.8}$$

The $Pr(k\ steps \mid i \to j)$ values were normalized because in this study arrivals when queues were full were ignored.

Substituting Equation (3.6) into (3.3) yields

$$Pr(i \to j) = \sum_k Pr(i \to j) \times \sum_{i \to j} Pr(k\ steps \mid i \to j) \times Pr(i \to j) \tag{3.9}$$

The value desired is expressed in terms of itself and is solved for by successive approximation. Element (i,j) of the $(l+1)$-by-$(l+1)$ state transition matrix is determined by solving Equation 3.9 using that i and j.

3.4. State Transition Matrix for the Network

The state of the network, as noted earlier, is represented by a set of $l+1$ numbers, $S = (q_0, q_1, \cdots, q_l)$, where q_i is the number of stations with i jobs queued. The probability of a network transition is the sum of the probabilities of each permutation of individual station's transitions resulting in the network transition.

For the case of a transition without load balancing, the probability is can by considering the possible transitions of a single station, and the corresponding transitions of the rest of the stations. The network's source state is specified as $S_s = (q_{s0}, q_{s1}, \ldots, q_{sl})$ and the destination as $S_d = (q_{d0}, q_{d1}, \ldots, q_{dl})$. The single station considered will be one of those with the lowest number of jobs queued of all the stations and its transition is specified as $h(i \to j)$. If the lowest number of jobs queued is b, the probability can be generally stated as

$$Pr((q_{s0}, q_{s1}, \ldots, q_{sl}) \to (q_{d0}, q_{d1}, \ldots, q_{dl}))$$

$$= Pr(h(b \to 0)) \times Pr((q_{s0}, q_{s1}, \ldots, q_{sb} - 1, \ldots, q_{sl}) \to (q_{d0} - 1, q_{d1}, \ldots, q_{dl})) +$$

$$Pr(h(b \to l)) \times Pr((q_{s0}, q_{s1}, \ldots, q_{sb} - 1, \ldots, q_{sl}) \to (q_{d0}, q_{d1} - 1, \ldots, q_{dl})) +$$

$$\ldots +$$

$$Pr(h(b \to l)) \times Pr((q_{s0}, q_{s1}, \ldots, q_{sb} - 1, \ldots, q_{sl}) \to (q_{d0}, q_{d1}, \ldots, q_{dl} - 1))$$

or

$$Pr((q_{s0},q_{s1},\dots,q_{sl}) \rightarrow (q_{d0},q_{d1},\dots,q_{dl})) \tag{3.10}$$
$$= \sum_{i=0}^{l} Pr(h(b \rightarrow i)) \times Pr((q_{s0},q_{s1}, \dots, q_{sb}-1,\dots,q_{sl}) \rightarrow (q_{d0},q_{d1}, \dots, q_{di}-1,\dots,q_{dl})) .$$

This is a recursive formula and at each step, the number of stations decreased until a boundary condition of a single stations transition is reached. The probability of a single station's transition was computed in section 3.3.

As an example, Figure 3.4 shows a state transition pictorially for a network with $l = 3$ and $n = 8$, $S_s = (1\ 2\ 3\ 2)$ and $S_d = (2\ 3\ 1\ 2)$. The lowest queue length, $b = 0$. The circled stations emphasize the transitions of the station considered singly. For this case specifically Equation 3.10 expands to

$$\begin{aligned} Pr(1\ 2\ 3\ 2 \rightarrow 2\ 3\ 1\ 2) = {} & Pr(h(0 \rightarrow 0)) \times Pr(0\ 2\ 3\ 2 \rightarrow 1\ 3\ 1\ 2) + \\ & Pr(h(0 \rightarrow 1)) \times Pr(0\ 2\ 3\ 2 \rightarrow 2\ 2\ 1\ 2) + \\ & Pr(h(0 \rightarrow 2)) \times Pr(0\ 2\ 3\ 2 \rightarrow 2\ 3\ 0\ 2) + \\ & Pr(h(0 \rightarrow 3)) \times Pr(0\ 2\ 3\ 2 \rightarrow 2\ 3\ 1\ 1) . \end{aligned}$$

The load balanced network transitions are computed in two parts, the transition of the maximally and minimally loaded stations (or group A), and the transition of the rest of the stations (or group B). Given a network state, the initial state of both groups is uniquely determined. Figure 3.5 shows a network with $n = 8$ and $l = 3$, and a transition where $S_s = (1\ 3\ 2\ 2)$ and $S_d = (3\ 2\ 1\ 2)$ The initial states of the group A and group B processors are shown in Figure 3.5a. For group A the minimally loaded station's source state is the same, but the maximally loaded station's source state is decreased by one. The group B stations are unchanged.

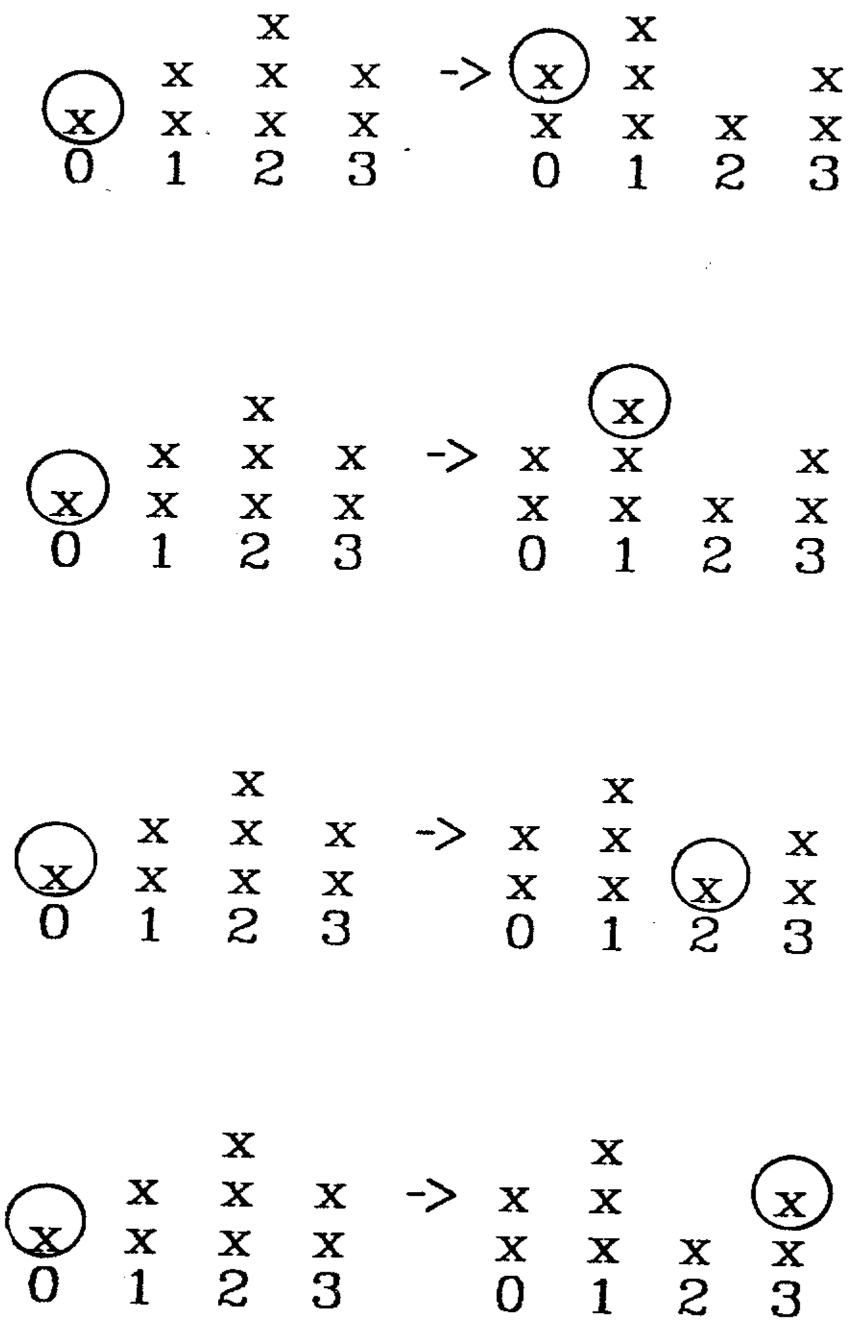

Figure 3.4 Model of a Network State Transition - Unbalanced

The group A stations have constraints on allowable transitions. As a job is load balanced, a location must be reserved for it at it's destination, the minimally loaded station's queue, and the job occupies a spot at it's source, the maximally loaded station's queue until the load balancing operation is complete. Consequently, the minimally loaded station can have transitions to state 0 through $l - 1$, and the job load balanced will be added to its queue at the end of the transition. The maximally loaded station can have transitions to states 1 through l and a job is removed at the end of the load balancing operation. In effect, the maximally and minimally loaded stations have a queue limit one less than the other station with adjustments made at the completion of the load balancing operation.

As an example, if $l = 3$ and maximally loaded processor has three jobs queued (as in Figure 3.5a), for the duration of the load balancing operation it has an effective queue length of two. For the same situation, if the minimally loaded processor has no jobs queued, then it must reserve a location for the job being sent, and for the duration of the load balancing operation it has an effective queue length of two. When the load balancing operation is completed, the job is removed from the maximally loaded processor's queue and it will have a no more than two jobs queued. The job is added to the minimally loaded processor's queue and it will have no less than one job queued.

Group A is a network with two stations, so its the destination states are limited in number and can be enumerated. The transitions of group A are combined with the corresponding transitions of group B. The probability of the entire network transition is the sum of the products of the group A and corresponding group B transition probabilities. This results in

$$Pr(S_1 \to S_2) = \sum_{\substack{max\ to\ min \\ transition}} \left[Pr\left\{ \begin{matrix} max\ to\ min \\ transition \end{matrix} \right\} \times Pr\left\{ \begin{matrix} other \\ transitions \end{matrix} \right\} \right]. \qquad (3.11)$$

As an example, consider again a network with $l = 3$ and $n = 8$. A transition is pictorially shown in Figure 3.5a. The source state is (1 3 2 2), and the destination state is (3 2 1 2). The source states of the two groups are shown in Figure 3.5b. The source state for group A is (1 0 1 0) and the source state for group B is (0 3 2 1). The sum over the destination states (Equation (3.11)) is expanded below.

$$\begin{aligned} Pr(S_1 \to S_2) = {} & Pr(1\ 0\ 1\ 0 \to 1\ 1\ 0\ 0) \times Pr(0\ 3\ 2\ 1 \to 2\ 1\ 1\ 2) + \\ & Pr(1\ 0\ 1\ 0 \to 0\ 2\ 0\ 0) \times Pr(0\ 3\ 2\ 1 \to 3\ 0\ 1\ 2) + \\ & Pr(1\ 0\ 1\ 0 \to 0\ 1\ 1\ 0) \times Pr(0\ 3\ 2\ 1 \to 3\ 1\ 0\ 2) + \\ & Pr(1\ 0\ 1\ 0 \to 1\ 0\ 1\ 0) \times Pr(0\ 3\ 2\ 1 \to 2\ 2\ 0\ 2) + \\ & Pr(1\ 0\ 1\ 0 \to 1\ 0\ 0\ 1) \times Pr(0\ 3\ 2\ 1 \to 2\ 2\ 1\ 1) + \\ & Pr(1\ 0\ 1\ 0 \to 0\ 1\ 0\ 1) \times Pr(0\ 3\ 2\ 1 \to 3\ 1\ 0\ 1) + \\ & Pr(1\ 0\ 1\ 0 \to 0\ 0\ 1\ 1) \times Pr(0\ 3\ 2\ 1 \to 3\ 2\ 0\ 1) \end{aligned}$$

If the final state of the network is S_d, and the final states of groups A and B are S_{Ad} and S_{Bd} respectively, then the number of jobs at some queue length i for the network destination state is

$$q_{di} = q_{Adi} + q_{Bdi} .$$

As each destination state for group A is enumerated, the destination state for group B that will result in S_d is determined. If any q_{Adi} is such that q_{Bdi} must be less that zero for the sum of the two to equal q_{di}, then the transition is illegal and has probability zero. As an example consider the transition

```
    x          x
    x  x  x -> x  x     x
 x  x  x  x    x  x  x  x
 0  1  2  3    0  1  2  3

                  x
                  x  x
 x     x          x  x  x
 0  1  2  3    0  1  2  3
```

Figure 3.5 Model of a Network State Transition - Balanced

$Pr(1\ 3\ 2\ 2 \rightarrow 3\ 2\ 1\ 2)$. The group A transition $(1\ 0\ 1\ 0) \rightarrow (0\ 0\ 2\ 0)$ results in the group B transition $(0\ 3\ 2\ 1) \rightarrow (3\ 2\ -1\ 2)$ which is illegal.

The steady state transition matrix can by found using a combination of this equations 3.10 and 3.11.

3.5. The Load Balancing Decision

The state transition matrix for the network is an S_n-by-S_n matrix where

$$S_n = \left[\frac{(n+l)!}{n!}\right]$$

as shown in Section 3.1. As each state transition is calculated, the decision of which transition equation to use must be made based on the load balancing decision. From a given configuration, the stations with the most and least jobs queued are located. The local execution time for a job at the end of a maximally loaded station's queue is calculated. If the sum of the load balancing interval and the execution time at the minimally loaded station is less than the execution time at the maximally loaded station, then the equation modeling load balancing, equation (3.11) is used, otherwise, equation (3.10) is used.

3.6. Average Response Time

Once the state transition matrix is found, the steady state probability vector, π is found by solving

$$\pi = \pi p \ .$$

In this study, the method used to solve $\pi = \pi p$ was to compute the difference between the identity matrix I and the state transition matrix p

[Sne75]. Referring to this matrix as R,

$$R = I - p.$$

Next, the last column of the matrix R is replaced by 1's, resulting in the matrix $\overline{R}$. Now

$$\pi\overline{R} = (0,0,...,1)$$

and

$$\pi = (0,0,...,1)\overline{R}^{-1} .$$

The average response time is a weighted average of the average response times for each of the network states. For a network state $S = (q_0,q_1, \ . \ . \ . \ , q_l)$, the average response time is

$$\sum_i \frac{\left(\left(\sum_{j=1}^{i} j\right) \times q_i \times i \times \textit{average service time}\right)}{\sum_{k=1}^{l} q_k \times k} . \tag{3.13}$$

This value is computed for all states. The average response time for the network of stations is the sum over all states of the product of the average response time for that state, and the probability of being in that state.

Two programs were written to determine the average response time of the network. The first computes the state transition matrix for a single station. It creates an input file to the second program that computes the state transition matrix for the network of stations, the steady state probability vector, and the average response time of the system. Both programs were written in the C programming language and source listings are included in the Appendix.

3.7. Analysis Results

The average response time for a network was computed for this method using load balancing intervals ranging from 0.0001 to 10.0 seconds, an arrival rate of 0.1 jobs per second, and service rates from 1.0 job per second to 0.1 jobs per second. The system modeled had 8 processors and a queue limit of 3.

Before considering the results of this analysis some performance limitations of the system will be considered. The system consists of autonomous M/M/1 subsystems. Load balancing is designed to lessen the load of a subsystem when the traffic intensity is above average. The delay for an isolated M/M/1 system with an average arrival rate λ and a service distribution with average rate μ is

$$d_{M/M/1} = \frac{1}{\mu - \lambda} .$$

The delay obtained through the analysis, d_a should be less than $d_{M/M/1}$.

The best possible system performance would occur when jobs always arrive at the optimal location. This would occur if the system operates as a single M/M/1 with arrival and service rates equal to n times those of a single subsystem. The delay of this single fast system is

$$d_f = \frac{1}{n\mu - n\lambda} = \frac{d_{M/M/1}}{n} .$$

The d_a will be larger than d_f because of the limitation that only one job can be balanced at a time. To approximate the single fast M/M/1 system, jobs would have to be evenly distributed at each step, and this is not possible when a bus is used to balance the load. Figure 3.6 shows a comparison of the delay for the M/M/1 with parameters the same as a single station and the fast M/M/1.

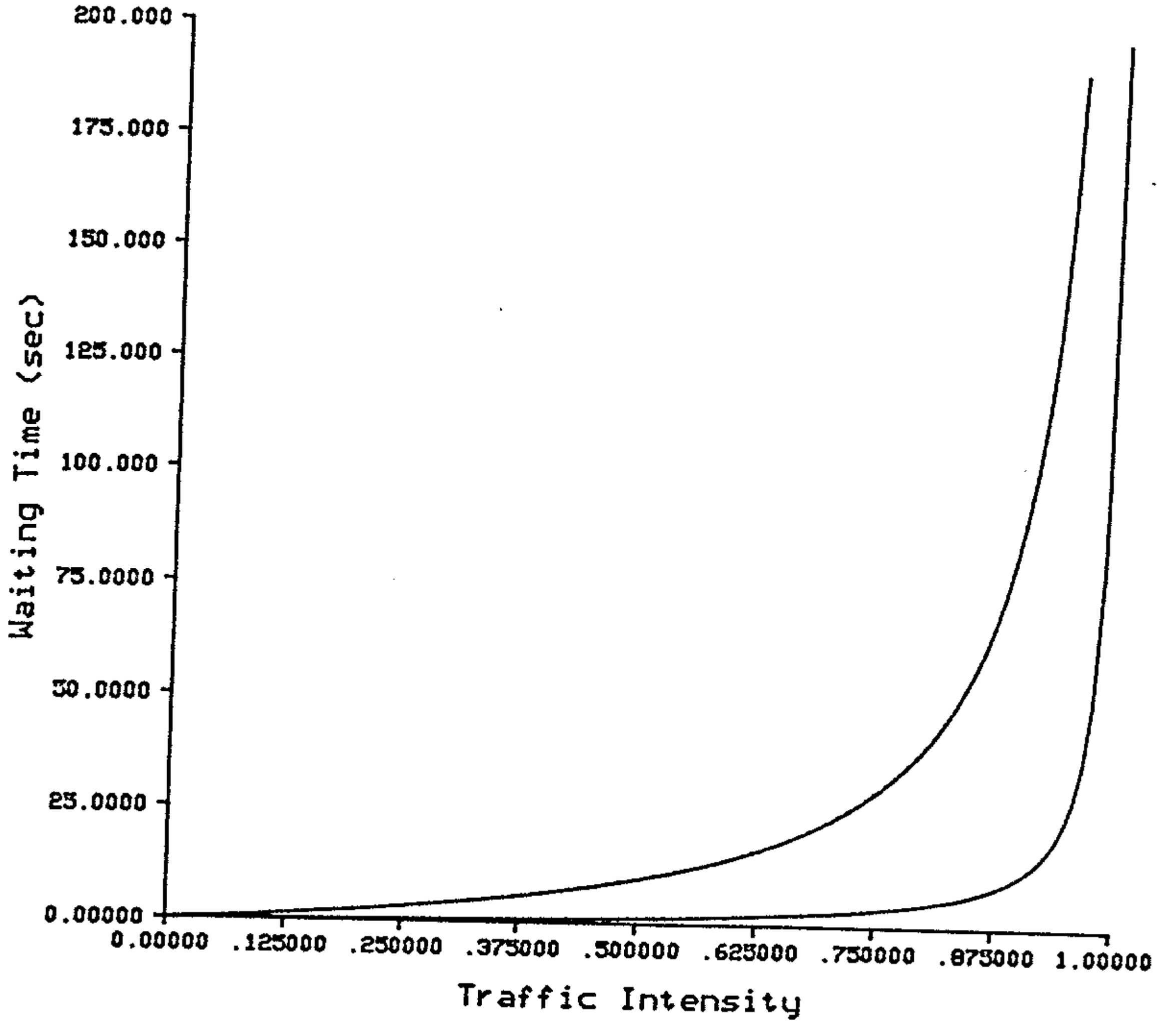

Figure 3.6 Comparison of Isolated and Fast M/M/1 Systems

As mentioned above, the system had 8 processors and a queue limit of three. That resulted in a 165 network states. Approximately 300 minutes of cpu time and 2.5 M bytes of memory were required to find the average response time for the network, so larger queue lengths are computationally unfeasible.

A station with a queue limit can be modeled as in Figure 3.7. Point A in this figure corresponds to a user producing jobs at a rate λ. Before jobs are queued at point C, they are stopped at point B and proceed only if there is room in the queue. If the queue is full, the user is notified, and stops producing jobs. Jobs held at point B check the queue at exponentially distributed intervals to see if space has become available. The effective arrival rate to the station is λ' which is equal to $Pr(queue\ not\ full) \times \lambda$. The probability that the queue is not full is determined by forming the sum of products for all states of the probability of being in a network state times the fraction of queues that are full in that state. That gives the probability that the queues are full, and the probability that the queues are not full is calculated by subtracting that quantity from one.

Table 3.1 summarizes the load balancing interval, traffic intensities, the probability of queues having available space, and the corresponding effective traffic intensity. In general, the effective traffic intensity is constant for low load balancing intervals, and decreases for intervals larger than one second.

The results of the analysis are summarized in Figure 3.8. General trends that can be seen in Figure 3.8 are that the delay slightly increases as the load balancing interval increases. The delay increase becomes most noticeable as the load balancing interval approaches the average service time of a station. The benefit from load balancing decreases at this point, and load balancing stops. No jobs were balanced at load balancing intervals greater than 10

Table 3.1 Table of Effective Traffic Intensities for Analysis

Load Balancing Interval	Traffic Intensity (ρ)	Probability Queue Not Full	Effective Traffic Intensity (ρ')
0.0001	0.1	0.9997	0.1
0.001	0.1	0.9997	0.1
0.01	0.1	0.9997	0.1
0.1	0.1	0.9997	0.1
1.0	0.1	0.9989	0.0999
10.0	0.1	0.9955	0.0995
0.0001	0.2	0.9979	0.1996
0.001	0.2	0.9979	0.1996
0.01	0.2	0.9979	0.1996
0.1	0.2	0.9979	0.1996
1.0	0.2	0.9921	0.1984
10.0	0.2	0.9840	0.1968
0.0001	0.4	0.9861	0.3944
0.001	0.4	0.9861	0.3944
0.01	0.4	0.9861	0.3944
0.1	0.4	0.9861	0.3944
1.0	0.4	0.9519	0.3807
10.0	0.4	0.9500	0.3800
0.0001	0.6	0.9602	0.5761
0.001	0.6	0.9602	0.5761
0.01	0.6	0.9602	0.5761
0.1	0.6	0.9602	0.5761
1.0	0.6	0.9099	0.5459
10.0	0.6	0.9102	0.5461
0.0001	0.8	0.9221	0.7377
0.001	0.8	0.9221	0.7377
0.01	0.8	0.9221	0.7377
0.1	0.8	0.9221	0.7377
1.0	0.8	0.8693	0.6955
10.0	0.8	0.8695	0.6956
0.0001	1.0	0.8783	0.8783
0.001	1.0	0.8783	0.8783
0.01	1.0	0.8783	0.8783
0.1	1.0	0.8783	0.8783
1.0	1.0	0.8330	0.8330
10.0	1.0	0.8307	0.8330

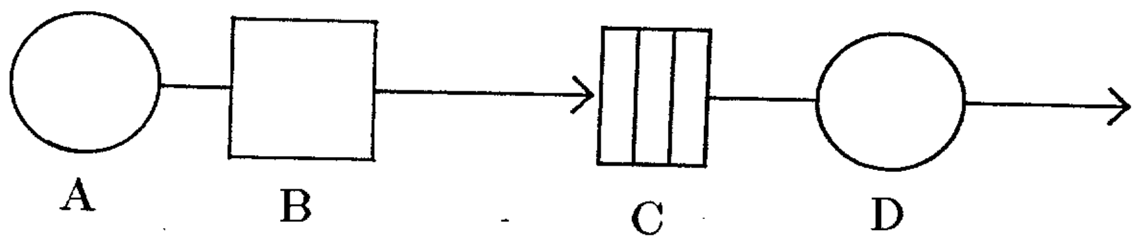

Figure 3.7 Effective Station Model

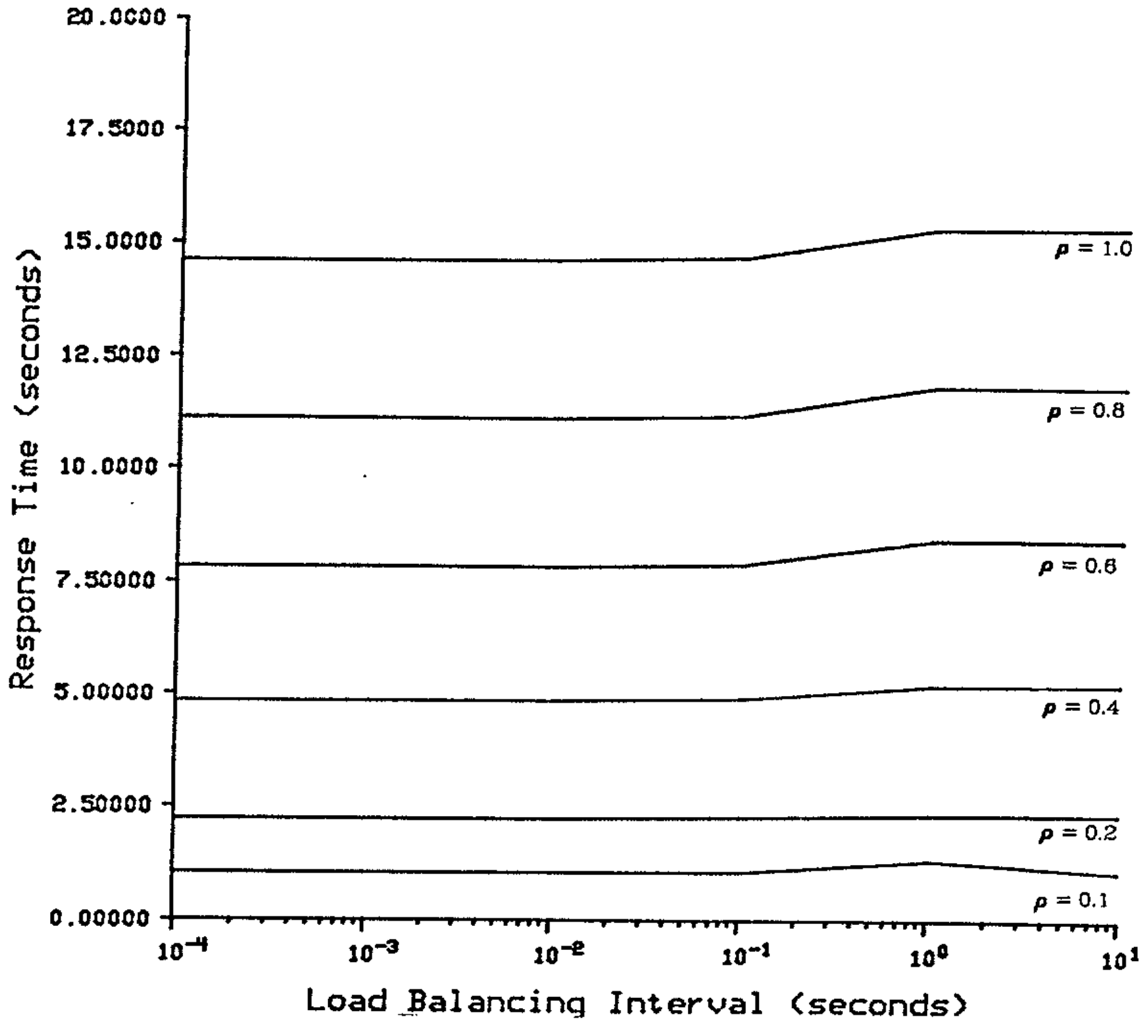

Figure 3.8 Response Time Versus Load Balance Interval for the Analysis

seconds.

Considering both Figure 3.8 and Table 3.1, the increase in delay as load balancing intervals greater than 1 second shown in Figure 3.8 corresponds to a decrease in the effective traffic intensity shown in Table 3.1. The delay at a load balancing interval of 10 seconds indicates system behavior without load balancing. The system performance is worse than indicated by the slight increase in delay when load balancing stops since the delay is higher for a lower traffic intensity. The queues are more heavily loaded as indicated by a decrease in probability that the queue is not full, and delay increases. While load balancing at this large balancing interval would not improve performance, this behavior does show load balancing to be effective at shorter intervals.

CHAPTER IV
SIMULATIONS OF A LOAD BALANCED NETWORK

The first part of this chapter is a description of SMPL, a simulation language, and the models used to simulate the network of stations. The model is described in two parts: the active and passive portions. Results of the simulations are provided in the remainder of the chapter.

4.1. Description of the Simulation Language

The simulations of the network of stations were performed using SMPL, a simple portable simulation language. SMPL was designed to model discrete event systems: it is an event driven language. The language is a package of subroutines, and the subroutines handle queue management and statistic gathering. A SMPL simulation model consists of *facilities* and *tokens,* where facilities are a static part of the model and tokens are active and move from facility to facility. An event corresponds to a token reserving or releasing a facility.

To simulate a network of queues using SMPL, the model must first be initialized which involves declaring facilities, initializing tokens, and scheduling initial events. After initialization, events are caused and scheduled until the simulation is complete. The types of events are the reserve and release of facilities. When the reserve of facility is caused, the following sequence of actions occurs: (1) if the facility is busy (a token has previously reserved the facility) the token is queued; (2) if the facility is idle, the facility is reserved and a release is scheduled at at some future time dependent on the service characteristics of the facility (for example, an M/M/1 system would have a release scheduled after an exponentially distributed time interval). The sequence of actions associated with the release of a facility are : (1) release the facility; (2)schedule a reserve for the released token at its next facility if necessary.

SMPL routines are called to reserve and release facilities and cause events. Routing of tokens and initialization are done in the calling language.

4.2. Description of the Simulation Model

The assumptions made for the simulations are slightly different than those for the analyses. The system is again composed of identical stations connected by a CSMA/CD network, and the stations have a finite queue limit. the time required to send a job across the network, however, is considered to exponentially distributed with a mean equal to one half the load balancing interval.

The passive portion of the system is made up of facilities. Each station connected to the CSMA/CD network is modeled by three facilities as shown in Figure 4.1. The producer models users generating jobs with a Poisson distribution and average rate λ. The server models the actual service of the job, so has a Poisson distributed service time with average μ. There is no physical significance to the exit monitor: it is used for bookkeeping and has a service time of zero.

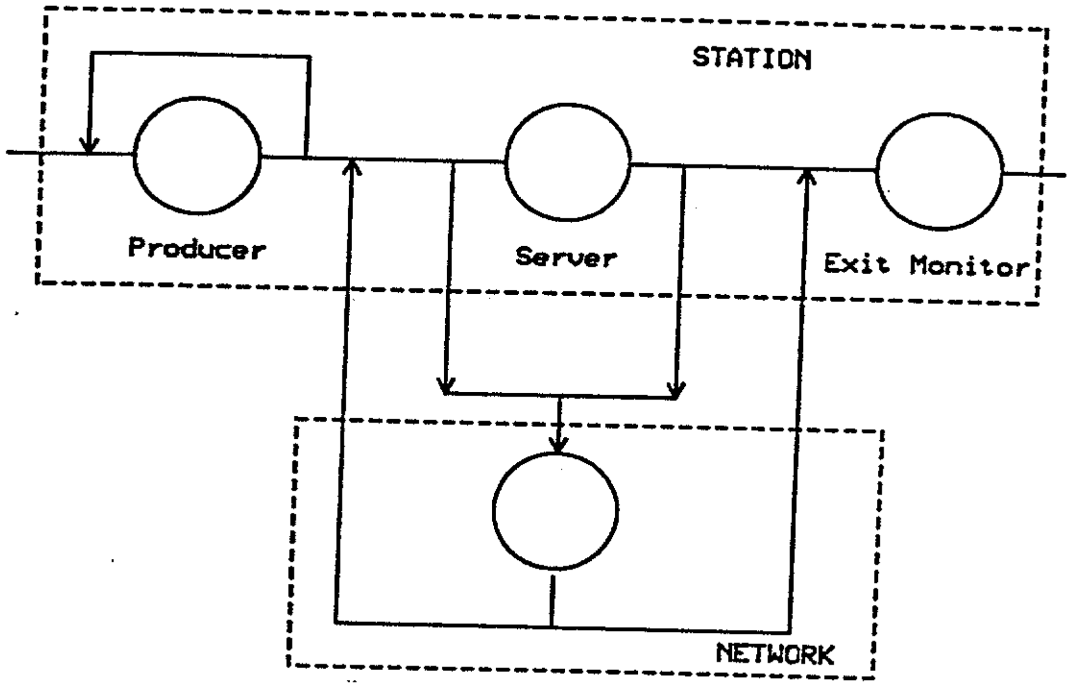

Figure 4.1 Simulation Facility Model

The CSMA/CD network is modeled by one facility. The service time at this facility corresponds to the time required to send a job across the network, and is considered Poisson distributed with a mean rate corresponding to one half the load balancing interval. This is because a load balanced job is sent across the network twice, once for job migration and once for result return.

The tokens are the active part of the model. The path that a job takes through the passive model depends on whether or not the job is involved in a load balancing operation. The job paths for local and remote execution are shown in Figure 4.2a and b respectively. All jobs are generated at the producer at one of the stations. If the job is not involved in a load balancing operation, it is routed from producer to server to exit monitor all at the same station. If the job is involved in a load balancing operation, it is routed from the producer to the CSMA/CD network (job migration), to a server at another station (remote station). When processing is complete, the job is routed to the CSMA/CD network (result return) and to the exit monitor at the station where it was generated. Jobs sent to the CSMA/CD network after being processed (result return) have a higher priority than those sent immediately after being generated (job migration). Before a job is sent for remote execution, a location is reserve for it at the destination stations queue.

4.3. The Load Balancing Decision

The objective of the simulations was to imitate reality rather than the analytical model. Behavior that is difficult to describe with equations was allowed in simulations if it could be duplicated in reality. The type of behavior referred to is reaction to fluctuating network load.

$$\text{PRODUCER}_{\text{local}} \rightarrow \text{SERVER}_{\text{local}} \rightarrow \text{EXIT MONITOR}_{\text{local}}$$

(a)

$$\text{PRODUCER}_{\text{local}} \rightarrow \text{NETWORK} \rightarrow \text{SERVER}_{\text{remote}} \rightarrow \text{NETWORK} \rightarrow \text{EXIT MONITOR}_{\text{local}}$$

(b)

Figure 4.2 Simulation Job Paths

The load balancing decision is made with the following steps. First the queue lengths for the maximally loaded station and the minimally loaded station are determined. If the queue length at the current station is the same as the queue length at the maximally loaded station, then the decision proceeds to the next step. Information is maintained at all stations about the time required to send and return the last ten load balanced jobs. The time for local execution, and execution at the minimally loaded station (remote execution) are calculated. The average time to send and return jobs are added to the remote execution time, and if the sum plus a threshold savings is less than the local execution time, then the load balancing operation is initiated.

The values maintained on the last ten jobs can increase to a point where no jobs are load balanced, so the average time for a load balancing operation does not change. In order to keep this from happening, an entry is zeroed periodically to keep the average current. This also lowers the average slightly in all cases, so a threshold savings of load balancing interval is used.

The source listing for a simulation is shown in the Appendix. The program was written in the C programming language. Sample input and output are also included.

4.4. Simulation Results

The system had eight stations modeled as M/M/1 queueing systems each with queue limit ten. Load balancing intervals of 0.0001 seconds to 100.0 seconds were simulated. An arrival rate of 0.1 jobs per second seconds is used with service rates ranging from 0.2 jobs per second to 0.9 jobs per second.

The system performance expected for the simulations is similar to that expected for the analysis done in Chapter 3. Since load balancing is done, delays lower than the delay of an isolated M/M/1 queueing system with the same parameters as a station connected to the network are expected. Only one job is balanced at a time, so the delay is expected to be higher than the delay of a fast M/M/1 system with average arrival and departure rates of eight time those of a single station.

Stations have the model show in Chapter 3 where jobs are held until space is available in the station's queue. The effective arrival rate to the station is λ' where $\lambda' = \lambda \times Pr(queue\ is\ not\ full)$. For simulation, $Pr(queue\ is\ not\ full)$ was determined by counting arrivals and dividing the number when the queues are full by the total number of arrivals.

A summary of load balancing intervals, traffic intensity, probabilities of the queue being full, and the effective traffic intensity is shown in Table 4.1. Information is provided only for traffic intensities of 0.6 and greater because below those intensities, space is always available in the stations queues. A general trend seen is that the effective traffic intensity decreases as the load balancing interval increases for load balancing interval from 0.0001 seconds to 10 seconds. For higher traffic intensities, the drop at a load balancing interval of 10 seconds becomes more pronounced. There is an increase in the effective traffic intensity at an interval of 100 seconds which corresponds to no jobs being load balanced.

The system parameters monitored during the simulation were the number of jobs load balanced (Figure 4.3), the total time all jobs spent in the system (Figure 4.4), the throughput of the system (Figure 4.5), and the average delay of a job or the average response time (Figure 4.6).

Table 4.1 Table of Effective Traffic Intensities for Simulation

Load Balancing Interval	Traffic Intensity (ρ)	Probability Queue Not Full	Effective Traffic Intensity (ρ')
0.0001	0.6	1.0	0.6
0.001	0.6	1.0	0.6
0.01	0.6	1.0	0.6
0.1	0.6	1.0	0.6
1.0	0.6	1.0	0.6
10.0	0.6	0.9993	0.5958
100.0	0.6	0.9987	0.5992
0.0001	0.7	1.0	0.7
0.001	0.7	1.0	0.7
0.01	0.7	0.9993	0.6995
0.1	0.7	0.9998	0.6999
1.0	0.7	0.9997	0.6998
10.0	0.7	0.9348	0.6544
100.0	0.7	0.9948	0.6964
0.0001	0.8	0.9944	0.7955
0.001	0.8	0.9941	0.7953
0.01	0.8	0.9969	0.7975
0.1	0.8	0.9957	0.7966
1.0	0.8	0.9657	0.7726
10.0	0.8	0.5159	0.4127
100.0	0.8	0.9831	0.7865
0.0001	0.9	0.9724	0.8752
0.001	0.9	0.9712	0.8741
0.01	0.9	0.9744	0.8770
0.1	0.9	0.9722	0.8750
1.0	0.9	0.8535	0.7682
10.0	0.9	0.1611	0.1450
100.0	0.9	0.9525	0.8573

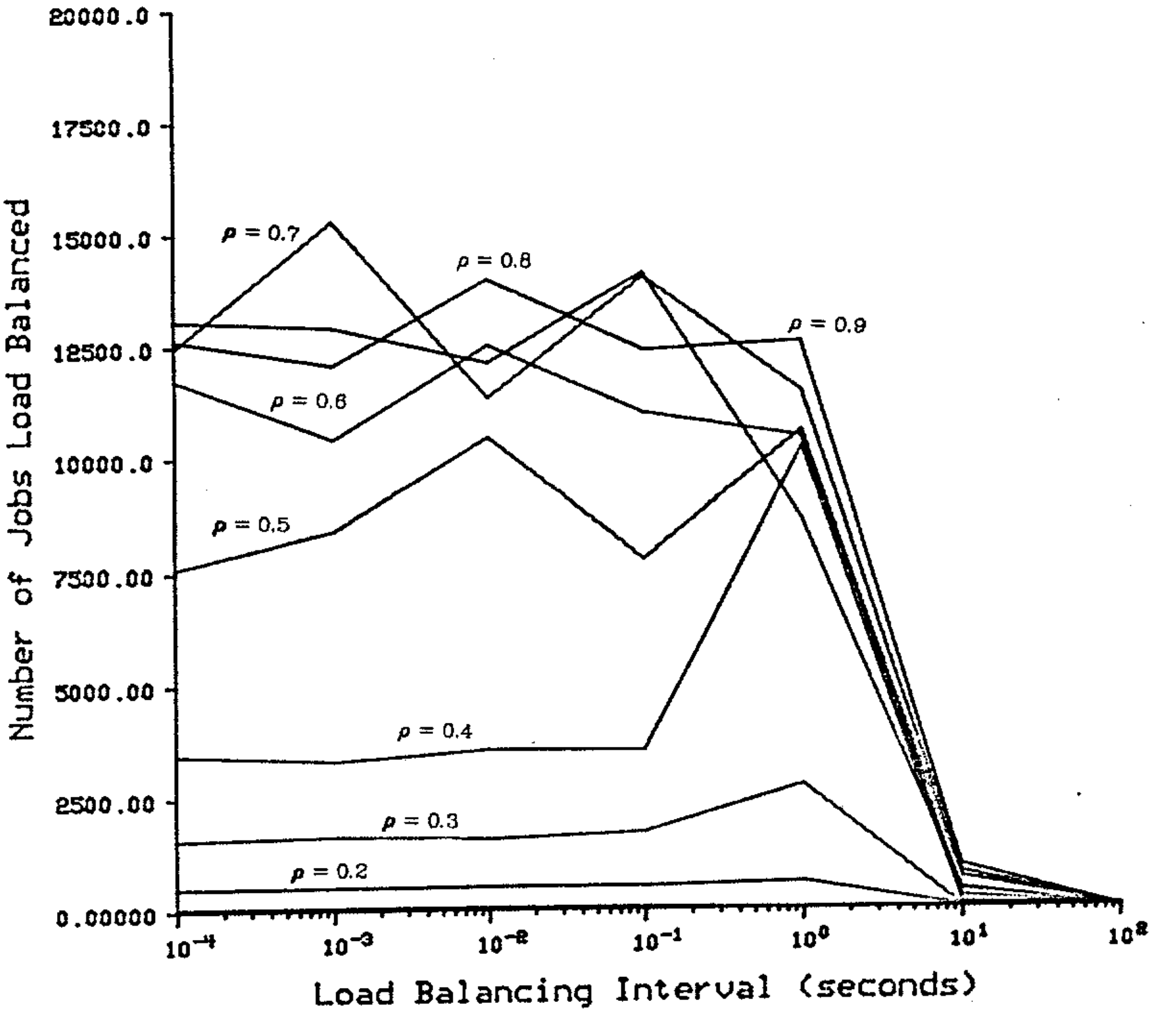

Figure 4.3 Number of Jobs Balanced versus Load Balancing Interval

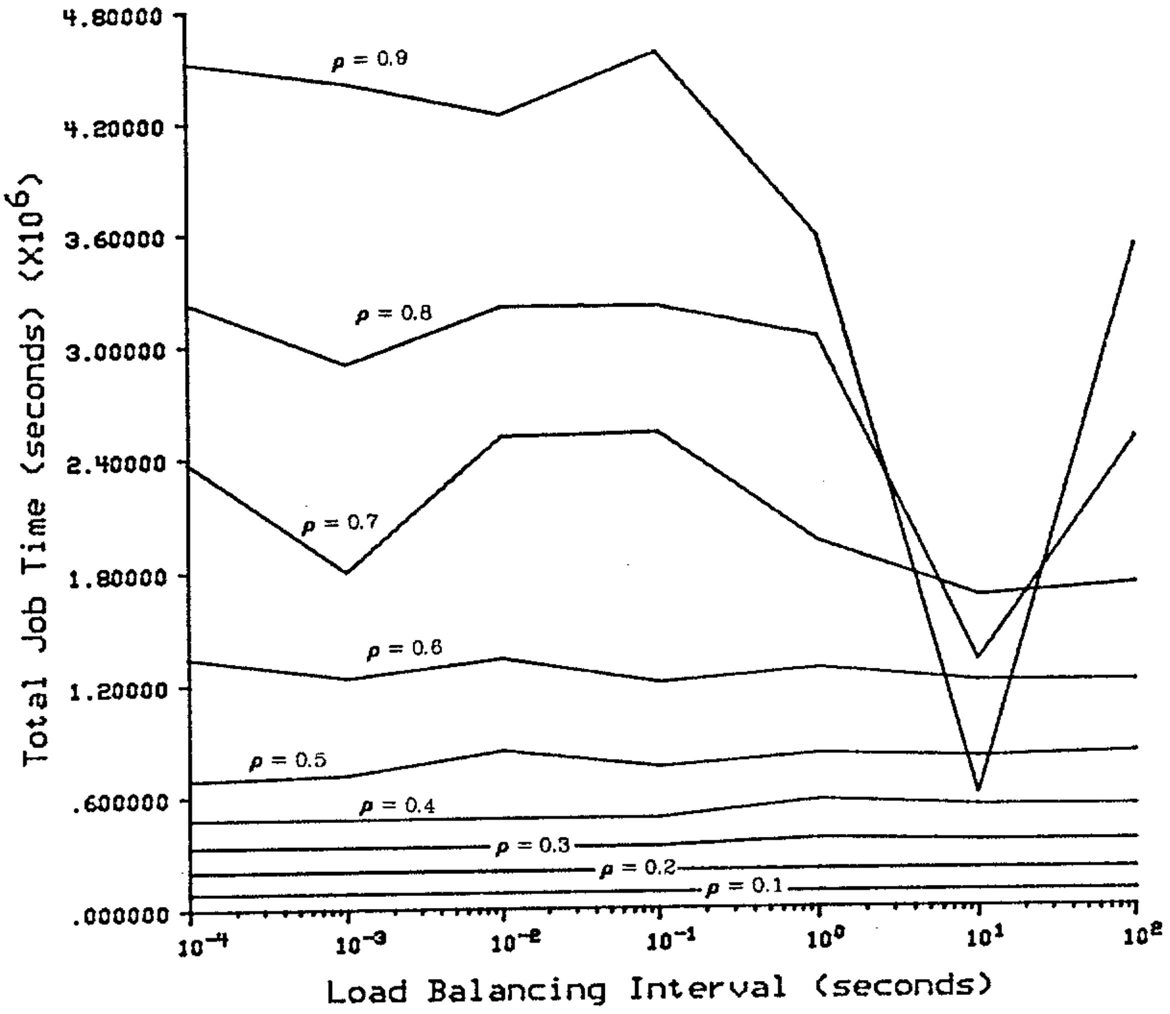

Figure 4.4 Total Job Time versus Load Balancing Interval

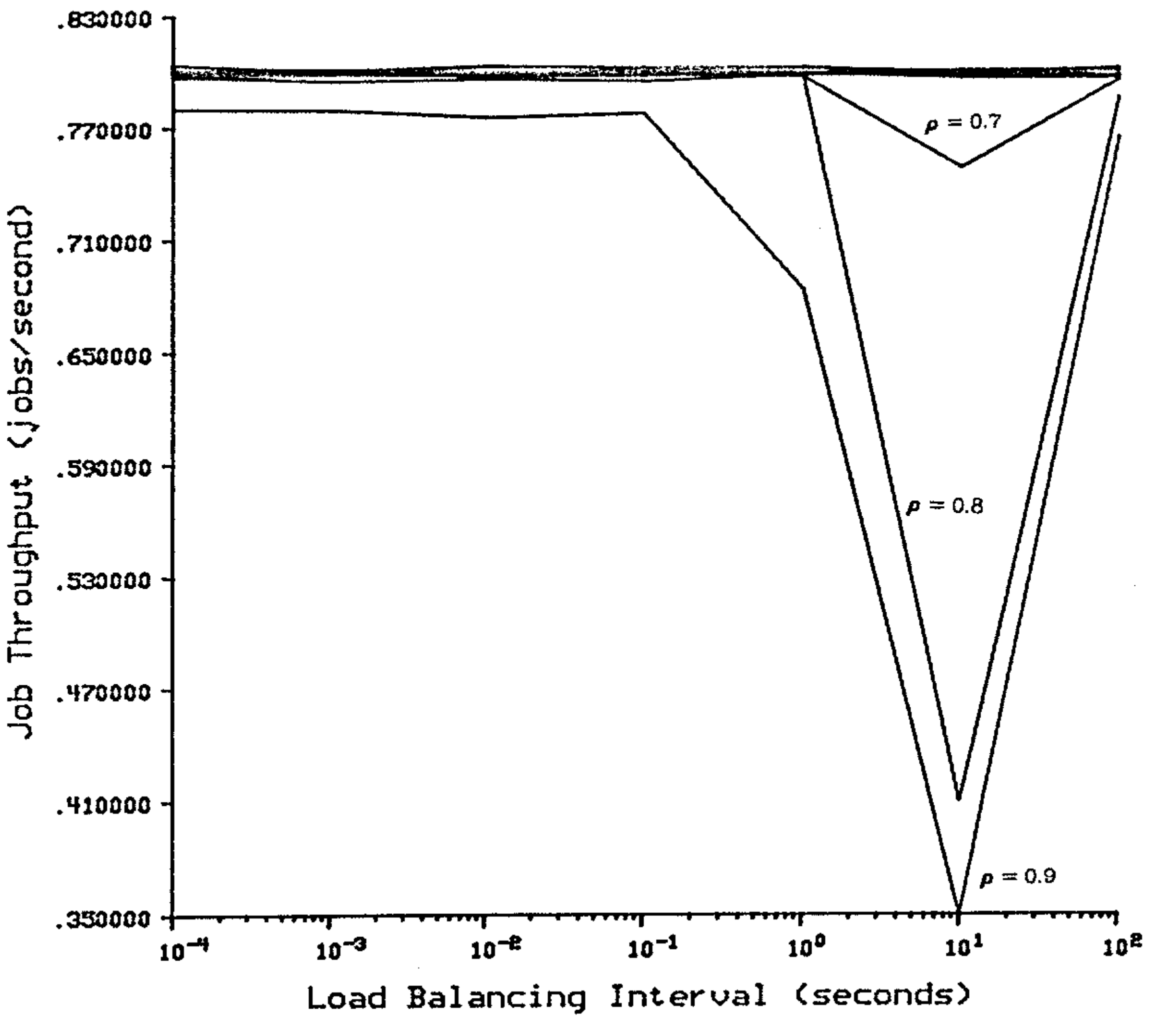

Figure 4.5 System Throughput versus Load Balancing Interval

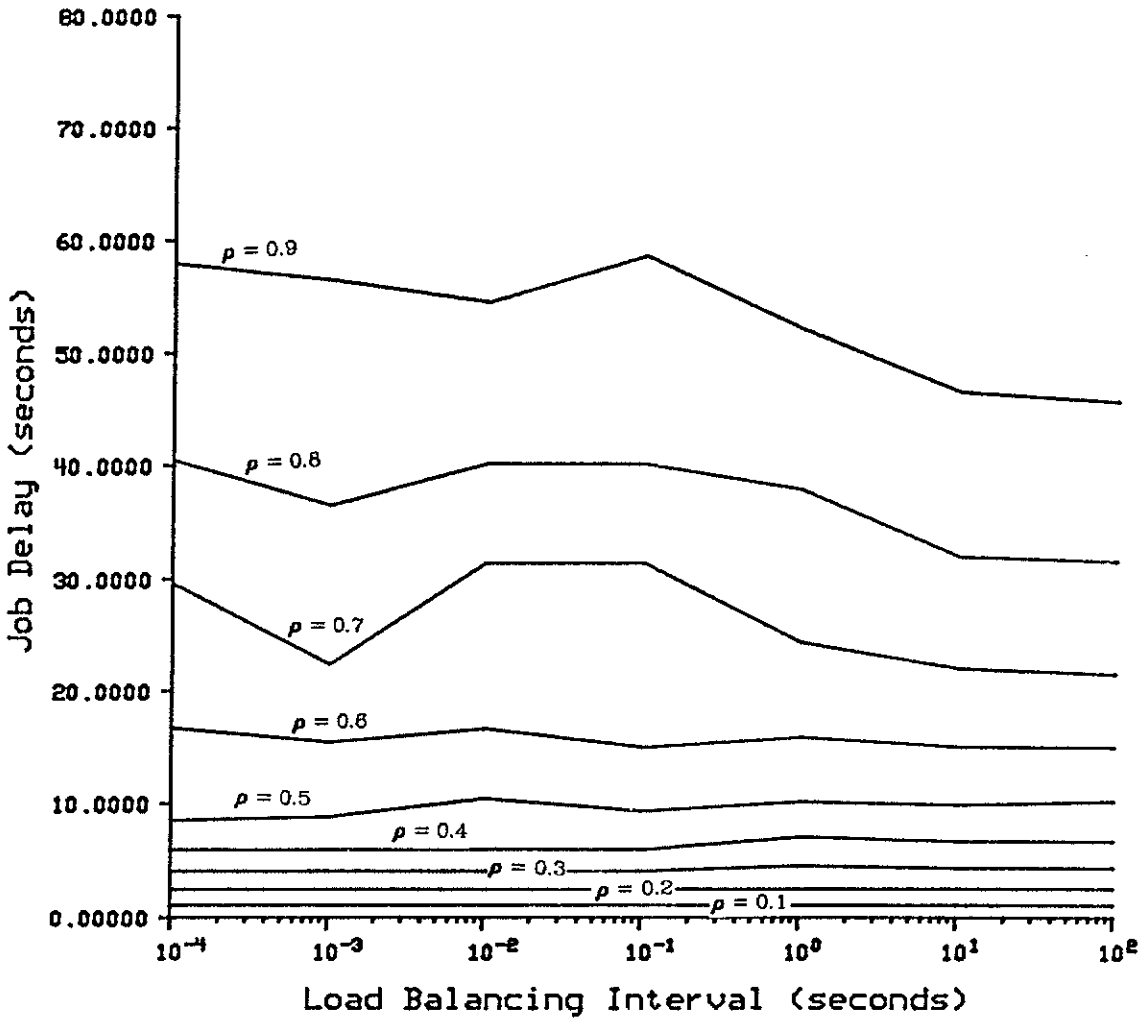

Figure 4.6 Average Job Delay versus Load Balancing Interval

The number of jobs load balanced versus the load balancing interval is shown in Figure 4.3. Each plot corresponds to a different traffic intensity. The general trend is that as the load balancing interval increases, the number of jobs load balanced decreases. At a load balancing interval of 10 seconds, the number of jobs balanced sometimes peaks. This will be explained when consideration is given to other parameters. The general trend for the number of jobs balanced is expected. Since a decision to load balance is made based on the amount of time required for a load balancing operation, it follows that fewer jobs will be balanced as the load balancing interval increases. No jobs are balanced at a load balancing interval of 100 seconds.

The next system parameter is total job time (Figure 4.4). The total job time is fairly constant at low traffic intensities. As ρ increases, a slight increase in the job time is seen with the increase in load balancing interval. As ρ nears 0.7, the increase is less apparent and at $\rho = 0.8$ the job time decreases as the load balancing interval increases. At a load balancing interval of 10 seconds, there is a large decrease in the total job time for traffic intensities of 0.8 and 0.9. These decreases correspond to large decreases in the effective traffic intensity as shown in Table 4.1. As the effective arrival rate decreases, the number of jobs in the system decreases causing an overall decrease in job time.

The job throughput, Figure 4.5, is constant for lower traffic intensities but shows a decrease at a load balancing interval of 10 seconds for a traffic intensity of 0.7, and a much larger decrease at the same interval for traffic intensities of 0.8 and 0.9. These points once again correspond to a decrease in the effective traffic intensity. The sharp decrease in effective traffic intensity corresponds to a decrease in the probability that the queues have available space for more jobs and that indicates that fewer jobs are completed.

Consequently the throughput will decrease.

The final system parameter monitored is the average job delay (Figure 4.7). For low traffic intensities, the delay increases slightly as the load balancing interval increases. This makes intuitive sense as the delay for a job involved in a load balancing operation will increase with increases in the load balancing interval, this in turn increases average job delay. As ρ reaches values of 0.6 and larger, the delay shows little increase at the load balancing interval increases and actually shows a decrease at high traffic intensities. Considering the effective traffic intensity, the decrease in delay at higher traffic intensities corresponds to a decrease in the effective traffic intensity, as well as a decrease in the number of jobs load balanced. Improvement in system performance is not entirely because load balancing is less frequent, but is in part due to the decrease in the effective traffic intensity. Load balancing has an effect on the arrival rate at destination processors which can be detrimental. As ρ increases, load balancing increases the arrival rate at the destination processors which can cause instability so high delays at that processor. This consequently increases the average job delay. This implies that load balancing is detrimental to system performance at high traffic intensities.

The general conclusions from simulation is that load balancing becomes less valuable as the traffic intensity increases. Load balancing is beneficial, however, at low traffic intensities. An additional conclusion is that the benefit of load balancing decreases at the load balancing interval nears the average service time a station. In this case, the average service time ranged from one to ten seconds. Delays increased most noticeably at intervals of 10 seconds.

CHAPTER V
CONCLUSIONS

The first section of this chapter compares the analytical and simulation models presented in Chapters 3 and 4. The results from both the analysis and simulation are compared with consideration given to the differences in the models. In the remainder of the chapter some general remarks about this research are made.

5.1. Comparison of Analytical and Simulation Results

The differences between the simulation model and the analytical model were the queue limit and the manner in which the CSMA/CD network was modeled. The queue limit for the analytical model was three compared to a queue limit of ten for the simulation model.

The CSMA/CD network was modeled as an M/M/1 system for the simulations. This allowed queueing delays associated with job transmission for load balanced jobs. For the analytical model, the transmission time was simply the load balance interval, a constant. There was no delay associated with gaining access to the network.

The reason for the differences between the two models is the amount of computation necessary for the analysis. If the queue limit were ten, the state transition matrix would be 1.59×10^{11} by 1.59×10^{11}. The time and memory required to solve for average response time make the problem intractable.

The CSMA/CD network modeling also arises from a need to make the problem computationally feasible. If the network were modeled as an M/M/1 system for the analysis, the state transition matrix for a single station during a load balancing interval would be very difficult to compute.

Given these differences and model limitations, the following results are expected at low traffic intensities. The delay found by the analysis will be less than that found by simulation because of the queueing delay associated with the way the CSMA/CD network is modeled for simulation. The effective traffic intensity for the analysis will be smaller than for the simulation because the model for the analysis has a shorter queue length, the probability that space will be available in a stations queue is smaller. This causes the effective arrival rate, and so the effective traffic intensity to be smaller. Both the analysis and simulation should result in lower delays than the M/M/1 system with the same arrival and service rates as a station connected to the network because of the models' queue limits. At low traffic intensities, the improvement will not be large because the average queue lengths are low and thus load balancing operations are not frequently beneficial.

Figure 5.1 shows a comparison of the delay versus traffic intensity for the largest load balancing interval for both models and for the isolated M/M/1 system with the same parameters as a station. The values for all are close at low traffic intensities, as expected. As the traffic intensity increases, the delays for both models decrease relative to the M/M/1's delay due to the decrease in effective traffic intensities. The delay from the analysis is always less than the delay from the simulation which is expected because the effective traffic intensity is always lower for the analysis for the corresponding actual traffic

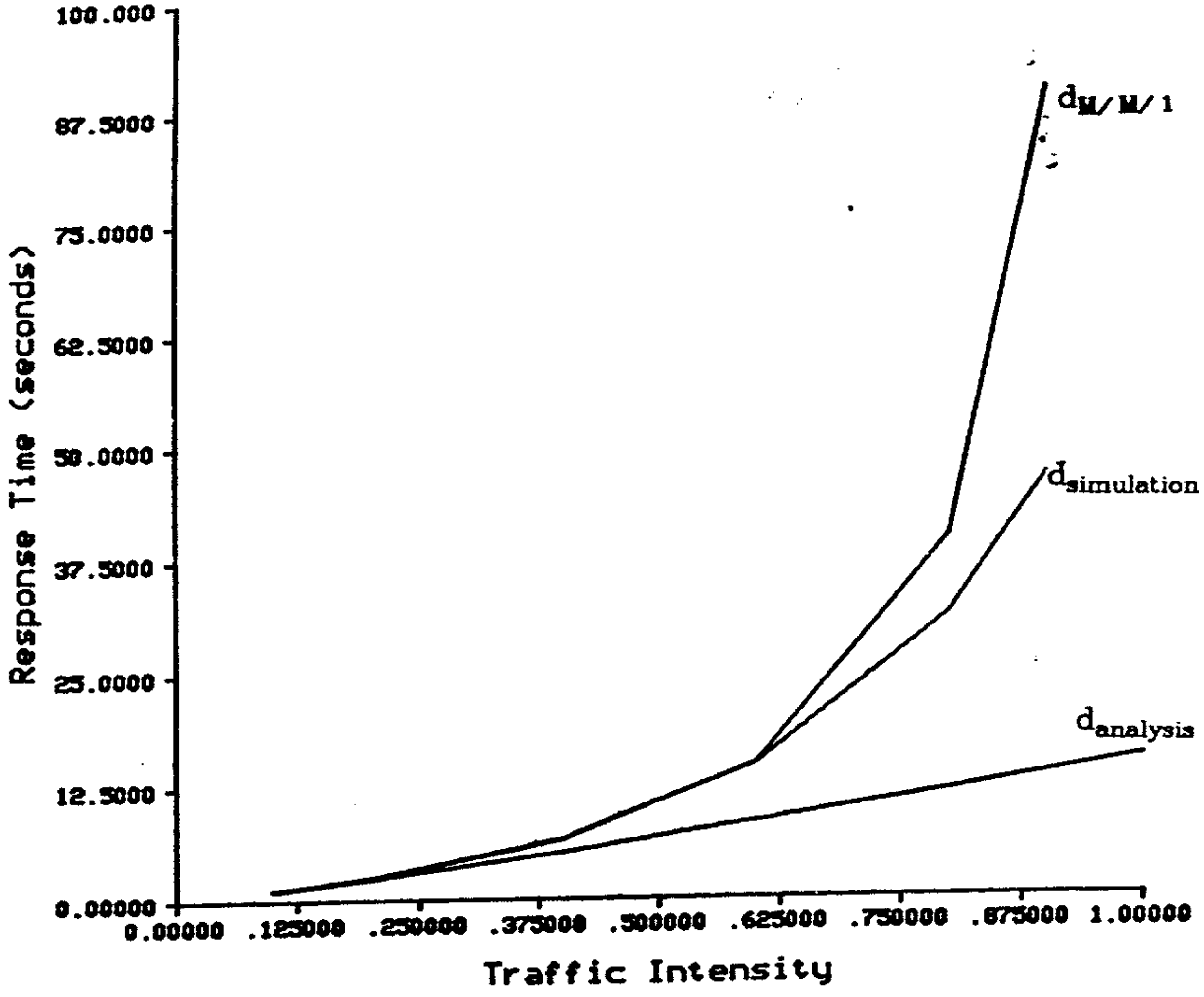

Figure 5.1 Comparison Between Analysis and Simulations without Load Balancing

intensity.

Figure 5.2 shows a comparison of the models for a short load balancing interval at which load balancing is more frequently performed. The delays are slightly less than the M/M/1 for low traffic intensities. As traffic intensities increase the delays of the models are much improved over the M/M/1. The analytical model continues to show lower delays as the traffic intensity becomes higher, but the simulation delay becomes greater. This is because of the combined effects of the queueing delay incurred in gaining network access and the instability that occurs when load balancing at high traffic intensities.

5.2. Final Remarks

The models we have studied are accurate for low traffic intensities. The simulations indicate that load balancing as not beneficial for high traffic intensities due to instabilities that can result at stations where jobs are sent for remote processing. In addition the benefits of load balancing decrease as the load balance interval approaches the average service time of a station.

It is not possible to obtain a probability distribution for the response times analytically. This information readily obtained from the history of the systems behavior.

The information gathered can be applied to load balancing on networks with bus topologies. If the characteristics of the station connected by the network are varied and the loads are varied, this conclusion may not apply. Additionally load balancing will be most effective at lower traffic intensities and load balancing intervals that are small compared to the average job service time. For this research, delay was decreased most significantly for these conditions.

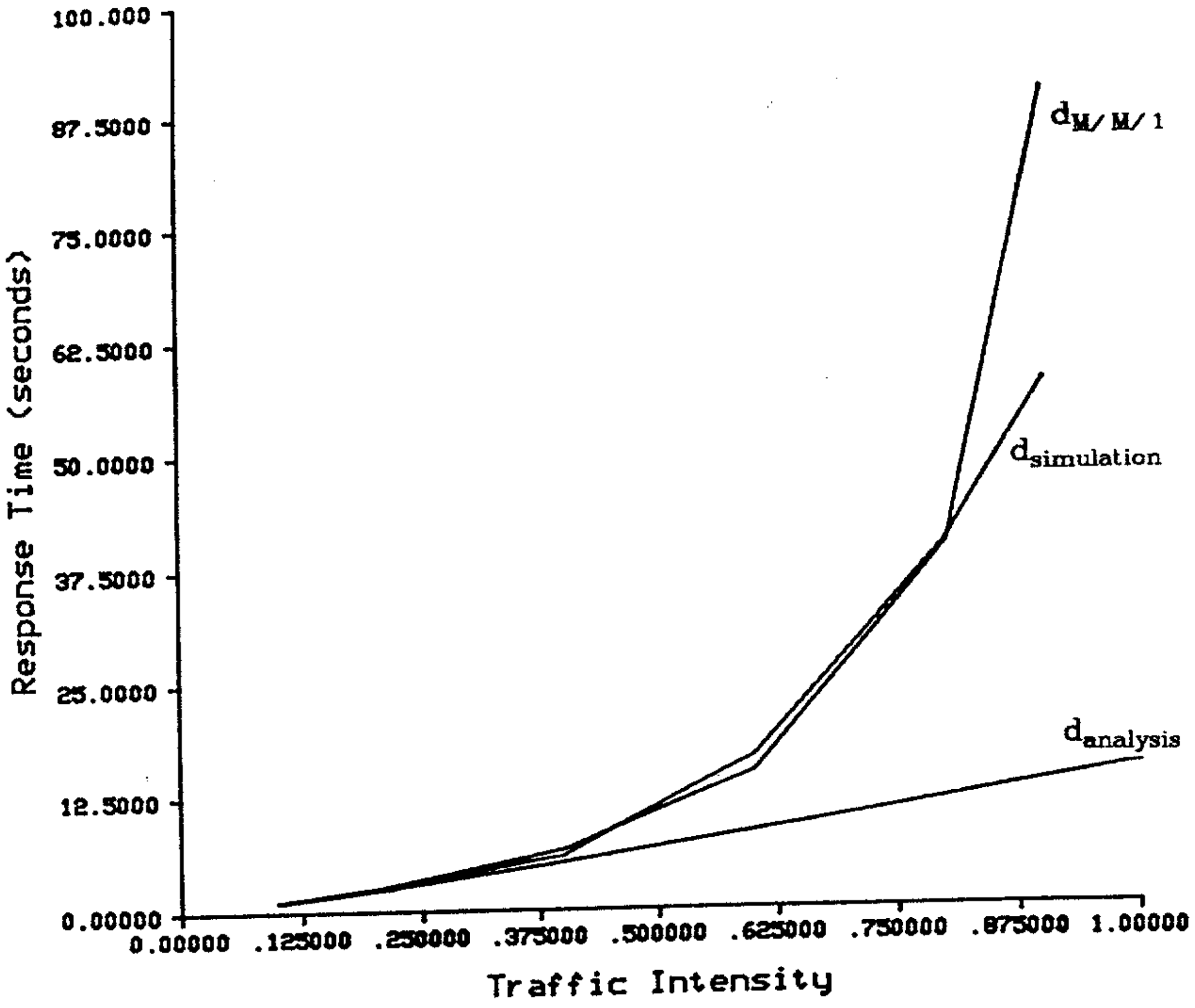

Figure 5.2 Comparison Between Analysis and Simulations with Load Balancing

The method for making load balancing decisions for the simulations could be extended to make more conservative decisions at higher traffic intensities. One way to accomplish this is to require a savings of some minimum amount, or threshold. The threshold could be made a function of the load causing the number of jobs load balanced to decrease significantly at high loads. Another extension of this study could be to determine the benefit of connecting the stations by two CSMA/CD networks and dedicating one to load balancing.

LIST OF REFERENCES

LIST OF REFERENCES

[Abr70] Abramson, N., The ALOHA System - Another Alternative for Computer Communications," *AFIPS Conference Proceedings,* Vol. 37, September 1970, pp. 281-285.

[Cap77] Capetanakis, J. I., *The Multiple Access Broadcast Channel: Protocol and Capacity Considerations,* Phd Thesis, Massachusetts Institute of Technology, 1977.

[ChK79] Chow, Y. C. and Kohler, W., Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Transactions on Computers,* Vol. C-28, May 1979, pp. 334-361.

[Fay77] Fayolle, G. et al., Stability and Optimal Control of the Packet Switching Broadcast Channel," *JACM,* Vol. 24, July 1977, pp. 375-386.

[Gal78] Gallagher, R., Conflict Resolution in Random Access Broadcast Networks," *Proceedings of AFSOR Workshop on Communication Theory and Applications,* September 1978, pp. 74-76.

[Hlu81] Hluchyj, M. J., *Multiple Access Communcation: The Finite User Population Problem,* Phd Thesis, Massachusetts Institute of

Technology, 1981.

[Hwa82] Hwang, K. et al., A Unix-Based Local Computer Network With Load Balancing," *IEEE Computer,* Vol. 15 , April 1982, pp. 55-66.

[JuW84] Juang, J. Y. and Wah, B. W. , Unified Window Protocol for Local Multiaccess Networks," *Proceedings of Third Annual Joint Conference of the IEEE Computer and Communications Societies,* April 1984, pp. 23.

[Kle75] Kleinrock L., *Queueing Systems Volume 1: Theory,* John Wiley, New York, 1975.

[Mar70] Martin, J., *Teleprocessing Network Organization,* Prentice Hall, Englewood Cliffs, New Jersey, 1970.

[NiH81] Ni, L. M. and Hwang, K., Optimal Load Balancing Strategies for a Multiple Processor System," *Proceedings of 10th International Conference on Parallel Processing,* August 1981, pp. 352-357.

[RoW77] Rothauser, E. H. and Wild, D., MLMA - A Collision-Free Multi-Access Method," *Proceedings IFIP Congr.,* December 1977, pp. 431-436.

[Sch76] Scholl, M., *Multiplexing Techniques for Data Transmission over Packet Switched Radio Systems,* Phd Thesis, Computer Science Department, UCLA, 1976.

[SiM81] Siegel, H. J. and McMillen, R. J., The Multistage Cube: A Versatile Interconnection Network," *IEEE Computer,* December 1981, pp. 65-76.

[Sne75] Snell, J. L., *Introduction to Probability Theory with Computing,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[Sta84] Stallings, W., *Local Networks,* Macmillan Publishing Company, New York, New York, 1984.

[Tan81] Tanenbaum, A. S., *Computer Networks,* Prentice Hall Inc., New Jersey, 1981.

[ToV82] Towsley, D. and Venkatesh, G., Window Random Access Protocols for Local Computer Networks," *IEEE Transactions on Computers,* Vol. C-31, August 1982, pp. 715-722.

[WaJ83] Wah, B. W. and Juang, J. Y. , Load Balancing on Local Multiaccess Networks," *Proceedings of 8th Conference on Local Computer Networks,* October 1983, pp. 55-61.

APPENDICES

Appendix A. Analysis Programs

```
/*
 *      This is the header file for part 1 of the response time finder
 *
 *      PART 1
 *      Part 1 of the response time finder computes the states transition
 *      probabilities for a single station (processor).
 *
 *      related files:
 *      main.c
 *      stepmat.c
 *      matmanip.c
 */
#include        <math.h>
#include        <stdio.h>
#include        <ctype.h>

/* isinteger - returns true if the argument is an integer */
#define isinteger(x)  (fabs((double)(x) - (double)((int)(x))) < EPSILON)

/* lnf - does a lint feed from standard input */
#define lnf(f)  {while(getc(f) != '\n');}

/************* DEFINING CONSTANTS ***************/

#define         MAXDIM          8       /* maximum matrix dimensions */

/* small numbers */
#define         EPSILON         1e-18
#define         ACCURACY        1e-8

#define         TRUE            1
#define         FALSE           0

/************* DEFINING SUBPROGRAMS *************/
double  stepprob();
double  poisson();
double  factorial();
int     eqmat();
double  atof();


/************* DEFINING GLOBAL VARIABLES *************/
/*      markov  pr(i --> j | k steps)
 *      markov0 initial value of markov
 *      ps      state transition matrix for a given interval pr( i --> j)
 *      adn     number of legal combinations of arrivals and departures
 *      adn0    initial values of adn
 *      est     estimated values of the state transition matrix pr( i --> j)
 *      norm    sum over k of pr(k steps | i-->j)
 */
double  markov[MAXDIM][MAXDIM],
        markov0[MAXDIM][MAXDIM],
        ps[MAXDIM][MAXDIM],
        adn[MAXDIM][MAXDIM],
        adn0[MAXDIM][MAXDIM],
        est[MAXDIM][MAXDIM],
        norm[MAXDIM][MAXDIM];
```

```
int     stepping;       /* flag indicating incremental probabilities are
                         * significant */
int     nonzeroseen;    /* flag to indicate if non zero value found */
int     nstat;          /* number of stations */
int     qlen;           /* queue limit */
double  arrate;         /* arrival rate */
double  deprate;        /* departure rate */
double  interval;       /* load balance interval */

/*
 *      This program does preliminary calculations for determination of the
 *      average response time of a network of processors.
 *
 *      State transition matrices for a single station (processor) are
 *      calculated for queue limit specified by input parameters, and for queue
 *      limit one less than specified by input parameters.
 *
 *      INPUT:  read from standard input.  Data should be entered on separate
 *              lines.  Only the first field on each line is significant
 *              Input format follows:
 *
 *              (1) number of stations (processors)
 *              (2) queue limit
 *              (3) arrival rate
 *              (4) departure rate
 *              (5) load balance interval
 *
 *      OUTPUT: written to standard output in the format
 *
 *              (1) queue limit
 *              (2) number of stations
 *              (3) arrival rate
 *              (4) departure rate
 *              (5) load balance interval
 *              (6) state transition matrix for queue limit
 *              (7) state transition matrix for (queue limit - 1)
 */
main()
{
        /* read input parameters from standard input, and echo to standard
         * output
         */
        readin();
        qlen++;

        /* compute the state transition matrix for the queue limit specified by
         * input parameters, and for queue limit one less than specified by
         * input parameters
         */
        computetrans();
        qlen--;
        computetrans();

}
/*      computetrans()  compute the state transition matrix
 *
 *      globals         ps              two dimensional double matrix
 *                                      state transtion matrix
 *                      est             two dimensional double matrix
 *                                      estimate of ps
 */
computetrans()
{
        int     noest;          /* counts number of estimates */

        /* initialize beginning of interval */
        compnorm();
        noest=0;

        zeromat(est);
        initialize();

        /* check if current estimate is stable */
        while ((noteq(ps,est)) && (noest < 100)){
                noest=noest+1;
                newest();
        }

        /* output the state transition matrix */
        matout(ps);
}
/*
 *      newest()                calculate a new estimate based on the
 *                              old estimate
 */
newest()
{
        int     step;

        /* initialize for next estimate */
        estinit();
        step=0;
        nonzeroseen = FALSE;

        /* continue computing probabilities until increments are
        * insignificant */
        while (stepping == TRUE || nonzeroseen == FALSE ){
                stepping = FALSE;

                /* compute current probability increments */
                curprob(step);

                /* compute probability of number of steps */
                matmul(markov0,markov);
                matmul(adn0,adn);

                step++;
        }
}
/*
 *      readin()        read input parameters and echo to output
 *
 *      globals         qlen            int     queue limit
 *                      nstat           int     number of processors
 *                      arrate          double  arrival rate
 *                      deprate         double  departure rate
 */
readin()
{
        /* get number of stations */
        scanf("%d",&nstat); lnf(stdin);

        /* get maximum queue length */
        scanf("%d",&qlen); lnf(stdin);

        /* get arrival rate */
        scanf("%f",&arrate); lnf(stdin);

        /* get departure rate */
        scanf("%f",&deprate); lnf(stdin);

        /* get load balance interval */
        scanf("%f",&interval); lnf(stdin);

        /* echo input to standard output */
        printf("%d\t;Queue Length\n",qlen);
        printf("%d\t;Number of stations\n",nstat);
        printf("%g\t;Arrival Rate\n",arrate);
        printf("%g\t;Departure Rate\n",deprate);
        printf("%g\t;Load Balance Interval\n",interval);
        fflush(stdout);
}

/*
 *      stepprob()      compute pr(k steps | i --> j)
 *                      function value returned is the probability
 *
 *      parameters      source  int     source state
 *                      dest    int     destination state
 *                      steps   int     num of steps to get from source to dest
 */
double stepprob(source,dest,steps)
int     source,
        dest,
        steps;
{
        double ar,de,pr;

        ar = ((double)(steps) + (double)(dest) - (double)(source))/2;
        de = ((double)(steps) - (double)(dest) + (double)(source))/2;

        if ((isinteger(ar) && isinteger(de)) && ((ar>=0) && (de>=0))){
                pr = poisson(arrate,interval,ar) * poisson(deprate,interval,de);
        }else{
                pr = 0.0;
        }
        pr = pr * adn[source][dest];
        return(pr);
}
/*
 *      poisson()       computes the probability of k events for
 *                      a poisson distribution with parameter
 *                      lamda * T
 *                      function value returned in the probability
 *
 *      parameters      lamda   double  arrival or departure rate
 *                      T       double  time interval
 *                      k       double  number of events
 */
double  poisson(lamda,T,k)
double  lamda;
double  T;
double  k;
{
        double  parameter,
                numerator;
        register        int     i;

        parameter = lamda * T;

        numerator = exp(-parameter/2);
        numerator *= pow(parameter,k/2);

        i = k;
        if(i == 0) i++;
        do {
                numerator /= i;
        } while(--i);
        numerator *= exp(-parameter/2) * pow(parameter,k/2);
        return(numerator);
}
/*
 *      curprob()       computes probabilities for current step
 *
 *      parameters      step    int     number of steps
 */
curprob(step)
int     step;
{
        int     row,col;
        double  pr;             /* pr - sum over j of
                                        pr(k steps | i --> j) * pr(i --> j) */
        double  prij;           /* prij - pr(k steps | i --> j) */

        for (row=0;row<qlen;row++){
                pr = 0.0;
                for (col=0;col<qlen;col++){
                        if ((prij=stepprob(row,col,step))>= EPSILON){
                                nonzeroseen = TRUE;
                                stepping = TRUE;
                                pr += prij/norm[row][col] * est[row][col];
                        }
                }
                for (col=0;col<qlen;col++){
                        ps[row][col] += markov[row][col] * pr;
                }
        }
}
/*
 *      initialize()    initializes the matricies est and ps for the
 *                      beginning of an interval.  est is zeroed.
 *                      ps is set to the initial estimate which sums
 *                      to 1 across the rows.
 *
 *      globals         est     two dimensional double matrix
 *                      ps      two dimensional double matrix
 *                      qlen    int     queue limit
 */
initialize()
{
        int row,col;

        for (row=0;row<qlen;row++){
                for (col=0;col<qlen;col++){
                        ps[row][col] = 1.0 / (double) (qlen);
                }
        }
        zeromat(est);
}

/*
 *      estinit()       initializes matricies ps and est, the variable
 *                      stepping for the beginning of an estimate
 *
 *      globals         est             two dimensional double matrix
 *                                      estimate of ps
 *                      ps              two dimensional double matrix
 *                                      state transition matrix
 *                      stepping        int
 *                                      flag to for significant steps
 */
estinit()
{
        eqmat(ps,est);
        loadpmat();
        zeromat(ps);
        stepping=TRUE;
}

/*
 *      stepprob()      compute pr(k steps | i --> j)
 *                      function value returned is the probability
 *
 *      parameters      source  int     source state
 *                      dest    int     destination state
 *                      steps   int     num of steps to get from source to dest
 */
double stepprob(source,dest,steps)
int     source,
        dest,
        steps;
{
        double ar,de,pr;

        ar = ((double)(steps) + (double)(dest) - (double)(source))/2;
        de = ((double)(steps) - (double)(dest) + (double)(source))/2;

        if ((isinteger(ar) && isinteger(de)) && ((ar>=0) && (de>=0))){
                pr = poisson(arrate,interval,ar) * poisson(deprate,interval,de);
        }else{
                pr = 0.0;
        }
        pr = pr * adn[source][dest];
        return(pr);
}
/*
 *      poisson()       computes the probability of k events for
 *                      a poisson distribution with parameter
 *                      lamda * T
 *                      function value returned in the probability
 *
 *      parameters      lamda   double  arrival or departure rate
 *                      T       double  time interval
 *                      k       double  number of events
 */
double  poisson(lamda,T,k)
double  lamda;
double  T;
double  k;
{
        double  parameter,
                numerator;
        register        int     i;

        parameter = lamda * T;

        numerator = exp(-parameter/2);
        numerator *= pow(parameter,k/2);

        i = k;
        if(i == 0) i++;
        do {
                numerator /= i;
        } while(--i);
        numerator *= exp(-parameter/2) * pow(parameter,k/2);
        return(numerator);
}
/*
 *      curprob()       computes probabilities for current step
 *
 *      parameters      step    int     number of steps
 */
curprob(step)
int     step;
{
        int     row,col;
        double  pr;             /* pr - sum over j of
                                        pr(k steps | i --> j) * pr(i --> j) */
        double  prij;           /* prij - pr(k steps | i --> j) */

        for (row=0;row<qlen;row++){
                pr = 0.0;
                for (col=0;col<qlen;col++){
                        if ((prij=stepprob(row,col,step))>= EPSILON){
                                nonzeroseen = TRUE;
                                stepping = TRUE;
                                pr += prij/norm[row][col] * est[row][col];
                        }
                }
                for (col=0;col<qlen;col++){
                        ps[row][col] += markov[row][col] * pr;
                }
        }
}
/*
 *      initialize()    initializes the matricies est and ps for the
 *                      beginning of an interval.  est is zeroed.
 *                      ps is set to the initial estimate which sums
 *                      to 1 across the rows.
 *
 *      globals         est     two dimensional double matrix
 *                      ps      two dimensional double matrix
 *                      qlen    int     queue limit
 */
initialize()
{
        int row,col;

        for (row=0;row<qlen;row++){
                for (col=0;col<qlen;col++){
                        ps[row][col] = 1.0 / (double) (qlen);
                }
        }
        zeromat(est);
}

/*
 *      estinit()       initializes matricies ps and est, the variable
 *                      stepping for the beginning of an estimate
 *
 *      globals         est             two dimensional double matrix
 *                                      estimate of ps
 *                      ps              two dimensional double matrix
 *                                      state transition matrix
 *                      stepping        int
 *                                      flag to for significant steps
 */
estinit()
{
        eqmat(ps,est);
        loadpmat();
        zeromat(ps);
        stepping=TRUE;
}
```

```
/*
 *  header file for state transition matrix solver part 2
 */

#include <stdio.h>
#include <math.h>
#include <ctype.h>

/******************** constant definition */

#define         HASHSIZE        512     /* length of hash table */
#define         AVGLEN          50      /* average number of configurations for
                                         * each hash table entry
                                         */
#define MAXQLENGTH      9       /* maximum queue length */

#define TRUE            1
#define FALSE           0

#define ENDF            -1      /* end of configuration marker */

#define MAJORCONFIGURATIONS     200     /* maximum number of major
                                         * configurations */
#define MINORCONFIGURATIONS     25      /* maximum number of minor
                                         * configurations */

#define         lnf(f)          {while(getc(f) != '\n');}


/************ stucture and variable definition */

struct transition {     /* from to configurations */
        int     from[MAXQLENGTH];
        int     to[MAXQLENGTH];
};

struct statetrans {     /* from to configurations with associated probability */
        double                  prob;
        struct transition       confg;
};

struct block {          /* block of state transitions for the hashh table */
        struct  block           *next;
        struct  statetrans      trans[AVGLEN];
};


/* configurations for minor configurations */
struct  statetrans      minorpr[MINORCONFIGURATIONS];

/* hash table */
struct  block           *hashtab[HASHSIZE];

/* state transition for probabilities for a single processor with maximum
 * queue length of qlen
 */
```

```
double          ps[MAXQLENGTH][MAXQLENGTH];

/* state transition for probabilities for a single processor with maximum
 * queue length of qlen - 1
 */
double          psmin[MAXQLENGTH-1][MAXQLENGTH-1];

/* table of configurations for a network of nstat processors each with
 * maximum queue length qlen
 */
int             configtab[MAJORCONFIGURATIONS][MAXQLENGTH];

/* tables of source and destination configurations for the minor case
 * (maximum queue length qlen and 2 processors
 */
int             minorconfstab[MINORCONFIGURATIONS][MAXQLENGTH];
int             minorconfdtab[MINORCONFIGURATIONS][MAXQLENGTH];

int     qlen;           /* queue limit */
int     nstat;          /* number of processors (stations */
double  interval;       /* load balance interval */
double  arrate;         /* arrival rate */
double  deprate;        /* service rate */


/*********************** function definitions */

struct block    *getfree();
double          prtrn();
double          prtrnmin();
double          bprtrn();
double          lookpr();
double          lookminpr();
double          findconf();
double          calpr();
double          transpr();
double          balance();
double          looktab();
double          intab();
double          netresp();
char            *malloc();


/******************* IMSL stuff */

#define NUCONFGS        165     /* number of configurations */

/* IMSL error codes */
#define NOTACC          34      /* not accurate */
#define SINGULAR        129     /* matrix is singular */
#define ILLCOND         131     /* matrix is ill conditioned */

/* matrix containing state transitions probabilities */
double  a[NUCONFGS][NUCONFGS];

/* inverse of a, used to find steady state probabilities */
double  ainv[NUCONFGS][NUCONFGS];
```

```
/*      This section of code contains the main program for the
 *      state transition matrix solver.  The routines called directly
 *      by the main program for computation are also included.
 *
 *      The values for the state transition matrix are calculated
 *      first, then the steady state probability values are calculated.
 *      These are used to determine the average response time.
 *
 *      INPUT:  read from standard input.  The input format is described
 *              before init().
 *
 *      OUTPUT: written to standard output.
 */
main()
{
        /* initialize parameters */
        init();

        /* generate major and minor configurations and set the minor
         * probabilities for computations
         */
        confggen();
        minorgen();
        setminorpr();

        /* compute state transition probabilities */
        statetrans();

        printf("Calling imsl ...\n");
        fflush(stdout);
        /* compute steady state probabilities */
        steadystate();
        printf("... back from imsl\n");
        fflush(stdout);

        /* compute average response time */
        computeavg();
}
/*
 *      steadystate()   find the steady state probabilities by inverting
 *      the matrix (I - state transition matrix) after replacing the last
 *      column with 1's.  The last row of the resultant matrix will be
 *      the steady state problablitiy vector.
 *      NOTE: rows and columns are transposed due to IMSL conventions
 *
 *      globals used    a
 *                      ainv
 */
steadystate()
{
        FILE    *errorf;        /* pointer to file containing error messages */
        double  sum;
        int     row;
        int     col;
        int     n = NUCONFGS;
        int     ia = NUCONFGS;
        int     idgt= 0;
	double	wkarea[NUCONFGS * NUCONFGS + 3 * NUCONFGS];
	int	ier;


	/* compute I - a */
	for (row = 0;row<NUCONFGS;row++){
		for (col = 0;col<NUCONFGS;col++){
			if (row == col){
				a[col][row] = 1.0 - a[col][row];
			}else{
				a[col][row] = 0.0 - a[col][row];
			}
		}
	}

	/* set last column of resultant matrix to 1's */
	col = NUCONFGS - 1.0;
	for (row = 0;row < NUCONFGS;row++){
		a[col][row] = 1.0;
	}

	/* compute the inverse of a */
	linv2f_(&a[0][0],&n,&ia,&ainv[0][0],&idgt,&wkarea[0],&ier);

	/* check for errors */
	switch(ier){
		case SINGULAR:
			errorf = fopen("error\0","w");
			fprintf(errorf,"MATRIX IS SINGULAR\n");
			break;
		case ILLCOND:
			errorf = fopen("error\0","w");
			fprintf(errorf,"NOT INVERTABLE\n");
			break;
		case NOTACC:
			errorf = fopen("error\0","w");
			fprintf(errorf,"NOT ACCURATE TO DEGREE REQUESTED\n");
			break;
		default:
			/* everything is okay */
			break;
	}

	/* the last row (col) of R (ainv) is all we're interested in */
	row = NUCONFGS - 1;
	sum = 0.0;
	for (col=0;col<NUCONFGS;col++){
		printf("%g\t;number %d\n",ainv[col][row],col);
		sum += ainv[col][row];
	}
	printf("%g\t;sum of the components of w\n",sum);
	fflush(stdout);
}
/*	statetrans()	compute state transition matrix values
 *
 *	Consider all possible source and destination
 *	configurations, check if load balancing is feasible, and
 *	compute the state transition probability appropriately
 *
 *      no parameters passed
 *
 *      globals used    configtab
 *                      qlen            int     queue limit
 */
statetrans()
{
        struct transition cconfg;       /* current configuration */
        int source;                     /* index of current source in
                                           the configuration table */
        int     bal=0;

        for (source = 0;configtab[source][0] != ENDF;source++){
                concpy(cconfg.from,configtab[source]);
                if (canbalance(cconfg.from)){
                        bal ++;
                        balancetrans(&cconfg,source);
                }else{
                        unbalancetrans(&cconfg,source);
                }
        }
        printf("Number of rows balanced: %d\n",bal);
}
/*      setminorpr()    compute state transitions for minor
 *                      configurations
 *
 *      no parameters passed
 *
 *      globals used    minconfstab
 *                      minconfdtab
 *                      minorpr
 *                      qlen            int     queue limit
 */
setminorpr()
{
        int.    i=0;    /* index to table of minor config transition */
        int     s,d;    /* index to source and destination tables for
                         * minor configurations */

        for(s=0;minorconfstab[s][0] != ENDF;s++){
                for(d=0;minorconfdtab[d][0] != ENDF;d++){
                        concpy(minorpr[i].confg.from,minorconfstab[s]);
                        concpy(minorpr[i].confg.to,minorconfdtab[d]);
                        minorpr[i].prob = prtrnmin(&minorpr[i].confg);
                        i++;
                }
        }
}
/*      prtrnmin()      compute transition probabilities for the minor
 *                      configurations
 *
 *      parameter       confg - transition for a queue length of
 *                              qlen and two processors
 *
 *      globals used    psmin
 *                      qlen    int     queue limit
 */
```

```
double  prtrnmin(confg)
struct  transition      *confg;
{
        int loso,lode;  /* low load source and destination */
        int hiso,hide;  /* high load source and destination */
        double pr;

        for (loso = 0;confg->from[loso] == 0;loso++);
        for (hiso = qlen;confg->from[hiso] == 0;hiso--);

        if (confg->to[0] == 1){

                /* high loaded processor has forced transition */
                hide = 0;
                for(lode = 1;confg->to[lode] == 0;lode++);
                pr = psmin[hiso][hide] * psmin[loso][lode-1];

        }else if (confg->to[qlen] == 1){

                /* low loaded processor has forced transition */
                lode = qlen;
                for (hide = 0;confg->to[hide] == 0;hide++);
                pr = psmin[hiso][hide] * psmin[loso][lode-1];

        }else{

                /* just your average case */
                switch (confg->to[1]) {
                        case 0: pr = psmin[hiso][qlen - 1] * psmin[loso][1];
                                break;
                        case 1: pr = psmin[hiso][1] * psmin[loso][1] +
                                        psmin[hiso][qlen - 1] * psmin[loso][0];
                                break;
                        case 2: pr = psmin[hiso][1] * psmin[loso][0];
                                break;
                        default:
                        fprintf(stderr,"%d why are you here ?\n",confg->to[1]);
                                break;
                }
        }
        return(pr);
}

/*      init()  Read the queue length and number of stations from stdin
 *
 *      INPUT : read from standard input
 *
 *              (1) queue limit
 *              (2) number of stations
 *              (3) arrival rate
 *              (4) departure rate
 *              (5) load balance interval
 *              (6) state transition matrix for queue limit
 *              (7) state transition matrix for (queue limit - 1)
 *
 *      globals         nstat           integer number of processors in network
 *                      qlen            integer maximum qlength of processors
 *                      arrate          double  arrival rate
 *                       deprate         double  service rate
 *                       interval        double  load balance interval
 */
init()
{
        /* get the queue length */
        scanf("%d",&qlen); lnf(stdin);

        /* get the number of stations */
        scanf("%d",&nstat); lnf(stdin);

        /* get arrival rate */
        scanf("%f",&arrate); lnf(stdin);

        /* get departure rate */
        scanf("%f",&deprate); lnf(stdin);

        /* get load balance interval */
        scanf("%f",&interval); lnf(stdin);

        /* echo input data */
        printf("%d\t;Number of processors\n",nstat);
        printf("%d\t;Queue Limit\n",qlen);
        printf("%g\t;Arrival Rate\n",arrate);
        printf("%g\t;Service Rate\n",deprate);
        printf("%g\t;Load balance interval\n",interval);
        fflush(stdout);

        /* read state transition matrices */
        readps();
}
/*      minorgen()      generate the source and destination configurations for
 *                      the maximum and minumum loaded processors
 *
 *      no parameters passed
 *
 *      globals accessed        minorconfstab   minor configuration source
 *                                              table
 *                              minorconfdtab   minor configuration destination
 *                                              table
 *                              both tables are arrays of configurations
 *
 *                              qlen    int     queue length
 *                              nstat   int     number of stations
 */
minorgen()
{
        int     min,            /* max/min source and destination */
                max;
        int     tabind;         /* index into tables */
        int     i,j;
        int     inc;            /* flag to indicate if table entry is unique */

        /* determine source configurations */
        tabind = 0;
        for (min=0;min<qlen-1;min++){
                for(max=min+1;max<qlen;max++){
                        minorconfstab[tabind][min]++;
                        minorconfstab[tabind][max]++;

                        /* check if configurations is unique */
                        inc = TRUE;
                        for (j=0;j<tabind;j++){
                                if(eqconf(minorconfstab[j],
                                                minorconfstab[tabind])==TRUE){
                                        inc = FALSE;
                                }
                        }
                        if (inc){       /* store if config unique */
                                tabind++;

                        }else{          /* clear location otherwise */
                                for(i=0;i<qlen+1;i++) {
                                        minorconfstab[tabind][i] = 0;
                                }
                        }
                }
        }
        /* set end of table flag */
        for(j=0;j<qlen+1;j++){
                minorconfstab[tabind][j] = ENDF;
        }

        /* determine destination configurations */
        tabind = 0;
        for (min=1;min<qlen+1;min++){
                for(max=0;max<qlen;max++){
                        minorconfdtab[tabind][min]++;
                        minorconfdtab[tabind][max]++;

                        /* check if configurations is unique */
                        inc = TRUE;
                        for (j=0;j<tabind;j++){
                                if (eqconf(minorconfdtab[j],
                                        minorconfdtab[tabind]) == TRUE){
                                        inc = FALSE;
                                }
                        }

                        if (inc){       /* store configuration if unique */
                                tabind++;

                        }else{          /* clear location otherwise */
                                for(i=0;i<qlen+1;i++){
                                        minorconfdtab[tabind][i] = 0;
                                }
                        }
                }
        }

        /* set end of table flag */
        for(j=0;j<qlen+1;j++){
                minorconfdtab[tabind][j] = ENDF;
        }
}
/*      confggen()      generate configurations for a network with
```

```
 *                     'nstat' processors and a maximum queue length
 *                     of 'qlen'.
 *
 *      globals accessed        configtab       table of configurations
 *                              nstat   int     number of processors
 *                              qlen    int     maximum qlength of processors
 */
confggen()
{
        int     sum;                    /* sum of number of stations */
        int     tabin = 0;              /* index into configuratins table */
        int     digit[MAXQLENGTH];      /* temporary configuration storage */
        int     l;                      /* index into temporary configuration */

        /* clear temporary configurations */
        for (l = 0;l < qlen+1;l++) digit[l] = 0;

        /* generate configurations in order until last location equals
         * nstat (highest number
         */
        while (digit[qlen] != nstat){

                /* count */
                digit[0]++;
                for (l=0;l<qlen;l++){
                        if (digit[l] == nstat + 1){
                                digit[l+1]++;
                                digit[l]=0;
                        }
                }

                /* check if value number (# stations equals nstat) */
                sum = 0;
                for (l=0;l<qlen+1;l++){
                        sum += digit[l];
                }

                /* store configuration if so */
                if (sum == nstat){
                        concpy(configtab[tabin++],digit);
                }
        }

        /* store flag for end of configurations table */
        for (l=0;l<qlen+1;l++){
                digit[l]=ENDF;
        }
        concpy(configtab[tabin],digit);
}
/*      readps()        Function to read in the state transition
 *                      matricies for the single processor case.
 *
 *      globals accessed        ps,psmin        state transition matricies
 *                              qlen            int     queue length
 */
readps()
{
        int     i;
        int     j;

        for(i=0;i<qlen+1;i++){
                for(j=0;j<qlen+1;j++){
                        scanf("%f",&ps[i][j]);
                }
        }
        for(i=0;i<qlen;i++){
                for(j=0;j<qlen;j++){
                        scanf("%f",&psmin[i][j]);
                }
        }
}

/*      balancetrans()  compute state transition probabilities for
 *                      a configuration that can be load balanced
 *
 *      parameters      confg - from to processor locations
 *                      source - index of source configuration in
 *                              confgtab
 *
 *      globals         a
 */
balancetrans(confg,source)
struct  transition      *confg;
int     source;
{
        int     destination;            /* destination index to configtab */

        /* confg minus most and least heavily loaded processors */
        struct  transition      majconf;

        /* position of most and least heavily loaded processors */
        struct  transition      minconf;

        balanceinit(&majconf,&minconf,confg);
        for (destination=0;configtab[destination][0] != ENDF;destination++){
                concpy(confg->to,configtab[destination]);
                a[destination][source] = bprtrn(&majconf,&minconf,confg);
        }
}

/*      balanceinit()   initializes major and minor from configuations
 *                      for the current row.  The major configuration
 *                      is the current from location minus the most and
 *                      least loaded processors.  The minor configuration
 *                      is the location of the least and most heavily loaded
 *                      processors in a configuration with maximum queue
 *                      length of qlen
 *
 *      parameters      majconfg - major configuration
 *                      minconfg - minor configuration
 *                      confg - current configuration
 *
 *      globals         none accessed
 *
 */
balanceinit(majconf,minconf,confg)
```

```
struct  transition      *majconf;
struct  transition      *minconf;
struct  transition      *confg;
{
        int     u,l;    /* indices of upper and lower processors */
        int     i;

        /* find most and least loaded processors */
        for (u = qlen; confg->from[u] == 0; u--);
        for (l = 0; confg->from[l] == 0; l++);

        /* compute starting confg for major configuration */
        concpy(majconf->from,confg->from);
        majconf->from[u]--;    majconf->from[l]--;

        /* compute starting confg for minor configuration */
        for (i=0;i<qlen+1;i++){
                minconf->from[i] = 0;
        }
        minconf->from[u-1]++; minconf->from[l]++;
}
/*      bprtrn()        probability of transition for load balance case
 *                      determine the method of finding the probability
 *                      (1) - Degenerate case, pr = 1
 *                      (2) - Simple transition, look it up
 *                      (3) - Default calculate it
 *
 *      parameters      majconfg - major configuration
 *                      minconfg - minor configuration
 *                      confg - configuration
 */
double bprtrn(majconfg,minconfg,confg)
struct  transition      *confg;
struct  transition      *majconfg;
struct  transition      *minconfg;
{
        int i;
        int sums=0;     /* sum of source configurations */
        int sumd=0;     /* sum of destination configurations */
        double pr;      /* transition probability */

        for (i=0;i<qlen+1;i++){
                sums += confg->from[i];
                sumd += confg->to[i];
        }

        /* check to see if things are okay */
        if (sums != sumd){
                fprintf(stderr,"unequal numbers of processors");
                exit(-2);

        /* zero state to zero state transition */
        }else if (sums == 0){
                return(1.0);

        /* reduced to simple transition */
        }else if (sums == 1){
                pr = lookpr(confg);
 *
 *      parameters              confg - from to processor locations
 *                              source - index of from configuration
 *                                      in configtab
 *
 *      globals                 a - the state transition matrix
 */
unbalancetrans(confg,source)
struct  transition      *confg;
int     source;
{
        int     destination;    /* destination index to configtab */

        for (destination=0;configtab[destination][0] != ENDF;destination++){
                concpy(confg->to,configtab[destination]);
                a[destination][source] = prtrn(confg);
        }
}
/*      prtrn()         probability of transition
 *                      determine the method of finding the probability
 *                      (1) - Degenerate case, pr = 1
 *                      (2) - Simple transition, look it up
 *                      (3) - Default calculate it
 *
 *      parameter       confg - from to processor locations
 *
 *      globals         none accessed
 */
double prtrn(confg)
struct transition *confg;
{
        int i;
        int sums=0;     /* sum of source configurations */
        int sumd=0;     /* sum of destination configurations */
        double pr;      /* transition probability */

        for (i=0;i<qlen+1;i++){
                sums += confg->from[i];
                sumd += confg->to[i];
        }

        /* check if things are okay */
        if (sums != sumd){
                fprintf(stderr,"unequal number of processors\n");
                exit(-3);

        /* zero state to zero state transition */
        }else if (sums == 0){
                return(1.0);

        /* reduced to simple transition */
        }else if (sums == 1){
                pr = lookpr(confg);
                return(pr);

        /* calculate it */
        }else{
                pr = transpr(confg);
                return(pr);
        }
        return(-1);
}

/*
 *      transpr()       calculates  the transition probability for the
 *                      configuration
 *
 *      parameters      confg - from to processor locations
 *
 *      globals         ps - table of simple transitions
 */
double transpr(confg)
struct transition *confg;
{
        double  pr;     /* state transition probability */
        int     i,j;

        if ((pr = intab(confg)) != -1){
                return(pr);
        }
        pr = 0;

        /* find the first non-zero element in the source confg */
        for (i=0;confg->from[i] == 0;i++);

        /* set source for next state */
        confg->from[i]--;

        /* go thru all terms */
        for (j=0;j<qlen + 1;j++){
                if (confg->to[j] != 0){
                        confg->to[j]--;
                        pr += ps[i][j] * prtrn(confg);
                        confg->to[j]++;
                }
        }

        /* set source confg back */
        confg->from[i]++;

        /* put state transition probability in hashtable */
        store(confg,pr);

        return(pr);
}

/*      hash            compute hash value from current configuration
 *
 *      parameters      confg - from to locations of processors
 *
 *      globals         none
 */
hash(confg)     /* form hash value from configuration */
struct transition *confg;
{
        int i;
        int hashval = 0;

        for (i=0;i<qlen + 1;i++){
                hashval = (hashval * 10) + confg->from[i];
        }

        for (i=0;i<qlen + 1;i++){
                hashval = (hashval * 10) + confg->to[i];
        }

        return(hashval % HASHSIZE);
}

/*
 *      getfree()       allocate a block of state transitions
 *                      returns a pointer to the block
 *                      exits if there is no room
 *
 *      no parameters passed
 *
 *      no globals accessed
 */
struct block *
getfree()
{
        struct block *p;

        if ((p = (struct block *) malloc(sizeof(struct block)))
                        == (struct block *)0){
                fprintf(stderr,"malloc - out of room\n");
                exit(1);
        }
        return (p);
}
/*      intab()         look for current configuration in the hash
 *                      table
 *
 *      parameter       confg - processor from to locations
 *
 *      globals         hashtab - table of calculated state transition
 *                                probabilities
 */
double intab(confg)
struct  transition      *confg;
{
        /* state transition pointer into hashtable */
        struct  block   *stptr;
        int     i;      /* index into current block */

        stptr = hashtab[hash(confg)];
        while(stptr){
                for (i=0;i<AVGLEN;i++){
                        /* look through current block until config is
                         * found, or you come to last config
                         */
                        if ((stptr->trans[i].prob == -1) ||
                                (eqconfig(&stptr->trans[i].confg,confg))){
                                return(stptr->trans[i].prob);
                        }
                }
                stptr = stptr->next;
        }
        return(-1);
}

/*      store()         store state transition probability in the
 *                      hash table
 *
 *      parameters      confg - from to locations of processors
 *                      pr - state transition probability
 *
 *      globals         hashtab - table containing calculated values
 */
store(confg,pr)
struct  transition      *confg;
double  pr;
{
        /* state transition pointer (to current block) */
        struct  block   *stptr;
        int i=0;        /* index into current block of the hash table */

        /* store result for future reference */
        stptr = hashtab[hash(confg)];
        if (!stptr){
                /* allocate first block */
                hashtab[hash(confg)] = getfree();
                stptr = hashtab[hash(confg)];
                stptr->next = 0;
        }else{
                /* skip over to last block */
                while (stptr->next){
                        stptr = stptr->next;
                }

                /* find next free space in the current block */
                while ((i<AVGLEN) && (stptr->trans[i].prob != -1)){
                        i++;
                }

                /* if last item is in the last space for this block,
                 * allocate another block
                 */
                if (i == AVGLEN){
                        stptr->next = getfree();
                        stptr = stptr->next;
                        stptr->next = 0;
                        i=0;
                }
        }

        /* store info, and set a flag for last item */
        stptr->trans[i].prob = pr;
        stptr->trans[i].confg = *confg;
        if ((++i) < AVGLEN) {
                stptr->trans[i].prob = -1;
        }
```

```
}

/*
 *      computeavg()    compute average response time for the network
 *                      This is done by determining the amount of time
 *                      each job will spend in the queue and server
 *                      plus load balance time, then taking a weighted
 *                      average of these
 *                      times
 *
 *      globals         configtab       table of network configurations
 *                      qlen            int     queue length
 *                      ainv                    matrix of steady state
 *                                              trans probabilities
 *                      interval        double  load balnace interval
 */
computeavg()
{
        double  totresp;
        int     confno;

        /* for all configurations, compute the response times, and keep
         * a weighted sum
         */
        totresp = 0.0;
        confno = 0;

        while (configtab[confno][0] != ENDF){
                totresp += ainv[confno][NUCONFGS-1]
                                 * netresp(configtab[confno]);
                if (canbalance(configtab[confno]))
                        totresp += ainv[confno][NUCONFGS-1] * interval;
                confno++;
        }

        printf("%20.15f\t;average response time",totresp);
}
/*
 *      netresp()       Compute the average response time for a network
 *                      of processors for a given configuration of jobs.
 *
 *      parameters      config          number of processors at each
 *                                      queue length
 *
 *      globals accessed        qlen    int     maximum queue length
 */
double netresp(config)
int     config[];
{
        int     j,i;
        double  time = 0.0;
        int     nojobs = 0;

        for(j=0;j<qlen+1;j++){
                nojobs += j * config[j];
                for(i=1;i<j+1;i++)
                        time += (i / deprate) * config[j];
        }
        if (nojobs != 0 )
                return(time/nojobs);
        else
                return(0.0);
}

/*      lookpr()        looks at the from to configurations to find the
 *                      probability of the simple state transition
 *
 *      parameters      confg   the configuration to be looked up
 *
 *      globals accessed        ps      transitions of single processor
 *
 *      returns the probability of the simple state transition
 */
double lookpr(confg)
struct transition *confg;
{
        int i,j;

        for (i=0;confg->from[i] != 1;i++);
        for (j=0;confg->to[j] != 1;j++);

        return(ps[i][j]);
}
/*
 *      canbalance()    determines if a load balancing would make a difference.
 *                      returns TRUE if so, FALSE otherwise
 *
 *      parameter       state - current from config
 *
 *      globals         none accessed
 */
canbalance(state)
int state[];
{
        int     u,l;    /* upper and lower processor queue length */
        double  rtime,ltime;    /* local and remote processing time */

        /* find most and least loaded processors */
        for (u=qlen;state[u] == 0;u--);
        for (l=0;state[l] == 0;l++);

        if ((u-l) > 1){
                ltime = u * deprate;
                rtime = l * deprate + interval;
                return(rtime < ltime ? TRUE : FALSE);
        }else{
                return(FALSE);
        }
}
/*      concpy()        copies cons to cond
 *
 *      parameters      cons - source configuration
 *                      cond - destination configuration
 */
concpy(cond,cons)
int cond[];
int cons[];
{
        int i;          /* index into configuration */

        for (i=0;i<qlen+1;i++){
                cond[i] = cons[i];
        }
}
/*      eqconfig()      compare con1 to con2 and return true if equal
 *
 *      paramters       con1 - configuration to be compared
 *                      con2 - configuration to be compared
 *
 *      globals         none accessed
 */
eqconfig(con1,con2)
struct transition *con1,*con2;
{
        int flag=TRUE;
        int i;

        for (i=0;i<qlen+1;i++){
                if ((con1->from[i] != con2->from[i])
                                || (con1->to[i] != con2->to[i])){
                        flag = FALSE;
                }
        }
        return(flag);
}
/*      eqconf()        compare con1 to con2 and return true if equal
 *
 *      paramters       con1 - source or dest configuration to be compared
 *                      con2 - source or dest configuration to be compared
 *
 *      globals         qlen    int     queue limit
 */
eqconf(con1,con2)
int     *con1[],
        *con2[];
{
        int flag=TRUE;
        int i;

        for (i=0;i<qlen+1;i++){
                if (con1[i] != con2[i]){
                        flag = FALSE;
                }
        }
        return(flag);
}
/*      lookminpr()     look up probability for minor configuration in
 *                      table of minor configurtation probabilities
 *
 *      parameter       confg - minor configuration
 *
 *      globals         minorpr - table of minor configuration state
 *                              transitions
 */
```

```
double lookminpr(confg)
struct  transition      *confg;
{
        int     i;      /* index into minorpr */

        for (i=0;(!eqconfig(&minorpr[i].confg,confg));i++);
        return(minorpr[i].prob);
}
```

Appendix B. Simulation Programs

```
/*
 *      This is a simulation for a a group of identical processors
 *      connected by a CSMA/CD network.  The network is used for load
 *      balance operations.
 *
 *      The SMPL library is used to handle queue management and
 *      statistics
 *      gathering.  The following routines are used:
 *
 *      *** cause(&event,&job)
 *      cause the next event to occur returning the event type and
 *      the job
 *      associated with that event
 *
 *      *** reserve(facility,job,priority)
 *      reserve a facility for the job at the priority specified.
 *      If the job is queued a 1 is returned, otherwise a 0 is returned.
 *
 *      *** release(facility,job)
 *      release the facility previously reserved or preempted by the job
 *
 *      *** schedule(event,wait,job)
 *      schedule the event after a time interval wait.
 *      The event corresponds to the job passed.
 *
 *      *** time()
 *      returns the current simulation time
 *
 *      *** uniform(min,max)
 *      returns a uniformly distributed random variable
 *
 *      *** expntl(mean)
 *      returns an exponentially distributed random variable
 *
 *      *** qlength(facility)
 *      returns the current queue length at the facility
 *
 *      *** reset(flag)
 *      resets the statistics gathering.  If the flag is true,
 *      a report is generated.
 *
 *      INPUT:  read from standard input.  The input format is described
 *              before the procedure 'init()'
 *
 *      OUTPUT: written to standard output
 */

/*
 *      define some stuff
 */
#define         TRUE            1
#define         FALSE           0

#define         IDLE            0
#define         BUSY            1

#define         JOBSAV          10
#define         ZEROINT         500.0
```

```
/*
 *      classes of facilties
 */
#define         PRODUCER                1       /* job creator */
#define         SERVER                  2       /* service point */
#define         EXIT_MONITOR            3       /* end of service */
#define         BUS                     4       /* network connection */

/*
 *      priorities
 */
#define         DFLT            0.0     /* default priority */
#define         SEND_PRI        0.0     /* send priority */
#define         RETURN_PRI      10.0    /* return priority */

/*
 *      types of events
 */
#define         RESERVE         1       /* reserve a facility
                                         * for a period */
#define         RELEASE         2       /* release a previously
                                         * reserved facility */

/*
 *      types of distributions
 */
#define         UNIFORM         1
#define         EXPO            2

/*
 *      constants for array sizes
 */
#define         MAXPES          10

/*
 *      macro to skip to end of an input line
 */
#define         lnf(f)          {while (getc(f) != '\n');}

#include <stdio.h>
#include <math.h>

/*
 *      structure to hold a job description
 */
struct job{
        int     server_type;    /* class of current server  */
        int     source;         /* pe# where job originated */
        int     dest;           /* load balance destination */
        int     loadb;          /* flag to indicate if job load bal'd */
        int     location;       /* current location of job */
        double  jobstart;       /* time job started in the system */
        double  lb_time;        /* time on the bus */
};

/*
 *      structure to hold a distribution
 */
struct dist {
        double  min;            /* minimum value */
        double  max;            /* maximum value */
        int     type;           /* type of distribution */
};

/*
 *      structure to describe a processor
 */
struct processor{
        int     producer;       /* lamda producer */
        int     server;         /* processor server */
        int     exit_monitor;   /* facility to count exiting jobs */
};

/*
 *      external and forward procedure declarations
 */
double          distribution(), expntl(), uniform();
double          time();
char            *calloc();
struct  job     *getjob();
int             canbalance();
double          avg();

/*
 *      global variable declarations
 */

/*      facilities and facility parameters      */
struct  processor       pes[MAXPES];    /* processors */
struct  dist            d_ar;           /* arrival rate distribution */
struct  dist            d_se;           /* service rate distribution */
struct  dist            d_bu;           /* bus service rate dist */
int     bus;                            /* CSMA/CD Network */

int     qres[MAXPES];           /* queue locations resrvd for servers */
int     num_pes;                /* number of processing elements */
int     maxql;                  /* maximum queue length */
int     num_busses;             /* number of busses */

int     num_batches;            /* number of batches to run */
double  interval;               /* duration of a batch */

int     loadbal;                /* number of load balanced jobs */
int     jobslost;               /* jobs lost to queue limit violation */
int     jobcount;               /* number of jobs completed */
double  jobtime;                /* time it took to complete them */
double  starttime;              /* the start time for this batch */

double  sent[JOBSAV];           /* total time for jobs sent */
double  retrn[JOBSAV];          /* total time for jobs returned */
int     send_p;                 /* number of jobs sent */
int     return_p;               /* number of jobs returned */

int     trace_flag;
/*
```

```
 *      sim() -- the main simulation routine.  This is called
 *      from the "main()" in the SMPL library after it has initialized
 *      its data.
 */
sim()
{
        struct  job     *j;

        int     event;
        int     i;
        double  zerotime;
        double  cur_int;

        /* initialize */
        init();

        /* run batches for comparison */
        for (i=0;i<num_batches + 1;i++){
                jobtime = loadbal = jobslost = jobcount = 0;
                starttime = time();
                zerotime = time();

                if (i==0) {
                        cur_int = 50000.0;
                }else{
                        cur_int = interval;
                }

                /* process events through the simulation interval */
                while ((time() - starttime) < cur_int) {

                        /* zero an entry to keep average current */
                        if ((time() - zerotime) > ZEROINT){
                                sent[send_p] = 0;
                                if(++send_p == JOBSAV) send_p = 0;
                                retrn[return_p] = 0;
                                if(++return_p == JOBSAV) return_p = 0;
                                zerotime = time();
                        }


                        cause(&event,&j);

                        switch(event){

                        case RESERVE:
                                reserve_facility(j);
                                break;

                        case RELEASE:
                                release_facility(j);
                                break;

                        default:
                                fprintf(stderr,"Unknown event\n");
                                exit(-1);
                                break;
                        }
                }
                my_report();
        }
        reset(TRUE,0);
}
/*
 * report some results
 */
my_report()
{
        printf("\n\n\n\n\nSummary of results\n\n");
        printf("Load Balance time : %f\n",2*d_bu.min);
        printf("Elapsed Time       = %f\n",time()-starttime);
        printf("Jobs Completed     = %d\n",jobcount);
        printf("Jobs Lost          = %d\n",jobslost);
        printf("Jobs Load Balanced = %d\n",loadbal);
        printf("Total Job Time     = %f\n",jobtime);
        printf("Job Throughput     = %f\n",jobcount/(time()-starttime));
        printf("Average Job Delay  = %f\n",jobtime/jobcount);
        printf("Average time to send   = %f\n",avg(sent));
        printf("Average time to return = %f\n",avg(retrn));
        printf("Average load balance time = %f\n",
                                        (avg(sent) + avg(retrn)));
        fflush(stdout);
}
/*
 *      Init -  initialize all global network variables from input
 *              and echo results and set initial jobs at all processors
 *
 *      Input Format:
 *
 *        (1)  Model name
 *        (2)  Number of Processing Elements
 *        (3)  Max Queue Length Allowed
 *        (4)  Number of busses
 *        (4)  PE Service time distribution & parameters
 *        (5)  Arrival time distribution type & params
 *        (6)  Bus service time distribution type & params
 *        (7)  Number of batches to run
 *        (8)  Duration of a batch
 */
init()
{
        /* model name */
        char    model_name[256];

        /* facility names */
        char creator_name[9];
        char servicer_name[9];
        char exit_name[9];

        int     i;
        struct  job     *j;

        if (checkcmd('t')) {    /* set trace flag and activate SMPL  */
           trace_flag = TRUE;   /* trace features                    */
           trace(2);
        } else {
```

```
   trace_flag = FALSE;
}

/* get model name */
scanf("%s",model_name); lnf(stdin);

smpl(model_name);

/* get number of processing elements */
scanf("%d",&num_pes); lnf(stdin);

/* get maximum queue length */
scanf("%d",&maxql); lnf(stdin);

/* get number of busses */
scanf("%d",&num_busses); lnf(stdin);

/* get PE service time distribution */
scanf("%d %f %f",&d_se.type,&d_se.min,&d_se.max); lnf(stdin);

/* get arrival time distribution */
scanf("%d %f %f",&d_ar.type,&d_ar.min,&d_ar.max); lnf(stdin);

/* get bus service time distribution */
scanf("%d %f %f",&d_bu.type,&d_bu.min,&d_bu.max); lnf(stdin);

/* get number of batches */
scanf("%d",&num_batches); lnf(stdin);

/* get duration of a batch */
scanf("%f",&interval); lnf(stdin);

/* set up model facilities */
for(i=0;i<num_pes;i++){

        sprintf(creator_name,   "creatr %d",i);
        sprintf(servicer_name,  "server %d",i);
        sprintf(exit_name,      "exit   %d",i);

        pes[i].producer  = facility(creator_name,1);
        pes[i].server    = facility(servicer_name,1);
        pes[i].exit_monitor = facility(exit_name,1);
}

bus = facility("bus",num_busses);

/*
 * Initialize some jobs to start at each PE
 */
for(i = 0; i < num_pes; i++) {
        j = getjob();
        j->server_type = PRODUCER;
        j->source = i;
        j->dest = -1;
        j->loadb = FALSE;
        j->location = i;
        schedule (RESERVE, 0.0, j);
}
```

```
        /*
         * Report the input parameters
         */
        printf("\nParameter Dump\n\n\n");
        printf("Model Name            - %s\n",model_name);
        printf("Number of Processors - %d\n",num_pes);
        printf("Queue Limit           - %d\n",maxql);
        printf("Number of Batches     - %d\n",num_batches);
        printf("Duration of a Batch  - %g\n",interval);
        printf("Number of Busses      - %d\n",num_busses);

        printf("Processing Element Arrival Time Parameters \n");
        dumpdist(d_ar);
        printf("Processing Element Service Time Parameters \n");
        dumpdist(d_se);
        printf("Bus Service Time Paramters \n");
        dumpdist(d_bu);
        fflush(stdout);
}
/*      reserve_facility - reserve the facility indicated by the passed
 *      job
 *
 *      parameters      j       pointer to structure of type job
 */
reserve_facility(j)
struct  job     *j;
{

        int     minloc;         /* location of minimally loaded proc */

        switch(j->server_type){

        case PRODUCER:
                /* schedule job creation */
                if (reserve(pes[j->location].producer,j,DFLT) == IDLE)
                        schedule(RELEASE,distribution(d_ar),j);
                return;

        case SERVER:

                if (j->loadb == TRUE){
                        /* update statistics */
                        sent[send_p] = time() - j->lb_time;
                        if(++send_p == JOBSAV) send_p = 0;
                        if (reserve(pes[j->location].server,j,DFLT)
                                                        == IDLE){
                                qres[j->location]--;
                                schedule(RELEASE,distribution(d_se),j);
                        }

                } else if(qlength(pes[j->location].server)
                                >= (maxql-qres[j->location])){
                        /* queue length violation */
                        cfree(j);
                        jobslost++;

                } else if (qlength(pes[j->location].server)
                                        && canbalance(j,&minloc)){
                        /* check if job should be balanced*/
                        loadbal++;
                        j->loadb = TRUE;
                        j->dest = minloc;
                        j->server_type = BUS;
                        j->location = j->dest;
                        j->lb_time = time();
                        qres[minloc]++;
                        schedule(RESERVE, 0.0, j);

                } else {
                        /* service job locally  */
                        if (reserve(pes[j->location].server,j,DFLT)
                                                        == IDLE)
                                schedule(RELEASE,distribution(d_se),j);
                }
                return;

        case EXIT_MONITOR:
                /* job is completed, terminate it       */
                if(j->loadb){
                        retrn[return_p] = time() - j->lb_time;
                        if (++return_p == JOBSAV) return_p = 0;
                }
                if(reserve(pes[j->location].exit_monitor,j,DFLT)==IDLE)
                        schedule(RELEASE,0.0,j);
                return;

        case BUS:
                if (j->location == j->dest){
                        /* send the job across the network */
                        if (reserve(bus,j,SEND_PRI) == IDLE){
                                schedule(RELEASE,distribution(d_bu),j);
                        }
                } else {
                        /* return the job across the network */
                        if (reserve(bus,j,RETURN_PRI) == IDLE){
                                schedule(RELEASE,distribution(d_bu),j);
                        }
                }
                return;

        default:
                fprintf(stderr,"Reserve of unknown facility\n");
                exit(-1);
                return;
        }
}
/*      release_facility(j) - release the facility indicated by the
 *      passed job
 *
 *      parameters      j       pointer to structure of type job
 */
release_facility(j)
struct  job     *j;
{
        struct  job     *j1;
        switch (j->server_type){

        case PRODUCER:
                /* produce job, and send the current job for service */
                release(pes[j->location].producer,j);
                j1 = getjob();
                j1->server_type = SERVER;
                j1->source = j->source;
                j1->dest = j->dest;
                j1->loadb = FALSE;
                j1->location = j->location;
                j1->jobstart = time();
                schedule(RESERVE,0.0,j1);
                schedule(RESERVE,0.0,j);
                return;

        case SERVER:
                /* service complete return job if load balanced,
                 * else release */
                release(pes[j->location].server,j);
                if (j->loadb) {
                        j->server_type = BUS;
                        j->location = j->source;
                        j->lb_time = time();
                } else {
                        j->server_type = EXIT_MONITOR;
                }
                schedule(RESERVE,0.0,j);
                return;

        case EXIT_MONITOR:
                /* free job */
                release(pes[j->location].exit_monitor,j);
                jobcount ++;
                jobtime += (time() - j->jobstart);
                cfree(j);
                return;

        case BUS:
                /* take job off bus and service or release it */
                release(bus,j);
                if (j->source == j->location) {
                        j->server_type = EXIT_MONITOR;
                } else {
                        j->server_type = SERVER;
                }
                schedule (RESERVE,0.0,j);
                return;

        default:
                fprintf(stderr,"Release of unknown facility\n");
                exit(-1);
                return;
        }
}
/*
 * getjob() -  get a job from free storage pool
 */
```

```
struct job *
getjob()
{
        struct job *j;

        if((j = (struct job *) calloc(1,sizeof(struct job))) ==
                                                (struct job *) NULL){
                fprintf(stderr,"calloc -- out of room\n");
                exit(1);
        }
        return(j);
}
/*
 *
 *      Dumpdist - echo the parameters of a distribution
 *
 */
dumpdist(dist)
struct dist dist;
{
        switch(dist.type) {

           case UNIFORM:
              printf("Distribution Uniform\n");
              printf("Min %5f Max %5f\n",dist.min,dist.max);
              break;

           case EXPO:
              printf("Distribution Negative Exponential\n");
              printf("Mean %5f\n",dist.min);
              break;

        }
}
/*
 *
 *      Distribution - return a pseudo-random variate from a
 *                     distribution specified by type
 *
 */
double distribution(dist)
struct dist dist;
{
        double val;

        switch(dist.type) {

           case UNIFORM:
             val = uniform(dist.min,dist.max);
             break;

           case EXPO:
             val = expntl(dist.min);
             break;

           default:
             printf("\nUnknown dist. type is function <distribution> %d\n",
                        dist.type);
        }
        return(val);
}
/*
 *      canbalance - determine if network configuration allows load
 *      balance operations to be effective. "minloc" returns the
 *      location  of the minimally loaded processor in the network.
 *
 *      returns TRUE if job should be load balanced
 *      returns FALSE otherwise
 *
 *      parameters      j       struct  job     current job
 *                      minloc  int             minimum location
 *
 *      globals         num_pes int             number of processors
 *
 */
int canbalance(j,minloc)
struct  job     *j;
int     *minloc;
{
        int     min,max;                /* max and min queue lengths */
        double  remtime;                /* remote execution time */
        double  loctime;                /* local execution time */
        int     i,temp;

        min = maxql + 1;
        max = -1;

        for (i=0;i<num_pes;i++){
                if ((temp = qlength(pes[i].server)) > max){
                        max =temp;
                }
                if ((temp + qres[i]) < min){
                        min = temp + qres[i];
                        *minloc = i;
                }
        }

        if ((qlength(pes[j->location].server) == max)
                                && ((max - min) >= 2 )){
                loctime = d_se.min * (max + 1);
                remtime = d_se.min * (min + 1);
                remtime += avg(sent) + avg(retrn);;
                return(((loctime - remtime) > (4 * d_bu.min))
                                                ? TRUE : FALSE);
        } else {
                return(FALSE);
        }
}
double avg(a)
double a[];
{
        double  sum=0;
        int     i=0;

        for (i==0;i<JOBSAV;i++) sum += a[i];
        return (sum/(double)(JOBSAV));

 }
```

```
CSMA/CD_NETWORK_LOAD_BALANCED_CASE
8                        ;number of processors
100                      ;queue limit
1                        ;number of busses
2  8.0                   ;service time distribution
2  10.0                  ;arrival time distribution
2  0.50                  ;bus service time
2                        ;number of batches
100000                   ;simulation interval
```

```
Parameter Dump


Model Name            - CSMA/CD_NETWORK_LOAD_BALANCED_CASE
Number of Processors - 8
Queue Limit           - 10
Number of Batches     - 2
Duration of a Batch   - 100000
Number of Busses      - 1
Processing Element Arrival Time Parameters
Distribution Negative Exponential
Mean 10.000000
Processing Element Service Time Parameters
Distribution Negative Exponential
Mean 8.000000
Bus Service Time Paramters
Distribution Negative Exponential
Mean 0.500000




Summary of results

Load Balance time : 1.000000
Elapsed Time        = 50000.152474
Jobs Completed      = 39699
Jobs Lost           = 246
Jobs Load Balanced = 5737
Total Job Time      = 1552413.369025
Job Throughput      = 0.793978
Average Job Delay  = 39.104596
Average time to send   = 2.737639
Average time to return = 0.870162
Average load balance time = 3.607801




Summary of results

Load Balance time : 1.000000
Elapsed Time        = 100000.318429
Jobs Completed      = 79665
Jobs Lost           = 339
Jobs Load Balanced = 12610
Total Job Time      = 3209946.336462
Job Throughput      = 0.796647
Average Job Delay  = 40.293056
Average time to send   = 5.024928
Average time to return = 0.423172
Average load balance time = 5.448100
```

```
Summary of results

Load Balance time : 1.000000
Elapsed Time       = 100000.138465
Jobs Completed     = 79807
Jobs Lost          = 435
Jobs Load Balanced = 12559
Total Job Time     = 3027668.011506
Job Throughput     = 0.798069
Average Job Delay  = 37.937374
Average time to send   = 11.916958
Average time to return = 0.607872
Average load balance time = 12.524830
```

```
          S M P L   S I M U L A T I O N   R E P O R T


MODEL: CSMA/CD_NETWORK_LOAD_BALANCED_CASE          TIME:      250000.609368
                                                   INTERVAL:  250000.609368
                                                   CPU-TIME:    1469.000000

                                          MEAN BUSY            QUEUE LENGTH                      OPERATION  COUNTS
FACILITY       COMP. RATE      UTIL.        PERIOD     VARIANCE    MEAN        MAX      RELEASE   PREEMPT   QUEUE

creatr 0        0.100308     0.999985     9.969173 100.440643  0.000000        0      25077       0          0
server 0        0.101080     0.807964     7.993334  64.780472  3.314244       10      25270       0      22301
exit   0        0.099680     0.000000     0.000000   0.000000  0.000000        0      24920       0          0
creatr 1        0.099108     0.999966    10.089686 101.973293  0.000000        0      24777       0          0
server 1        0.091220     0.730496     8.008091  63.156457  3.746070       10      22805       0      20031
exit   1        0.098616     0.000000     0.000000   0.000000  0.000000        0      24654       0          0
creatr 2        0.100380     0.999974     9.961913  97.615792  0.000000        0      25095       0          0
server 2        0.106772     0.855618     8.013521  65.615213  2.944891       10      26693       0      23248
exit   2        0.099900     0.000000     0.000000   0.000000  0.000000        0      24975       0          0
creatr 3        0.100424     0.999958     9.957384  97.909794  0.000000        0      25106       0          0
server 3        0.108180     0.861847     7.966803  63.174620  2.790980       10      27045       0      23294
exit   3        0.099932     0.000000     0.000000   0.000000  0.000000        0      24983       0          0
creatr 4        0.099624     0.999996    10.037729 101.076039  0.000000        0      24906       0          0
server 4        0.102844     0.820736     7.980413  63.163115  2.816773       10      25711       0      21807
exit   4        0.099164     0.000000     0.000000   0.000000  0.000000        0      24791       0          0
creatr 5        0.099916     0.999975    10.008185 101.448002  0.000000        0      24979       0          0
server 5        0.097632     0.781838     8.008025  64.966311  3.002953       10      24408       0      20772
exit   5        0.099404     0.000000     0.000000   0.000000  0.000000        0      24851       0          0
creatr 6        0.100440     0.999885     9.955076  99.127270  0.000000        0      25110       0          0
server 6        0.089360     0.716320     8.016135  65.388716  3.282945       10      22340       0      18698
exit   6        0.099916     0.000000     0.000000   0.000000  0.000000        0      24979       0          0
creatr 7        0.100680     0.999580     9.932281  98.944603  0.000000        0      25170       0          0
server 7        0.099600     0.793673     7.968626  62.543290  2.748247       10      24900       0      20793
exit   7        0.100072     0.000000     0.000000   0.000000  0.000000        0      25018       0          0
bus             0.247235     0.123323     0.498610  -2.035855  0.023469        8      61809       0       9695
```