DYNAMIC PROCESS MIGRATION FOR LOAD BALANCING IN DISTRIBUTED SYSTEMS

By

Chong-wei Xu

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1986

ABSTRACT

DYNAMIC PROCESS MIGRATION FOR LOAD BALANCING IN DISTRIBUTED SYSTEMS

By

Chong-wei Xu

A distributed system is an integrated system consisting of a set of interconnected peer processors coordinated through distributed control algorithms. In distributed systems, a mechanism which moves a process from one processor to another is called process migration. A difficult problem in designing a distributed system is finding an efficient scheduling method to evenly utilize distributed resources. We have proposed a distributed control algorithm, called the distributed drafting algorithm, to balance the load of various processors in a homogeneous distributed system. Processes are migrated from heavily loaded processors to lightly loaded processors to improve system performance. The drafting algorithm is network topology independent and accommodates dynamically changing system behavior. It attempts to compromise between two contradictory goals : maximizing the processor utilization and minimizing the communication overhead. Simulation results show that the drafting algorithm efficiently achieves load balancing and outperforms the conventional bidding algorithm. The dynamic processor pairing concept comes from the drafting algorithm. Various dynamic pairing strategies are discussed and two major algorithms are implemented in a token ring environment by using the writable message format, a convenient vehicle for carrying negotiation information.

ACKNOWLEDGMENTS

I would like to heartly thank my adviser Dr. Lionel M. Ni for his significant contributions to this research, technical guidance, editorial advice and above all, his encouragement and patience.

I would like to thank Dr. Erik Goodman, Dr. Herman D. Hughes and Dr. Stephen Dragosh for the time and effort they devoted to helping me on developing and proofreading this thesis.

Numerous people contributed the efforts to complete this thesis. In particular, I would like to thank Thomas Gendreau for providing the formal description of the drafting algorithm, and Ching-Farn E. Wu for his helping on using the facilities in the Distributed Computing Research Laboratory.

There are many people who deserve thanks for encouraging and helping me to continue my study over the last three years after I was injured in a traffic accident. I must apologize to them for not having the space to list all of their names. I will never forget them.

Finally, I'd like to thank my wife and my mother for their concern and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

The rapidly decreasing cost of processors has created an economic incentive for distributed processing. The remarkable benefits of distributed processing have given a technical incentive for distributed processing itself. The trend toward development of distributed systems is becoming more and more obvious. A distributed system makes all data and information resources easily and uniformly accessible, provides high performance and increases reliable operations.

The goal of distributed systems is to provide a unified service environment. In other words, an ideal distributed system should be able to provide a location independent service to the users. Various services are provided in a fair, reliable, and efficient manner. To achieve this goal, a variety of features are required by the system. Among them, a desirable feature in designing a distributed system is to find an efficient scheduling method so that distributed resources may be evenly utilized, which is the topic of this study.

1.1. Distributed Computer Systems

A *distributed system* is an integrated system consisting of a set of peer processors interconnected by an underlying communication network and coordinated through distributed control algorithms. The physically distributed hosts and underlying network provide a spatially distributed architecture. The distributed control algorithms take advantage of the distributed architecture, hide the architecture from users, and construct a logically integrated and temporally distributed system.

There are three types of distributed architectures which depend on the physical configurations [MeBo76]. One is the *loosely-coupled multiprocessor,* in which the operating system is partitioned into utilities distributed around the system, such as Medusa for Cm* [OuSS80]. A process may also be partitioned into many subprocesses and executed concurrently on different processors. Another is the *long-haul network* where many processors are interconnected over low bandwidth communication links. The third one is the *local area network*, in which processors are spread in a limited geographical area. The local network has a high transmission rate and a low error rate. There is no shared memory among the processors in the local network, and the exchange of information between processors takes a non-negligible amount of time.

There are two types of communication subnets : point-to-point channels and broadcast channels [Tane80]. In the first one, a pair of processors may be connected by a dedicated communication link. Several communication links and processors may be interconnected to form various network topologies. A message may have to go through many intermediate processors to reach the destination processor. Such a subnet allows simultaneous transmission of messages over different channels. In a broadcast subnet, a single communication channel is shared by all processors. Messages sent by one processor are able to be received by all others. A proper channel allocation mechanism, such as Ethernet, must be adopted to allow mutually exclusive access to the communication channel.

From the characteristics possessed by processors, a distributed computer system may be classified into one of two categories: the *homogeneous distributed system* in which various processors have compatible instruction sets and employ the same operating system, or the *heterogeneous distributed system* in which processors may not be compatible and run different operating systems.

A distributed system may be viewed as a hierarchy of levels of abstraction [Lann79]. Each level of abstraction comprises a number of *processes* responsible for the management of ***resources*** existing at each level. The processes in distributed systems must interact with each other by ***message passing*** for the purpose of performing activities in utilizing resources, such as data, peripherals, wires, memory, and processors. Processes may be grouped according to the ***functions*** performed. Processes may initiate ***operations*** which must be atomic to maintain the ***consistency state*** of the system. The state of a system at time t is defined as the set of system parameter values of interest at t. Any variation in these values should be viewed as a transition to another state. Consistency means that the system is kept in a state such that the set of values of the system parameter of interest is meaningful to the processes or the external world. At some initial time, if the state of the system is consistent, only the indivisibility of operations keeps the system consistent as those activities are being performed.

The distributed control algorithms may reside at any level of abstraction. There is no unique process enforcing the consistent view of the global state. The processes spreading over different processors in the system do not share similar physical space and lack a common time reference. The absence of uniqueness, both in time and space, makes it possible at any time, for several processes or control algorithms to observe different or inconsistent views of the global system state and decide simultaneously to initiate activities. This may cause conflicts. In addition, unpredictable interprocess message transit delays force some control algorithms to work on an approximation of the global system state. These properties of distributed systems provide the difficult task of designing protocols and algorithms that do not lead to inconsistencies and deadlocks in the systems.

It is likely that some processors in a distributed system are idle while others not only have a process being served but also have some processes waiting.

This phenomenon implies that the system's workload is not balanced in that some processing resources are not fully utilized while others lack enough processing power to serve the processes. The average process response time increases nonlinearly when the processor becomes saturated. Both idle processors and saturated processors will reduce the utilization of system resources and degrade the system's performance.

1.2. Problem Statement

Some waiting processes may be migrated from busy processors to idle processors, which is known as *load balancing*, in order to improve overall system performance. Load balancing is used to balance the workload over the entire system and to adjust resource allocation, thereby improving overall performance and increasing the system's reliability.

In order to do so, a control algorithm is needed to make the decisions, such as *when* a waiting process may be migrated, *which* waiting process should be migrated, from *where* and to where the migration will take place, and *how much* global or local status of the system will be needed to make the decision. The solution should be the best one in terms of obtaining the highest benefits by paying the lowest overhead with respect to a given network environment. It is this research's goal to analyze properties of the distributed system, to investigate the design considerations of a control algorithm, and to provide efficient and effective answers to the above questions.

The existing algorithms for load balancing are classified into two general categories: *deterministic* and *stochastic*. A deterministic approach assumes prior knowledge of process execution time. Further, it is assumed that a process submitted to a distributed system is first split into pieces, called modules, by a software mechanism. The cost of processing each module in each processor, the cost of transferring modules to different processors and the number of data transfers among them are known in advance. The problem becomes one of assigning a fixed number of modules to the processors. This problem would be best called a *task allocation* problem.

Different approaches for solving task allocation problems have been surveyed by [CHLE80] and fall into three groups : graph theoretic [Ston78, StBo78], mathematical programming [LeMu77, ChAb82], and heuristic [GyEd76, BaMM76, Efek82]. These approaches are not mutually exclusive, as each uses techniques from other areas. Among these researchers, Stone [Ston78] defined a critical load factor which was determined for each module in a processor. Optimal assignment in a two-processor distributed system may be made by calculating all critical load factors ahead of time and comparing the computed load factors against actual load factors experienced. Efe [Efek82] developed a heuristic load balancing algorithm called the module clustering and reassignment algorithm for an arbitrary number of processors. Chou and Abraham [ChAb82] went further, taking processor reliability issues such as failure probability, checkpoint time, and restart time, into consideration.

All of the above approaches for deterministic process migration depend on the assumption that when a process is submitted to, or created in, a distributed network environment, it is dynamically partitioned into modules with the cost of processing each module known. In some applications, such as a query processing system, this assumption does not present serious difficulty [Efek82]. However, for most applications, these values may be difficult to obtain and are not user friendly, forcing the user or compiler to supply these estimates. They assume too much prior knowledge, take too much computation time, and do not react to dynamic changes in the system. They are of theoretical interest because they point the way to potential benefits of process migration.

The stochastic approach is more practical and deserves further study. The stochastic approach may fall into *static* (probabilistic) and *dynamic* (adaptive) balancing [NiL82b]. If the processing power of each processor and the capacity of each communication channel in the network are known with the average new process arrival rate estimated in advance for various classes of processes, a static balancing may be achieved by calculating the optimal process assignment probability for each class of processes to every executable processor [NiAb81]. Once the assignment probabilities are calculated, the scheduling overhead is small. Upon new process arrival, the destination processor is determined by picking a random number weighted by assignment probabilities. Once a process is assigned to a processor, it cannot be further migrated to another processor. Unless the process arrival rate is measurable and the system workload is stable, static balancing is only theoretically interesting. Its optimality is defined on a probabilistic basis and is not an absolute optimum. It does not cope with the change due to system workload fluctuations.

In dynamic load balancing, the destination processor is determined by the current status of processors and channels. If the transmission overhead is not considered and each process has a global view of the system, an optimal dynamic balancing essentially results in an M/M/N queue [Klei75]. In other words, the average process response time of an M/M/N queue provides the lower-bound a computer network with N processors can achieve. Dynamic balancing is definitely better than static balancing if scheduling overhead is not very significant. The detailed dynamic process migration strategy will be discussed in Chapter 2.

1.3. Review of Past Work

A few research efforts were conducted in the area of dynamic load balancing. Chow and Kohler [ChKo77] developed six queueing models for a distributed system consisting of two identical processors, where each processor was modeled as an M/M/1 queue. One of them, which was most relevant to the study, assumed that each processor had its own input source and that interprocessor communication was allowed for the purpose of process migration. The communication channel initiated a process migration from one waiting queue to another whenever the number of processes in one queue was two or more greater than the number of processes in the other queue. Unfortunately, the authors showed that there was no efficient technique to analyze the performance of this kind of system.

Stankovic [Stan81] formulated the distributed processes scheduling problem in Bayesian decision theory terms. The process scheduling problem was formulated such that each decision was made on the basis of imperfect knowledge; yet, taken together, the cooperation of the decision makers led to some global optimum. A utility function was defined for each state (number of processes in each processor) and each possible action (process movement). Using Bayesian decision theory for off-line calculation of maximizing actions, the system adapted with respect to varying processor loads. However, off-line calculation was not practical for an efficient and effective migration effort.

Bryant and Finkel [BrFi81] suggested a distributed and stable process migration algorithm for homogeneous point-to-point networks. Employing a time-sharing discipline, the estimated response time was based on the execution time that a process has received in the past. A pairing algorithm was used to provide a pair for two adjacent unbalanced processors. Once a pair was formed, each process in the heavy-load processor was evaluated according to its estimated response time in both processors including the one-way transfer time. The process migration procedure was repeated after each process movement until the pair was balanced. The pair was then broken and each processor formed another pair with its other neighbors. This is a dynamic preemptive algorithm. Although the algorithm is stable, too much computation overhead is involved.

A practical approach, the *bidding algorithm*, was originally designed for a ring-structure distributed computing system (DCS) [Farb73]. It contributed a rudimentary dynamic load balancing algorithm. In a bidding algorithm, whenever a new process arrived at a processor, a request for bids is broadcast to all other processors in the system. The bid could be estimated execution time, cost, etc. Processors returned bids to the requesting processor. When all bids were received, the processor chose the best bid as the winning bid and a message sent to the owner of that bid. At the point the bidder chose to accept or reject that notification due to the changes of the state between the time the bid was made and the time it was notified that it won the bid. If the bid was rejected, the bidding algorithm started over.

A simple version of a bidding algorithm was implemented in the Engineering Computer Network (ECN) at Purdue University [Hwan82]. In the ECN network, an estimated "load average" was maintained in each processor's kernel. The system provided a status report of the network in which the load average and utilization of each processor was updated periodically. Before a command was processed, the load average from every available processor was obtained and the one with the minimum load average was chosen. The process migration protocol did not consider the overhead of message transmission from the source processor to the destination processor and the return path. The algorithm was similar to the bidding algorithm mentioned above except that the selected processor was forced to accept the migration regardless of the current state of the processor. The communication overhead was thus reduced. But it was possible that a processor won many bids from many other processors and suddenly became overloaded. Obviously, both of the above process migration protocols should be classified as dynamic migration in which the migrant process could not be further remigrated.

Smith [Smit80] developed a contract net protocol based on the bidding principle. Task distribution among the processors in a distributed system was affected by a negotiation process which was an agreement between a processor with a task to be performed and a processor capable of performing that task. A processor, namely a *manager*, generating a task normally initiated contract negotiation by advertising existence of that task to the other processors. Whenever a processor was available, it checked the eligibility specifications of all task announcements that it received if it was eligible to bid on a task, and selected a task on which to submit a bid. The successful bidder was informed that it was a contractor for a task through an announced award message. Then, the manager and contractor communicated with each other via information message, report and termination message to cooperate in executing the task. Other related works on bidding algorithm can be found in [StSi84].

1.4. Summary of Research Contributions

As can be seen, in the dynamic process migration field, the bidding algorithm is the representative algorithm in the past. Its disadvantages, which will become clear later in our study, challenge us to look for a better control algorithm. The solution is the ***drafting algorithm***, which is network topology independent and accommodates dynamically changing system behavior. It attempts to compromise between two contradictory goals : maximize the processor utilization and minimize the communication overhead. Simulation results show that the drafting algorithm efficiently achieves load balancing and outperforms the conventional bidding algorithm. From the analysis shown in Chapter 3, the drafting algorithm achieves the optimal solution M/M/N queueing model to certain extents. The internal load state aspect (predetermined threshold value or dynamically determined threshold value) exposes the essence of the load balancing control algorithm and overcomes the major deficiencies of the bidding algorithm. The variation of the drafting algorithm described in Chapters 6 and 8 combines some concepts of the bidding algorithm with the drafting algorithm to give the best solution for specific environments, such as token ring and CSMA/CD network environments.

1.5. Organization of the Dissertation

The remainder of the dissertation is organized in the following manner. Chapter 2 describes the process migration mechanism, which is able to move a process from one processor to another in a distributed system, addresses the classification of processes, and results in what kind of processes can be migrated. Chapter 3 analyzes the unbalancing phenomenon existing in distributed systems and explores the necessities of load balancing and its implementation considerations. Chapter 4 deals with performance evaluation methods, emphasizes there is no tractable way for the performance analysis of dynamic load balancing except the simulation method, and presents the major points of the simulation program used in data gathering for the algorithm analysis. Based on the fundamentals of the process migration mechanism and load balancing considerations described in Chapters 2 and 3, the drafting algorithm is explained in Chapter 5. Simulation results support our qualitative analysis of the algorithms and point out the differences between static balancing and dynamic balancing. In Chapter 6, the dynamic pairing concept coming from the drafting algorithm is implemented within a specific network environment to further exploit the characteristics of the drafting algorithm and the properties of the network. Based on features possessed by the token ring network, a writable message format is designed as a convenient vehicle to carry negotiation information when studying dynamic pairing algorithms. Chapter 7 demonstrates the performance of dynamic pairing algorithms and further discusses the differences between two major dynamic load balancing strategies, the bidding algorithm and the drafting algorithm, on the performance issues. Chapter 8 attempts to extend the dynamic pairing concepts into token passing bus network and shows the implementation considerations on the CSMA/CD environment. Finally, in Chapter 9, conclusions about major improvements of the drafting algorithm over the weaknesses of the bidding algorithm are summarized. These conclusions give rise to some open questions, and suggest several possibilities for further extension.

CHAPTER 2
PROCESS MIGRATION

In a distributed system, a mechanism which is able to move a process from one processor, namely the *originating processor*, to another, namely the *remote processor*, is called *process migration*. The process which is migrated is known as a *migrant process*. It is process migration that attempts to improve the system performance, to allow the sharing of resources, and to increase the system reliability. With process migration, a process may be migrated from a busy processor to a less busy processor to reduce response time of the migrant process. Consequently, the system performance is improved. A process requiring a special resource may be migrated to another processor with the desired resources. Processes may also be migrated from a failed processor to other active processors to continue the service. Process migration is completed either explicitly by the user or implicitly by the system in a user-transparent manner.

2.1. Classification of Processes

The unit migrated by the mechanism of process migration is a *process*, which is different from a procedure or a program. A procedure is a portion of a program. A procedure call does not return to the caller until the called procedure is completed. But a process may spawn a new process and return to the caller immediately. For example, the fork() system call in the UNIX operating system returns to the caller after starting the child process and allows the execution of both the calling program and its child processes to proceed concurrently. Conversely, a program consists of a code executed by a single process. But processes are not uniquely associated with a piece of code; multiple processes may execute the same code simultaneously.

A process should be viewed as a sequential machine, namely a sequential program augmented by a named set of state information which allows the program to be restarted at any point. Processes have their own address spaces and do not interfere with each other due to the fact that each process has its own local variables, formal parameters and procedure calls. Memory for local variables and formal parameters is associated with the process executing the procedure, not with the code in which they appear. In virtual memory management systems, a process consists of the program being executed, along with the program's data, stack and state. The state consists of the execution status, dispatch information, incoming message queue, memory tables and the process's link table. Links are important paths by which a process communicates with the environment. A link is a kind of capacity which provides the only connections for a process to the operating system, system resources and other processes [PoMi83]. Thus, from the structure point of view, a process is an abstraction most operating systems use to provide the execution environment for running a program.

Processes are classified as *system processes* and *user processes* according to the functions they perform. A system process frequently becomes a *server* process, that is, most other processes can ask it to perform some functions. Those processes, for example, include process manager, memory scheduler, file system, command interpreter and packet reception processes, etc. User processes are those processes issued by users. They gain access resources under the support and control of system processes. According to the relationship between user processes, they can be grouped into two categories : *independent processes* and *cooperative processes.* An independent process is a process which is independent from every other process in the sense that it does not mention any variable occuring as a target variable in any other process. The independent processes

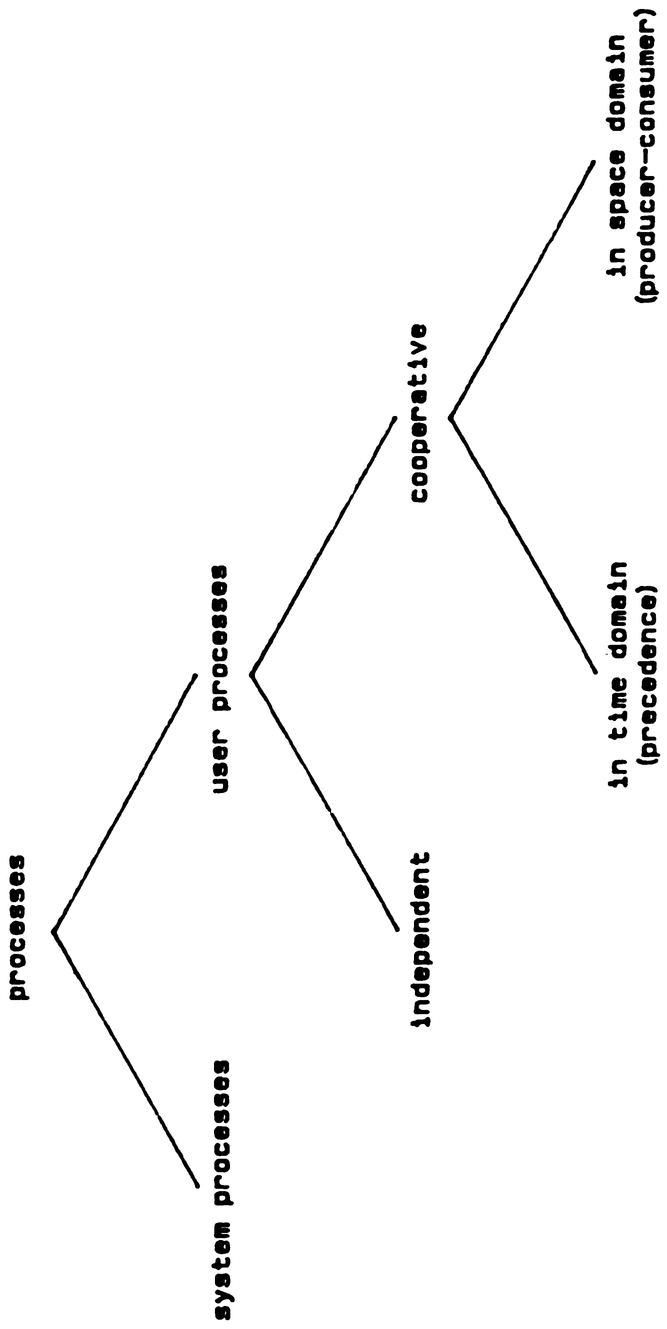

processes
system processes
user processes
independent
cooperative
in time domain (precedence)
in space domain (producer—consumer)

Figure 2-1 Classification of processes

may be executed asynchronously.

Cooperative processes require cooperation, of all related processes, involving either time or space domain. When cooperation refers to time domain, it indicates there is precedence between the cooperative processes. The precedence of any given pair of processes reflects the logical execution sequence with respect to time. There are potentially three basic relations [Huan85] between a pair of processes A and B: (1) A precedes B, (2) A succeeds B, and (3) A parallels B. A fourth precedence also exists : A and B precede as well as succeed each other; i.e., there is a two-way process communication or cooperation requirement between A and B. If A precedes B, A must be executed before B; if A succeeds B, A must be executed after B; if A parallels B, A and B may be executed independently. If A precedes as well as succeeds B, A and B must be executed concurrently.

When cooperation refers to space domain, the processes share a set of global memory space or data structures, and have a producer-consumer kind of relationship. A producer process creates output to be utilized by a consumer process. For example, a producer may accumulate a buffer of data from an input device to be processed by a consumer process. Some cooperation is required between the two processes, such as the transfer of full buffers, the return of empty ones, or insurance of data integrity, etc. The classification is summarized in Figure 2-1. In order to solve problems dealing with mutual exclusion, synchronization, deadlock, and determinancy associated with the cooperative processes, rapid communication between these processes is required.

2.2. Migratable Processes

It is natural that some processes are more suitable for migration than others in order to improve system performance. "Suitable" means easier migration and lower overhead. Thus, *migratable processes* must be distinguished from ordinary processes in order to identify which process will be migrated at a specific moment.

Processes are discriminated as *task* and *job* by referring to different granularity [Klei85]. Task migration takes a single job composed of multiple tasks and assigns them to different processors. Job migration assigns the entire job to a single processor. When a job arrives at a processor, the processor determines whether a job must be processed locally or scheduled for remote processing. In the latter case, not only is a delay incurred as a result of sending the job and its results over the network, but communications between processes (tasks or jobs) is increased due to the separation of processes over physical space. The higher the granularity employed, the more interprocess communication over the network is involved. This type of overhead caused by migration of cooperative processes must be reduced as much as possible. Therefore, the job is selected as the migrating unit whenever the process migration is referred to throughout this dissertation.

System processes in distributed systems reside in the local processor (it may not be true in the MIMD systems where a system process may belong to the whole rather than a local system). In homogeneous distributed systems, every processor has a similar operating system. It is not necessary to migrate system processes from one processor to another under normal conditions. System processes are often linked with other processes which make migration complicated and increase channel traffic due to communication between the linked processes. Similarly, the relationships of cooperative processes require clustering of the processes in one place for convenience. If they are separated over different physical spaces, not only are the migrations themselves difficult but the communication between them increases the burden of the channel and slows down execution speed of the processes. Consequently, only independent processes not sharing any variables with other processes are suitable for migration under certain conditions. The criteria for specifying these conditions may be classified as *static* and *dynamic*. Dynamic criteria are associated with the current states of the system in question while static criteria are predetermined.

Static criteria include:

(1) Processor discipline. Processes whose memories are allocated are usually not migratable processes because allocation implies the processor has the capacity to serve allocated processes. De-allocation increases migration overhead. For example, for a FCFS discipline, the process being served is not migratable while for a round robin discipline, the processes receiving a certain amount of CPU service are not migratable processes.

(2) Load state definition. By using threshold values, the load states of a processor are statically predefined. Some non-allocated processes in a heavy-load processor are considered migratable processes. Details are discussed in Chapter 3.

(3) Process priority. If processes are assigned priorities, the processes with lower priority are not migrated whenever there are migratable processes with higher priorities.

(4) Migration strategy. Migration strategy is described in Chapter 3. For instance, if a single migration strategy is employed, then the migrant process can not be migrated further. Conversely, in the repetitive strategy, the migrant process can be migrated repeatedly. Under the strategy of preemptive migration, the processes being served may be migrated, but under the strategy of nonpreemptive migration the processes being served are not allowed to migrate.

Figure 2-2 summarizes these specifications. Users may also specify whether processes are migratable or not. Some special resource requirements also affect the definition of the migratable processes.

Dynamic criteria can dynamically determine the threshold values, the estimated processor load, and the status of the channel traffic etc. We will not discuss these criteria further in this study.

Based on the discussions mentioned above, a ***migrant process*** is an independent and non-allocated process which satisfies applied criteria at the instant when the migration decision is being made.

2.3. Migration Procedure

Once the decision is made to migrate a process, the following steps are performed:

(1) Removal of the migrant process. The migrant process is marked "in migration" and removed from the process table.

(2) The destination kernel is asked to accept the process. A message is sent to the kernel on the destination processor, asking it to accept the migrant process. The message contains information about the size and location of the process's resident state, swappable state, and code.

(3) Process state allocation on the destination processor. An empty process state is created on the destination processor. The newly allocated process state has the same process identifier as the migrant process. Resources such as virtual memory swap space are reserved at this time.

(4) Process state transfer. Using the send data facility, the destination kernel copies the migrant process's state into the empty process state.

(5) Program transfer. Using the send data facility, the destination kernel copies the memory (code, data, and stack) of the migrant process into the

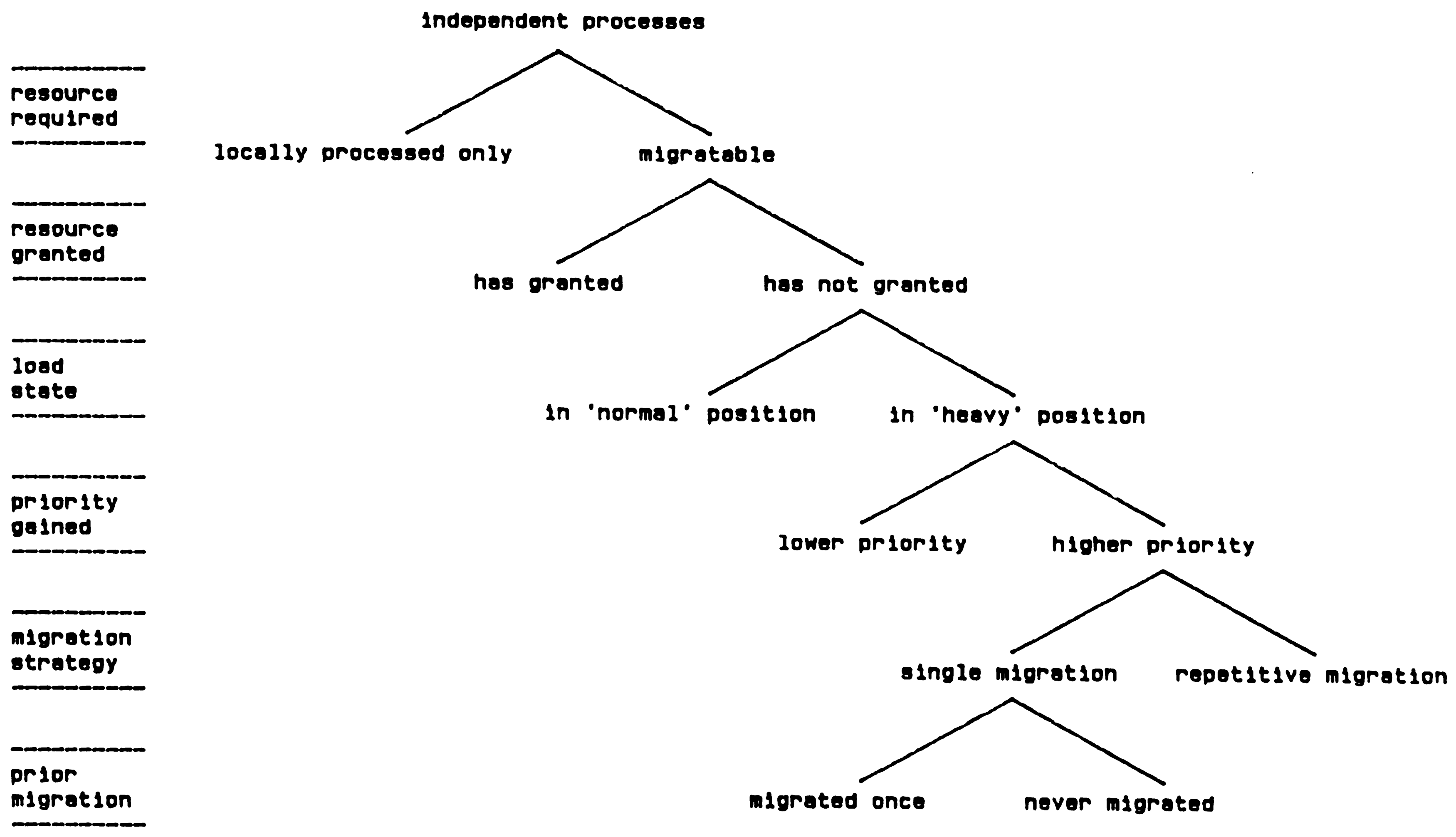


Figure 2-2 Classification of migratable processes

destination processor.

(6) Forward pending messages. Upon being notified that the migrant process is settled down on the new processor, the source kernel resends all messages in the queue when the migration started, or that have arrived since the migration began. The source kernel also changes the location part of the migrant process addressed to reflect the new location of the migrant process.

(7) Clean-up of the migrant process's state. On the source processor, all states for the migrant process are removed, and memory for tables is reclaimed. A forwarding address is left on the source processor to forward messages to the process at its new location. The forwarding address is a degenerate process state whose only contents are the (last known) machine to which the process was migrated. A normal message delivery system tries to find the migrant process when a message for it arrives. The source kernel has completed its work and control is returned to the destination kernel.

(8) Restarting the migrant process. The process is restarted in whatever state it was in before being migrated. Messages may now arrive for the process, although the only part of the system that knows the new location of the process is the source processor kernel.

The process migration considerations based on the CSMA/CD nerwork and the token-passing environment will be discussed in Chapter 8.

CHAPTER 3
LOAD BALANCING CONSIDERATIONS

In a distributed system, every processor has its own job entry point. New jobs arriving from this entry point are referred to as *local arrivals* or *external arrivals*. Jobs also arrive from other processors in the system. These jobs are *migrant processes* and are referred to as *remote arrivals* or *internal arrivals* to that processor. Each processor has a different local arrival rate with the flow of local arrivals fluctuating dynamically. These make load unbalancing a common phenomenon in distributed systems. In this chapter, we first give the theoretical foundation of load balancing consideration. Then we will explore the dynamic behavior of distributed systems and analyze the advantages and disadvantages of various load balancing approaches.

3.1. Theoretical Foundation

In this section, a theoretical upper bound of the system performance (or the lower bound of the average process response time) is attempted which can be achieved by employing load balancing algorithms. Then we try to show the reason why we would like to investigate dynamic load balancing approachs rather than the probabilistic ones. In order to analytically demonstrate these points, several assumptions must be made.

It is assumed that each processor in the distributed system is modeled as an M/M/1 queueing system [Klei75]. A distributed system with N processors is then modeled as N M/M/1 queues. For simplicity, the communication delay among these processors is ignored. For a homogeneous distributed system, all processors have the same service rate with a mean service rate μ. Also, let λ_i be the local arrival rate to the processor i for $i=1,2,...,N$. Thus, the total arrival rate to the system is

$$R = \sum_{i=1}^{N} \lambda_i . \tag{3-1}$$

In the ideal case, a well balanced system implies each processor has a global view of all processors in the system. Thus, if the communication delay is ignored, a process waiting for execution implies that all processors must be busy executing other processes. If a processor is idle, then there must be no process waiting for execution. This well-balanced behavior can then be modeled as an M/M/N queue with N processors sharing a single waiting queue. The average process response time of an M/M/N queue with total arrival rate R and total mean service rate μN provides the lower bound on the mean process response time we can achieve as shown below.

$$T = \frac{p_0 (R/\mu)^N \; R/\mu N}{N!\,(1-R/\mu N)^2\, R} + \frac{1}{R} \tag{3-2}$$

where

$$p_0 = \frac{1}{\sum_{i=0}^{N-1} \frac{(R/\mu)^i}{i!} + \frac{(R/\mu)^N}{N!} \frac{1}{1-(R/\mu N)}}$$

Note that Eq.(3-2) is well-known and can be found in many queueing books (e.g., [Klei75]). Also note that if we do consider the communication delay, the lower bound will increase.

Now we discuss why we want to study dynamic load balancing rather than probabilistic load balancing. If the total job arrival is uniformly distributed to all processors, the system is said to be ***probabilistic balancing.*** In this case, the local arrival rate to each processor is

$$\lambda_i = \lambda = R/N. \tag{3-3}$$

The system can then be modeled as N identical M/M/1 queues. The mean process response time in this case is

$$T_i = \frac{1}{\mu - \lambda_i} = \frac{1}{\mu - \lambda} \tag{3-4}$$

Again Eq.(3-4) can be found in most queueing books.

Figure 3-1 compares mean process response time of the theoretical well-balanced to the case of probabilistic balancing. It can be seen that probabilistic load balancing is not good enough or not well balanced because it is measured on a probabilistic basis and does not cope with the instantaneously changing behavior of the system.

A distributed system is said to be *unbalanced* if some processors have some processes waiting in the queue for service while some other processors are idle. This implies system resources are not fully utilized. A probabilistically balanced system exists where all processors have the same mean process arrival rate. Formally, we define the *degree of load imbalance* (DLI) as the probability that there is at least one processor idle while there are one or more processes ready for execution. An M/M/N queue is well-balanced. In other words, the DLI of an M/M/N queue, can be denoted by DLI(1-M/M/N) is zero.

For an M/M/1 queue, the probabilities of having an idle processor or at least one process in the waiting queue, with no idle processor and no waiting process, are denoted by a , b , and c , respectively. The degree of load imbalance for an N-M/M/1 system is

$$\text{DLI}(N\text{-}M/M/1) = \sum_{i=1}^{N-1} \left[\sum_{j=1}^{N-i} \frac{N!}{i!j!(N-i-j)!} a^i b^j c^{N-i-j} \right] \tag{3-5}$$

where

$$a = p(0), \quad b = 1\text{-}p(0)\text{-}p(1), \quad c = p(1) \tag{3-6}$$

and

$$p(0) = 1\text{-}(\lambda/\mu) \tag{3-7}$$
$$p(1) = (\lambda/\mu)(1\text{-}\lambda/\mu) \tag{3-8}$$

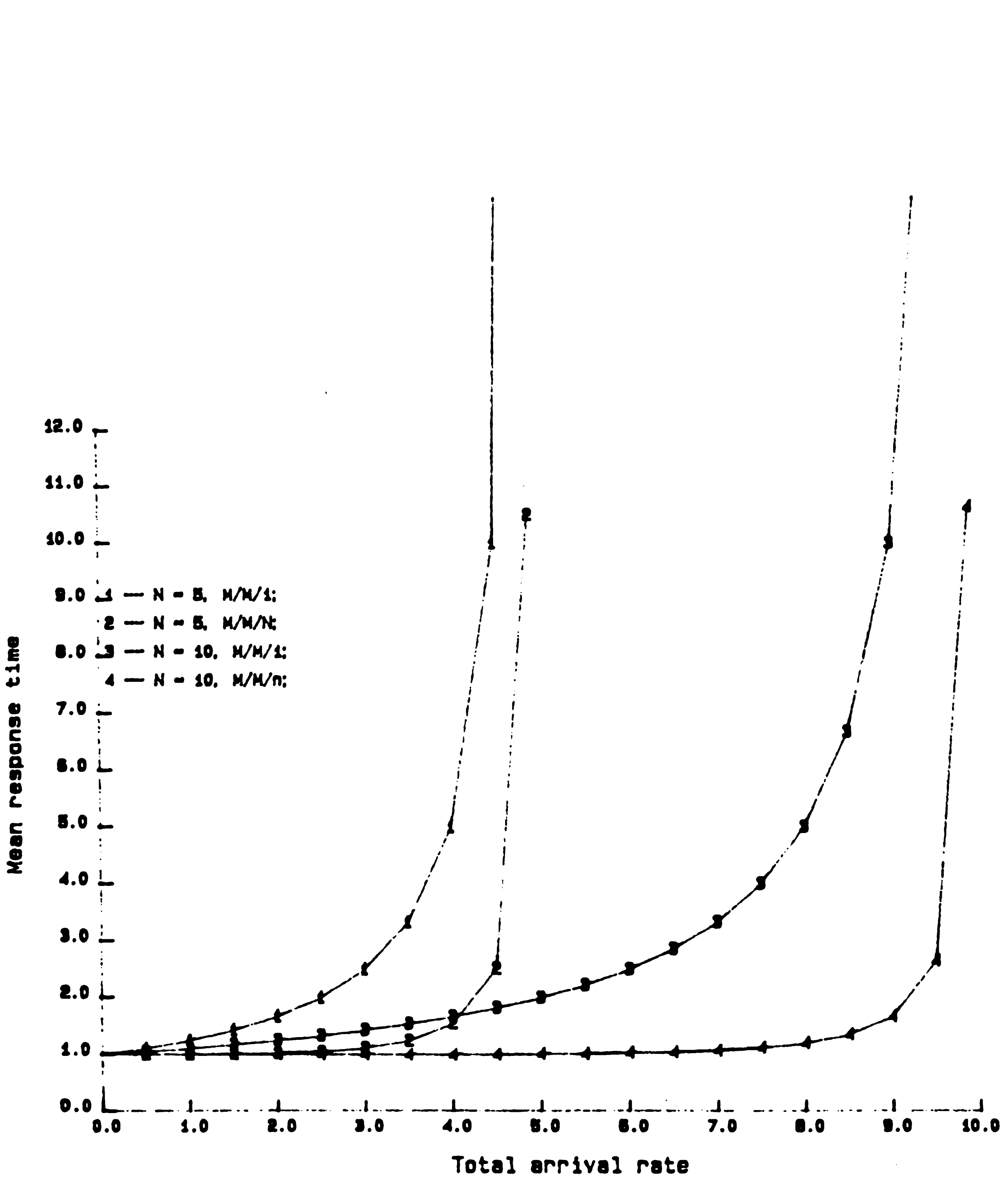


Figure 3-1 Lower bounds of the different models

Note that $p(i)$ is the probability of having i processes in an M/M/1 queue and its derivation can be found in many queueing books.

Let $\rho=1-p(0)=\lambda/\mu$ denote the processor utilization or the ***load*** of a processor. Figure 3-2 shows the value of DLI versus the processor load with respect to different number of processors (N). It can be observed that as N increases, the system performance decreases due to the greater degree of load imbalance. The more processors a distributed system has, the more likely the system is unbalanced.

The performance degradation due to load imbalance becomes small when the system is very lightly or very heavily loaded. With the lightly loaded situation, most processors are idle with a few waiting processes when the system is probabilistically balanced. In the heavily loaded situation, most processors have processes waiting for service. It is unlikely that some processor is idle. Note that if the total arrival rate is greater than the total system capacity (i.e. $R > \mu N$), the system is unstable and even load balancing cannot help. In other words, μN is the upper bound of the system capacity that it can serve.

A dynamic load balancing strategy tries to balance the system load at any instant, depending on the system's current status. In the above discussion, communications overhead is ignored and each processor has a global view of the whole system. This is unrealistic in a distributed system. First, the communication delay is non-negligible. Second, in order to have a global view of the system, each processor must periodically report its current status to all other processors. Thus, more communication overhead is involved. Due to the communication delay the current status may be outdated when it reaches other processors. Design of dynamic load balancing algorithms should always consider the trade-off between maximizing system utilization and minimizing communication overhead.

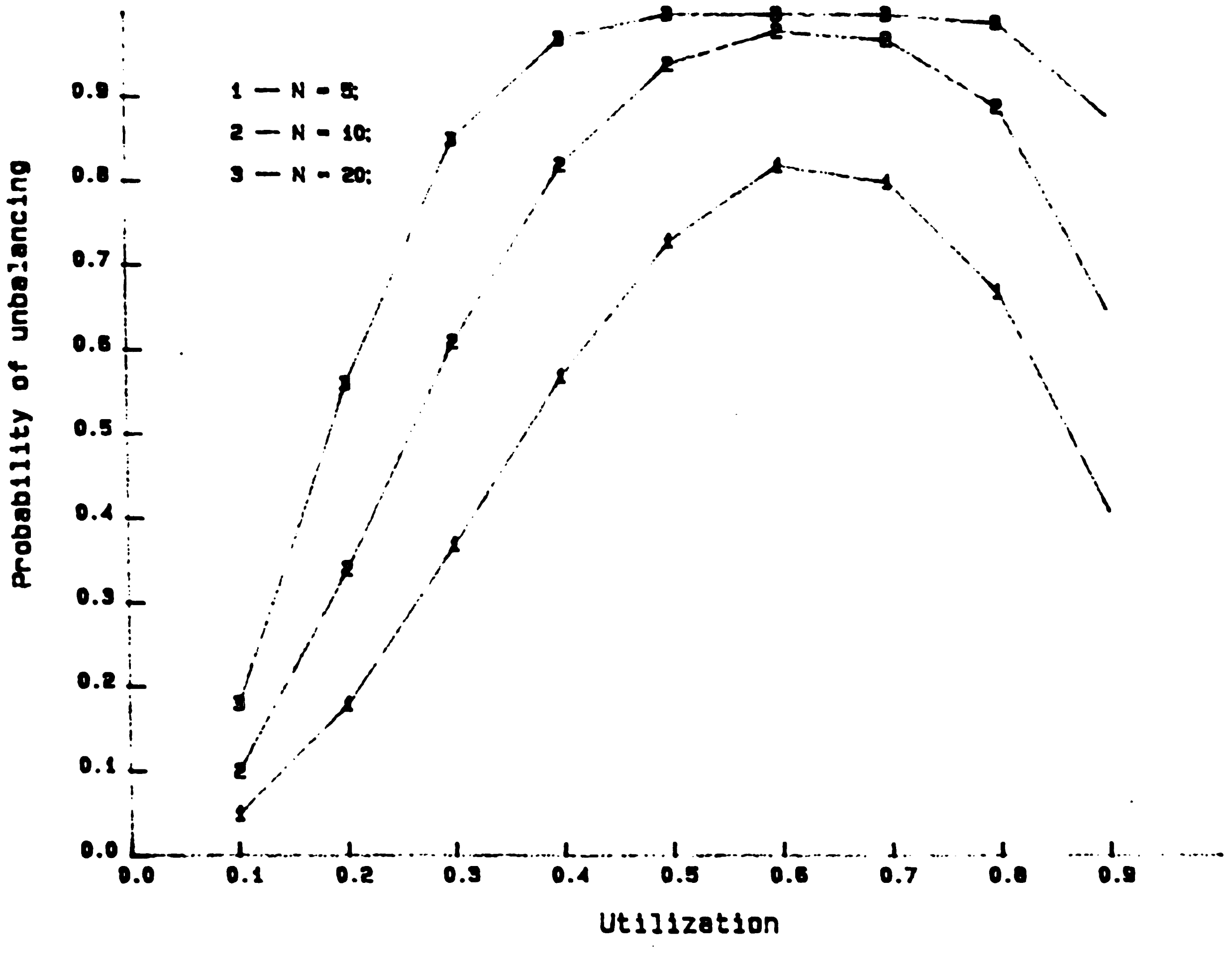


Figure 3-2 Unbalancing phenomenon in distributed systems

3.2. Classification of Load Balancing Strategies

A dynamic process migration algorithm is a high level protocol which considers the semantics of the information being passed. It offers a structure that assists the system designer in deciding what processors should say to each other, rather than how to say it [Smit80]. According to the ISO reference model [Zimm80], a process migration protocol should be implemented in the application layer. Algorithms of load balancing may be classified into deterministic and stochastic categories. The stochastic approach falls into static (probabilistic) and dynamic (adaptive) balancing as described in Chapter 1. We are only interested in dynamic load balancing.

In dynamic load balancing, the destination processor is determined by the current status of processors and channels. A dynamic balancing algorithm may allow a process to be migrated more than once, called *repetitive migration*, or only once, called *single migration*. In a time-sharing system, a processor preempts the execution of a process due to time quantum expiration or other events. In a homogeneous system, a preempted process is migrated to another processor along with its process control block to continue its execution. This strategy is called *preemptive balancing*. With *nonpreemptive balancing*, once a process begins its execution, it can not be migrated even if it is preempted. The classification of load balancing strategies is summarized in Figure 3-3. Other works emphasizing the modeling and analysis of the process migration for load balancing can be found in [ChKo77, Stan81, GaLR84].

The priority of load balancing over other traffic on the network channels is an undetermined issue, because load balancing is what is called "making perfection still more perfect." No doubt, it is suitable for the case in which the traffic of the network channels is not very heavy. In the other case, it will generate more traffic but cannot guarantee to reduce the response time of the migrant

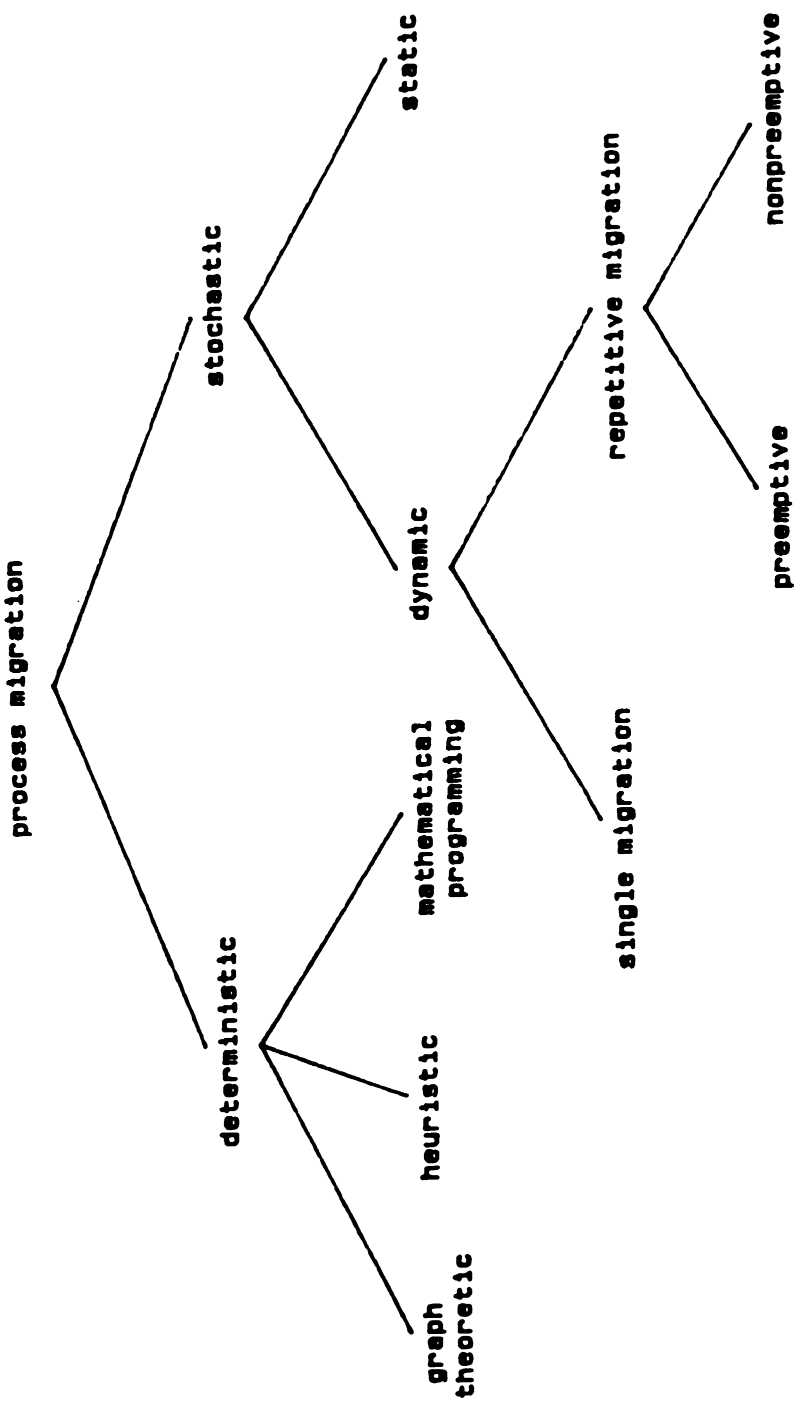


Figure 3-3 Classification of load balancing strategies

processes because the already heavily loaded channel traffic may unpredictablly postpone load balancing actions. What quantity could be used to distinguish the states of the channel traffic is unknown. Unfortunately, we ignore this issue in the first stage of study.

3.3. Algorithm Design Considerations

Load balancing can improve the performance of distributed systems. However, it will increase network traffic caused by the control information exchanges and process migration. Therefore, a good load balancing protocol design should consider the following trade-offs:

(1) maximize improvement of the system performance;

(2) minimize additional traffic caused by process migration;

(3) achieve fairness at the system level; i.e., maintain a FCFS discipline in the global sense.

The characteristics of a distributed system depend on the nondeterministic nature of concurrently running processors and the unpredictability of communication network traffic. These characteristics dynamically change network conditions and processor's loads.

In order to design a good algorithm, the trade-offs mentioned above must be considered, and the features of dynamic load balancing explored. Normal conditions of the system must be considered with special attention paid to the anomalies of the system as important design principles. We start by analyzing the bidding algorithm, described in Chapter 1 and briefly depicted in Figure 3-4, to investigate considerations of a good design. The bidding algorithm is briefly described as follows:

Suppose that there are N processors in a distributed system. A new process arrives at processor i. Processor i broadcasts a bid request message to all other

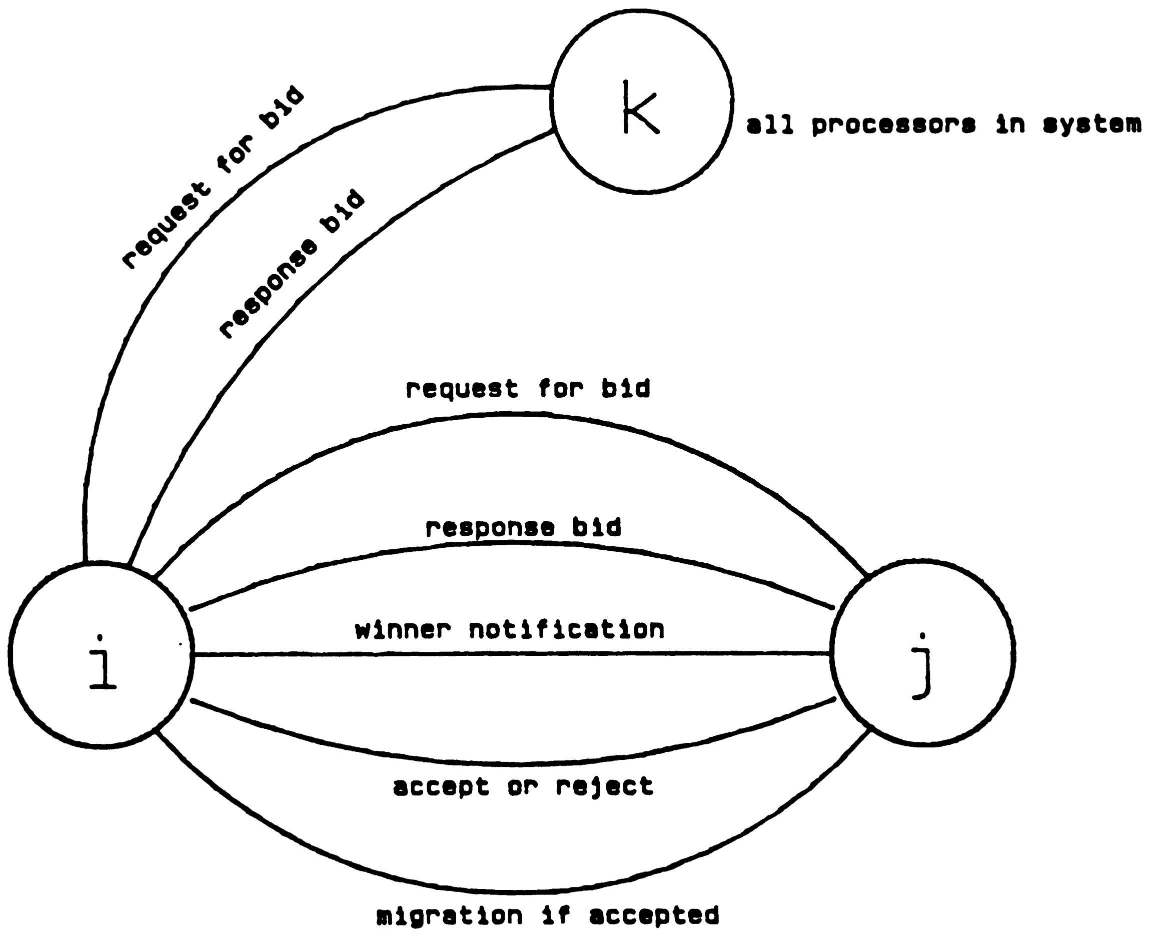


Figure 3-4 A brief illustration of bidding algorithm

N-1 processors. Every processor responds with a bid message to processor i. After receiving all the bids, processor i selects processor j as the winning bidder (because it is the least heavy processor) and then sends winner notification to processor j. At the moment processor j receives winner notification, it checks its current state. If it can accept a migrant process, it sends an accept message to processor i with processor i migrating the new arrival to processor j. Otherwise, processor j sends a reject message to processor i. The bidding algorithm, then, bins again.

Avoidance of wait-while-idle

A distributed system is unbalanced if some processes wait for service in some processors while some other processors are idle. This behavior is referred to as *wait-while-idle*. A good process migration strategy minimizes the wait-while-idle occurrence. One way of achieving this goal is to assign arrival processes to other active processors. The bidding algorithm migrates new arrivals to the least busy processor. However, it cannot avoid the occurrence of wait-while-idle because process migrations are triggered only by the new arrivals, i.e., the external arrivals. The fluctuation of the external arrivals influences the behavior of the system. As shown in Figure 3-5(a)(b), originally both queue 1 and queue 2 are idle with 20 new arrivals entering queue 1. The bidding algorithm causes 10 of them to migrate to queue 2. As a result, there are 10 processes in each of queue 1 and queue 2. Note the same length does not imply to the same load because the processes sizes are different. Suppose that no new arrival comes for a while and the 10 processes are all served by queue 2 but 2 processes are left in queue 1. The lengths of queue 1 and queue 2 become 2 and 0, respectively (Figure 3-5(c)) and wait-while-idle occurs. The only way to avoid wait-while-idle is to trigger process migration by the processes's internal states.

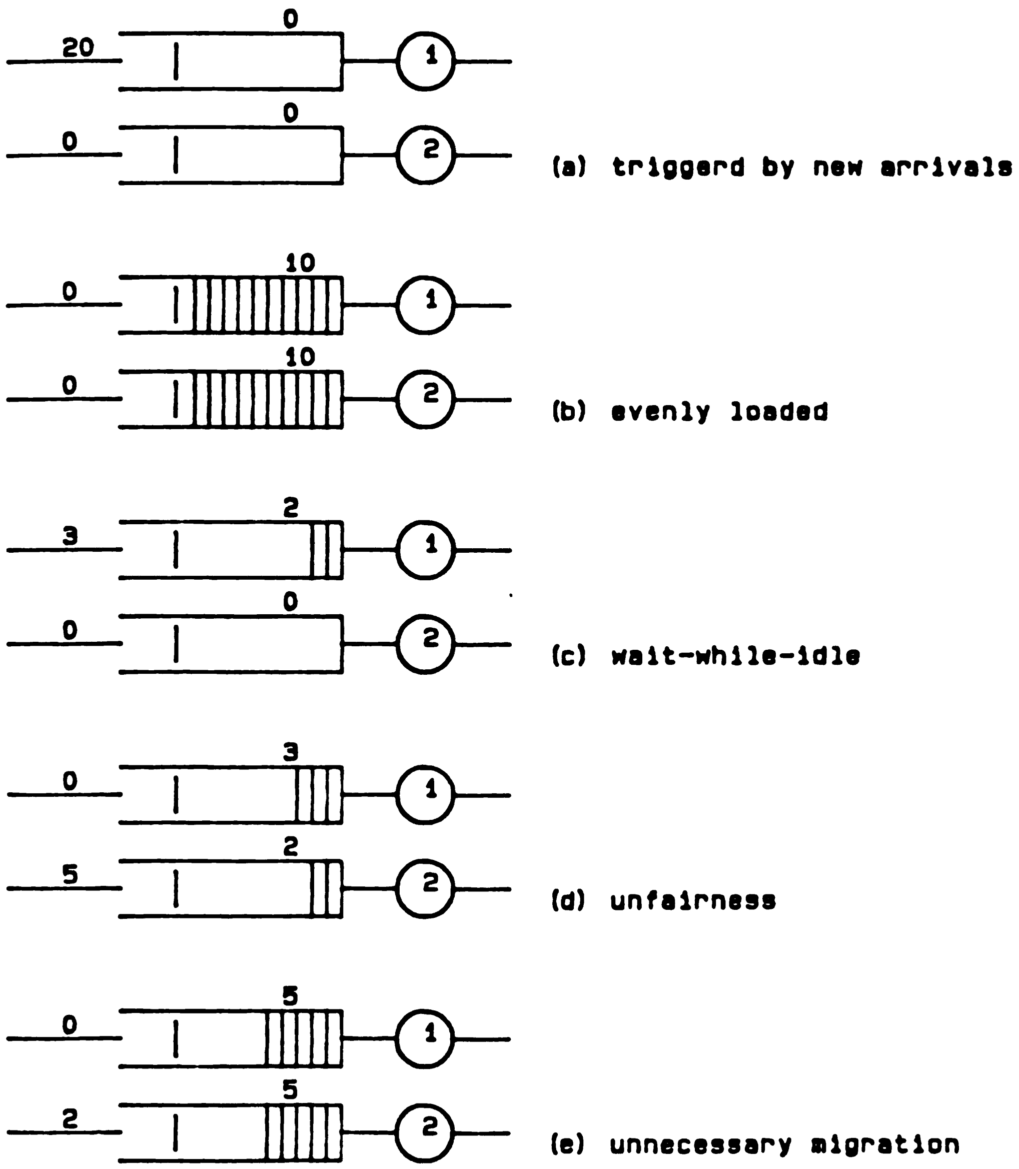


Figure 3-5 Problems with bidding algorithm

Avoidance of unfairness

Suppose that the discipline of all queues is FCFS. The migration algorithm would not only pay attention to load balancing but also keep FCFS discipline in the global sense of the system level in order to make services fair and to achieve the requirements of the M/M/N multiple processor FCFS queueing model. If an algorithm for load balancing arbitrarily chooses a process from a longer queue and migrates it to a shorter queue, then later arriving processes will be served earlier than earlier arriving processes. In Figure 3-5(c), 3 new arrivals come into queue 1. The bidding algorithm migrates 2 processes of the new arrivals to queue 2. These later arrivals may be served by queue 2 earlier than the earlier arriving processes in queue 1. The FCFS discipline is destroyed in the global sense. Thus the proper process must be selected from the waiting queue before migration in order to maintain FCFS discipline as much as possible. Proper process specification depends on the definition of the internal states of the processor.

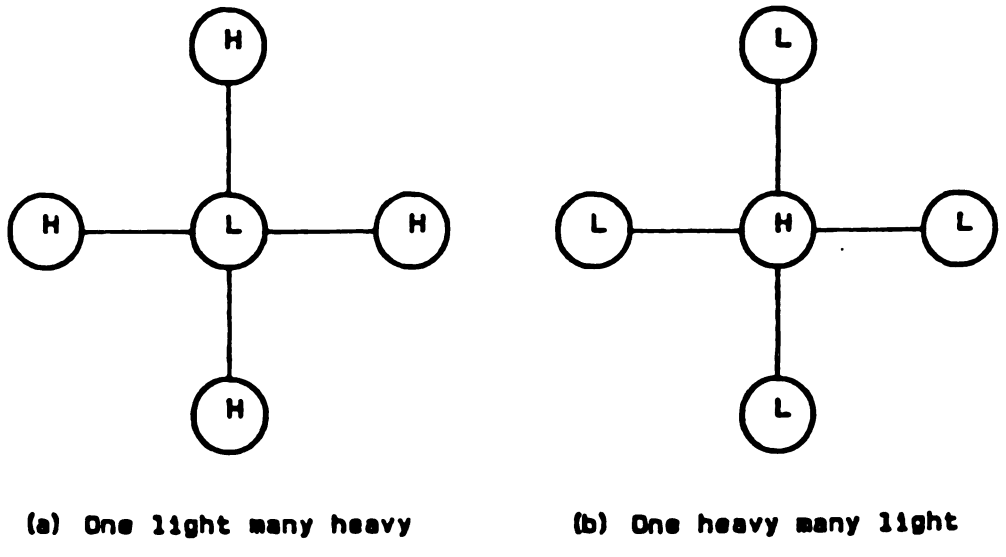


Figure 3-6 Problems with heavy-load processor initiating migration

Avoidance of unnecessary process migration

One way to balance the load is to evenly balance the number of processes in queues. Whenever a difference of 2 is reached between two queues, a process in the longer queue is migrated to the shorter queue. Later on, however, the original shorter queue may have more processes in its queue than the original longer queue; then some processes will be migrated back. From Figure 3-5(d)(e) we see that if 5 new arrivals enter queue 2, some will be migrated to queue 1, the original longer queue. This is called ***processor thrashing*** [BrFi81].

The goal of load balancing is to keep all processors busy rather than to keep them loaded evenly. The process migration algorithm is invoked only when migration is necessary. This means a process is migrated only when some processors have extra resource capacity while others lack resource capacity. Thus, by avoiding unnecessary process migration, process migration has to be triggered by the processor's internal load state.

Initiation of process migration

A bidding algorithm only migrates new arrivals from a given processor to a more lightly loaded processor. It is also possible to migrate processes from a heavily loaded processor to a less heavily loaded processor or from a lightly loaded processor to a more lightly loaded processor. There are three problems with the heavy-load processor initiating process migration. First, it is possible that several heavy-load processors each send one process to a light-load processor. As a result, the light-load processor immediately becomes overloaded. Second, executing the algorithm will further increase the burden of the already heavily loaded processor as shown in Figure 3-6. Third, it is much more difficult to achieve fairness by migrating the new arrival to a light-load processor because the new arrival process which has migrated to a light-load processor may be completed sooner than those processes entering the system earlier but remaining in some heavy-load processors. Allowing the light-load processor to initiate migration, permitting only one process migration at a time from a heavy-load processor to a light-load processor with proper definition of the migratable process, will alleviate these problems.

3.4. Internal Load States of Processors

So far, we have concluded that the internal load state of a processor must be defined. The processor's internal state is mainly reflected by the processor load. A processor is in the *light load* state if it has "extra capacity available" - namely, it can accept some migrant processes. If the processor lacks capacity, then it is in a *heavy load* state. Process migration is necessary only when there exists some light load processor(s) and some heavy load processor(s). Defining only the two states is not sufficient, however, because of processor thrashing and *state woggling.* State woggling means that the state swings between the adjacent states. It is necessary to define a third state, the *normal load* state between the light state and the heavy state. A processor in the normal state implies that it requires no migration. Two quantities must be defined which may be used as thresholds to specify three load states of a processor.

Load measurement

A dynamic load balancing algorithm concerns the dynamic changing of the processor load. The *processor load* indicates the utilization of the capacity of a processor (degree of "busyness"), which is usually determined by three major factors : operating system strategy, workload distribution and processor characteristics. The response time which a process experiences on a processor is an ideal measurement of the processor load. One approach to defining the load is to measure the increase in response time of a given process when it is run on the current processor over the response time needed if it is run on a completely idle base processor. However, this definition is not practical because many unmeasurable and time-varying parameters are involved, such as process execution time and memory requirement. Consequently, the response time is process dependent and is unknown information until the process is served. Thus, a load estimation method to measure the load based on the current status of the processor is needed. The method should be efficient and effective.

The operating system may estimate the current load based on certain measurable parameters, such as the number of processes in the system, mean execution time, etc. Therefore, the *estimated load* of a processor at any instant is process independent. The measurement of the estimated load for a single server queue with FCFS discipline is derived as follows [NiLi82]: Once a new process arrives, the number of processes in the processor, say N, can be measured. If N is equal to zero, the estimated response time for the new arrival is simply its own actual execution time. However, before the completion of the new arrival, the service time is unknown. The estimated response time is its mean service time, E[S]. If N is greater than zero, there is one process in service with other N-1 processes waiting. The estimated response time for the new arrival, in addition to its own mean service time, must include remaining service time of the process in progress and the service times of all other N-1 processes. Formally, we have the estimated processor load (EL) for the new arrival process

$$\begin{aligned} EL(N) &= E[S] && \text{if } N = 0 \\ EL(N) &= N{*}E[S] + E[S^2]/2E[S] && \text{if } N > 0 \end{aligned}$$

where E[S] and E[S^2] are mean and variance of service time distribution, respectively. The second term of the second equation is the remaining service time of the process being served. As the above equations show, the estimated load is proportional to the number of processes in the processor. For different disciplines, the equations of the estimated load are different. However, they are functions of the number of processes in the processor.

The number of waiting processes of a processor is a measurable parameter, while a processor can be modeled as a single server queue. The length of the queue may easily be partitioned into several parts by applying predefined threshold values. These parts are used to represent different load states. Even though the queue length does not take the size of the processes into account, it is an applicable parameter for load estimation.

A load estimation program may also be written which runs forever in a time-sharing system. Each time the program receives its CPU turn, it calculates the time interval between the last visit and the current visit to the CPU. The time difference is recorded and the program waits for its next visit. Based on the value of time difference, the processor load may be estimated. The greater the time difference is, the heavier the processor load is.

Estimation of processors' internal states

In looking at the conventional multiprogramming operating system, we can figure out some necessary modifications in order to facilitate dynamic process migration in a network environment. Multiprogramming is an operational technique allowing several processes to share the computer resources. In such a system, the processor, CPU, divides its time between a variety of processes. These processes are arranged for convenience in a number of queues, some containing processes that are free to run and some containing processes that are halted for one reason or another because they are waiting for some peripheral actions to be completed. A new process becomes a unit of work to the operating system when submitted to the system. Usually, an initial program structure is created

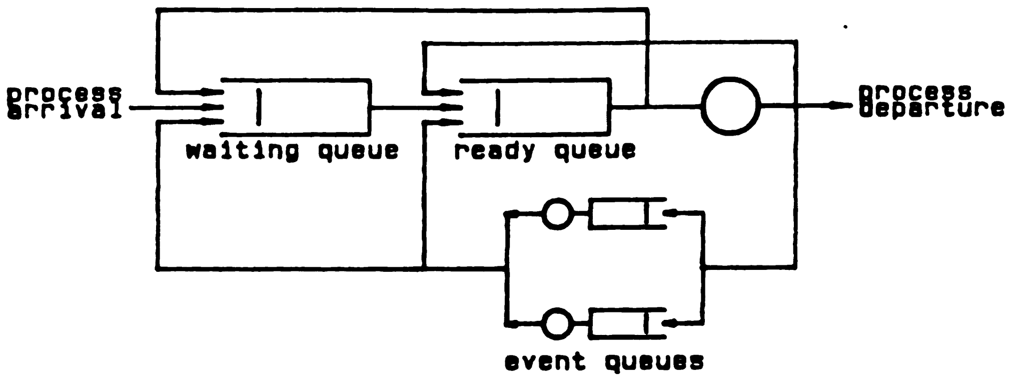


Figure 3-7 A queueing model of a conventional multiprogramming system

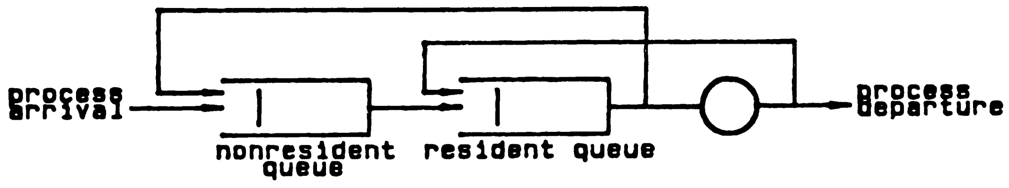


Figure 3-8 A simplified queueing model of a conventional multiprogramming system

for the process and a *control block* built to hold control information, such as registers, program counter, pointers to other control blocks, etc., which are dynamically maintained for that process. The operating system is responsible for maintaining a list of processes in various stages of passage through the system. The *waiting queue* contains processes that were already presented to the system, but were not allocated space in the main memory. The *ready queue* contains processes actually loaded, which are either subsequently given control of the CPU by a scheduler routine or removed to the waiting queue to leave space for a high priority process. There are many *event queues* for different events. It is possible that the memory of a process in the event queue is de-allocated. After the event is satisfied, the process is transferred to the ready queue if it is still allocated memory or is transferred to a waiting queue if the memory is de-allocated. Figure 3-7 shows the queueing model of a conventional multiprogramming system.

The model can be simplified by considering only the factor of memory allocation and eliminating event queues as depicted in Figure 3-8.

In a network environment, the queueing model of each processor has to be changed slightly to accommodate the connection of incoming and outgoing channels.

As mentioned above, the number of processes in the processor is selected as the measurement of the processor load. In order to reduce on-line calculation time, the three load states are statically predefined by dividing the waiting queue into three queues, *resident queue*, *threshold queue*, and *schedule queue*. For those processes which are allocated physical memory space or scheduled or partially executed, and are unlikely to be migrated, the resident queue is set up. The light-load state is defined as the state in which the number of waiting processes is less than or equal to the length limitation of the resident queue.

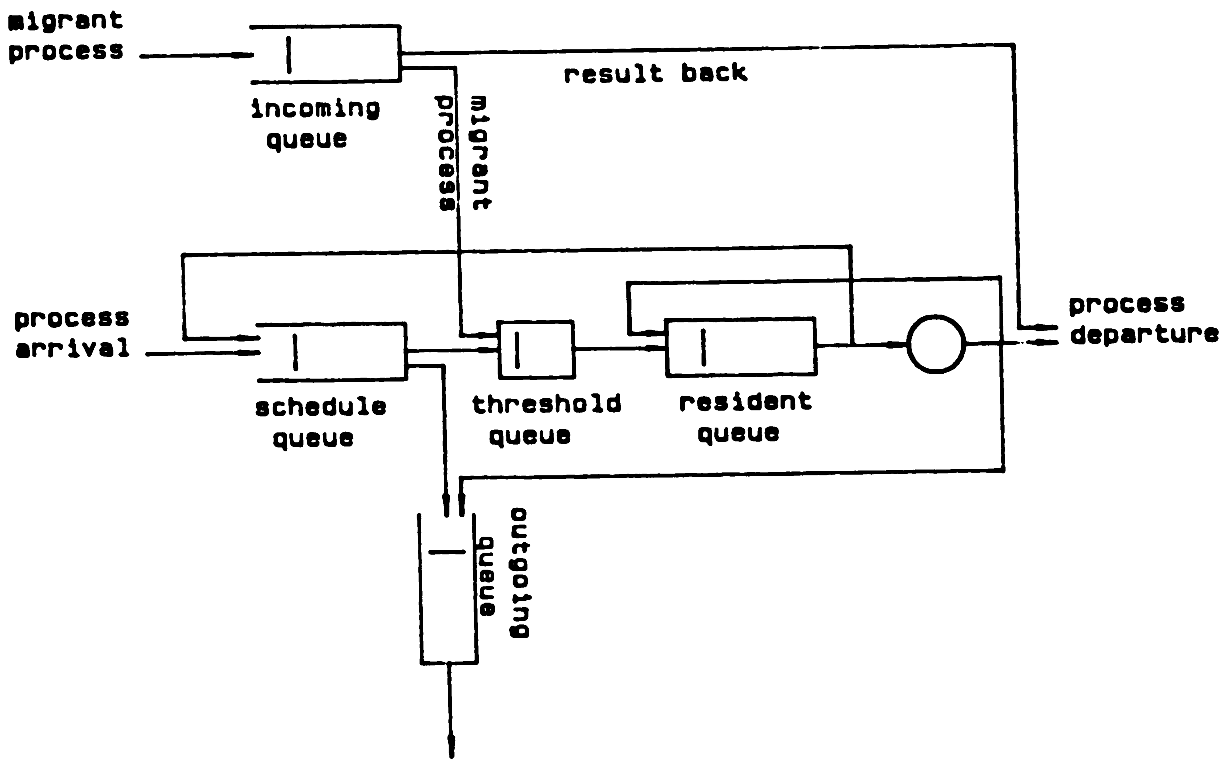
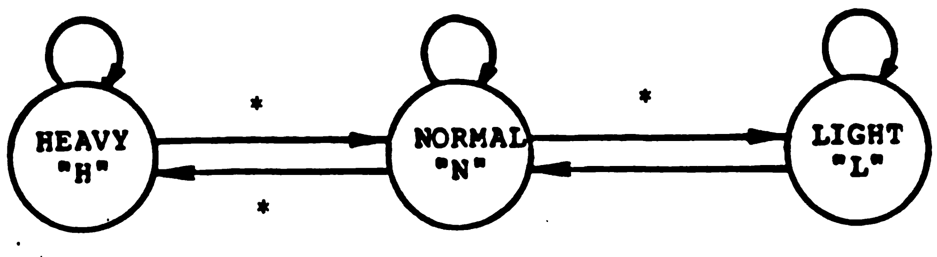


Figure 3-9 A completed queueing model for the drafting algorithm

The processes in the schedule queue are the *migratable processes*. If the schedule queue is not empty, the processor is in the heavy-load state. The threshold queue is located between the resident queue and the schedule queue. The length limitation of the threshold queue affects the definition of both normal and heavy states, and therefore affects the migration control algorithm. Thus, it is an important parameter of the algorithm. The choice of the length limitation of the threshold queue will be discussed later. Figure 3-9 shows the completed model which corresponds to the definition of the processor's internal states described above for the processors working in the network environment.

Since the load is defined as the number of processes in the processor, the load evaluation time is the instant of a process departure event or a process arrival event. If the load evaluation procedure is invoked only at fixed intervals, the state change from H-load to L-load or from L-load to H-load is possible if the time period is too long. A long load evaluation period is undesirable because it does not reflect its current processor load. In this study, the state transition is assumed such that a processor can capture its load change from H-load (or L-load) to N-load before the state changes to L-load (or H-load) as show in Figure 3-10. Detailed explanation of * will be presented in Section 5.2.



*: load-change message may be sent

Figure 3-10 The state transition diagram of the processor's load

CHAPTER 4
PERFORMANCE EVALUATION

Performance influences all aspects of computer systems or protocols, from their design to management. Hence, it is important to study in detail a system's or protocol's performance characteristics at every stage of design. It is especially important to be able to analyze and predict the performance of a newly designed system or protocol since its actual performance is unknown.

4.1. Methods of Performance Evaluation

Queueing network models are frequently used methods for analyzing distributed systems or computer networks. Various analytical, numerical, and simulation techniques are devloped to obtain exact or approximate solutions of queueing models. Notable among them are the power iteration method [WaRo66], generating function approach [Klei75], product form solution [GoNe67, BCMP75], and recursive solution technique [HeWC75]. A queueing model with adaptive or dynamic scheduling methods can not be analyzed exactly [LaWo81]. [ChKo77] studied various load balancing strategies for a distributed system with two processors. The only load balancing strategy they could not analyze was the dynamic one. As a result, an approximation method was used. In their approach, the communication delay was not considered. The analysis became even more complicated if the communication delay was considered.

The main reason for being unable to exactly analyze such a queueing network was due to the loss of the Markovian property. For a dynamic scheduling discipline, the arrival to a processor became state dependent and was no longer a Poisson process. Each processor then was modeled as a G/M/1 queue, where

G indicated a general distribution and could not be obtained.

Recently, Timed Petri Nets have been considered to be powerful tools for modeling and performance specifications of communication protocols. Timed Petri Nets were first introduced by Ramchandani [Ramc73] by associating firing times with the transitions of ordinary Petri Nets, in order to study their steady-state behavior. Garg and Ozsu [Garg85, Ozsu85] successfully used it in the field of performance specification of distributed protocols. However, they also experienced limitations of the method. It was hard to express an explosion of states for complex protocols. The model worked under the assumption that the data transfer time was constant. However, this assumption was not suitable for our situation in which both the control message exchanges and the process migrations were nondeterministic.

The last solution to measure the performance of distributed systems with dynamic scheduling methods was to conduct simulation experiments. For networks, the important aspects were the topology of networks, the bandwidth of channels, the conflicts in access to channels, the unpredictable nature of message passing , and piggyback technique, etc.. Simulation experiments were time consuming, but they accommodated the characteristics of a real distributed system. Section 4.2 details the structure of the simulator used.

4.2. Simulator

A simulation program was implemented in the Pascal and C languages based on the queueing network [SaCh81]. The simulation program was driven by events, using the regenerative method. The major points of the simulator are described as below.

Mapping concurrent events into sequential behaviors

Dynamic process migration in a distributed computing system deals with several concurrent processes which include not only the local external arrival processes in every processor but also the communication processes as well as the processes migrated by internal process migration. Computer systems are usually seen as discrete state systems. The discontinuous state transitions are called *events*. The mapping of concurrent behaviors into the sequential language is based on the assumption that there is no more than one event happening at exactly the same time over the system. Therefore, even though concurrent processes act in different processors of the system, the time order maps them into a sequential event chain. Thus, a sequential language may be used to mimic concurrent behaviors. An event list arranged by the time at which the event occurs is used to keep track of the events over the system. The event list is a doubly linked list. Process arrival and process completion are the two major events. Other events mainly depend on the algorithm to be simulated.

Data structure for constructing the topologies of networks

There are a variety of topologies of computer networks and a variety of models of the computer processors. A simple linked list is selected to unify the presentation of networks. The simple linked list consists of a set of processor records associated with a neighbor list. The neighbor list in each processor record expresses the connection between processors. For example, if processor 1 has connections to processor 2 and processor 5, then there are two neighbor records in the neighbor list of processor 1. Each processor is modeled as an open queueing network. Users easily specify the global parameters and system model by a data file without modifying the program. These data structures make the simulation program flexible for any kind of network topology and any kind of connection of queueing models within a simulated processor.

Data gathering

The simulation should gather those data generated in the steady-state condition of the real-world system. A statistical approach, called the regenerative method, which satisfies this requirement, is employed. The basic concept underlying this approach is that a simulation run can be divided into a series of cycles such that the behaviors of the system during different cycles are both statistically independent and identically distributed. The statistics within each cycle comprise a series of independent and identically distributed observations analyzed by standard statistical procedures [HiLi80].

The regenerative method is based on the properties of Markov processes. The future behavior of a Markov process is dependent only on the process's current state. Each time a Markov process enters a specific state, called a ***regeneration state***, the process has the same expected future behavior, and the Markov process is regenerated. The periods between successive entrances to the state are known as the ***regeneration cycles***. If there is an ability to observe enough regeneration cycles, then the ***central limit theorem*** is applied to find the mean value of the observations. The theorem says that if random variables are independent and identically distributed, their sum tends toward a normal distribution as the number of random variables becomes large.

A principal advantage of the regenerative method is that if the simulation is initiated in a regenerative state, then it is assumed that simulation is initialized in an equilibrium condition. The principal difficulty with the regenerative method is in finding a frequently occurring regeneration state. Usually the regeneration state is selected to be the occurrence of a new arrival with no customers left. At this point, future arrivals are completely independent of any previous history, and the state occurs very frequently.

Confidence interval and statistical formulas

The confidence interval and the statistical formulas are the important design bases of the simulator. The *confidence interval* of a parameter is a range of value 2c within which the sample data of the parameter being estimated is almost sure to lie [Lind76]. The confidence interval is used for indicating the reliability or precision of an estimate process. Correspondingly, a *confidence coefficient* α is chosen to indicate α percent of the samples lying in the confidence interval, which affects the width of the interval.

The notations declared below are used for the statistical formulas.

2c -- confidence interval of the parameter

α -- confidence coefficient of the parameter

μ -- mean of the parameter

σ^2 -- variance of the parameter

σ -- standard deviation of the parameter

$\bar{\mu}$ -- sample mean which is the estimate of the parameter

$\bar{\sigma}^2$ -- sample variance of the parameter

n -- number of samples

By the central limit theorem, the sample mean $\bar{\mu}$ tends to be normally distributed with mean μ and variance $\frac{\sigma^2}{n}$ as n grows larger. The minimum number of samples to be taken so that the error between the μ and its estimate $\bar{\mu}$ is tolerable may be expressed by the means of probability terms as

$$P[\,|\bar{\mu}-\mu| < c\,] \geq \alpha \tag{4-1}$$

The central limit theorem also tells us that

$$\begin{aligned} P[\,|\bar{\mu}-\mu| < c\,] &= P[\mu - c < \bar{\mu} < \mu + c] \\ &= \Phi\left(\frac{(\mu+c)-\mu}{\frac{\sigma}{\sqrt{n}}}\right) - \Phi\left(\frac{(\mu-c)-\mu}{\frac{\sigma}{\sqrt{n}}}\right) \end{aligned}$$

$$= \Phi(\frac{c\sqrt{n}}{\sigma}) - \Phi(\frac{-c\sqrt{n}}{\sigma}) \qquad (4\text{-}2)$$

Thus, (4-1) can be expressed as

$$\Phi(\frac{c\sqrt{n}}{\sigma}) - \Phi(\frac{-c\sqrt{n}}{\sigma}) \geq \alpha$$
$$\Phi(\frac{c\sqrt{n}}{\sigma}) - \Phi(\frac{-c\sqrt{n}}{\sigma}) \geq \frac{1+\alpha}{2} - \frac{1-\alpha}{2} \qquad (4\text{-}3)$$

Let $\frac{c\sqrt{n}}{\sigma}$ be chosen so that the probability of a random variable with the standard normal distribution falls within $(\frac{-c\sqrt{n}}{\sigma}, \frac{c\sqrt{n}}{\sigma})$ is α. In other words, let the following equation be true.

$$\Phi(\frac{c\sqrt{n}}{\sigma}) = \frac{1+\alpha}{2}$$
$$\Phi^{-1}(\frac{1+\alpha}{2}) = \frac{c\sqrt{n}}{\sigma} \qquad (4\text{-}4)$$

Then

$$c = \frac{\Phi^{-1}(\frac{1+\alpha}{2})\,\sigma}{\sqrt{n}} \qquad (4\text{-}5)$$

reveals that the confidence interval c may be estimated by giving the confidence coefficient α. However, the value σ in (4-5) is unknown so that it must be replaced by the estimate of the sample variance of the normal distribution [Lind76]

$$\bar{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{i=n}(s_i - \bar{\mu})^2$$
$$= \frac{1}{n-1}(\sum_{i=1}^{i=n} s_i^2 - n\,\bar{\mu}^2) \qquad (4\text{-}6)$$

where s_i is the value of the sample data.

If a parameter X estimated is obtained by calculating a statistic Y for a time period of regeneration cycle T, then

$$E(X) = \frac{E(Y)}{E(T)} \qquad (4\text{-}7)$$

When n complete cycles are generated during the simulation run, the data gathered are $Y_1, Y_2, \ldots, Y_n$ and $T_1, T_2, \ldots, T_n$ for the respective cycles, and $\overline{Y} = \frac{1}{n}\sum_1^n Y_i$, $\overline{T} = \frac{1}{n}\sum_1^n T_i$. The *sample variances* are

$$\bar{\sigma}_y^2 = \frac{1}{n-1}\sum_{i=1}^{i=n}(Y_i - \overline{Y})^2$$

$$= \frac{1}{n-1}\sum_{i=1}^{i=n} Y_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{i=n} Y_i\right)^2 \qquad (4\text{-}8)$$

$$\bar{\sigma}_t^2 = \frac{1}{n-1}\sum_{i=1}^{i=n}(T_i - \overline{T})^2$$

$$= \frac{1}{n-1}\sum_{i=1}^{i=n} T_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{i=n} T_i\right)^2 \qquad (4\text{-}9)$$

The *combined sample covariance* is

$$\bar{\sigma}_{yt}^2 = \frac{1}{n-1}\sum_{i=1}^{i=n}(Y_i - \overline{Y})(T_i - \overline{T})$$

$$= \frac{1}{n-1}\sum_{i=1}^{i=n} Y_i T_i - \frac{1}{n(n-1)}\left(\sum_{i=1}^{i=n} Y_i\right)\left(\sum_{i=1}^{i=n} T_i\right) \qquad (4\text{-}10)$$

The *total sample variance* is

$$\bar{\sigma}^2 = \bar{\sigma}_y^2 - 2\left(\frac{\overline{Y}}{\overline{T}}\right)\bar{\sigma}_{yt}^2 + \left(\frac{\overline{Y}}{\overline{T}}\right)^2 \bar{\sigma}_t^2 \qquad (4\text{-}11)$$

Finally, the confidence interval of the parameter of interest is

$$c = \frac{\Phi^{-1}\left(\frac{1+\alpha}{2}\right)\sqrt{\bar{\sigma}^2}}{\sqrt{n}\,\overline{T}} \qquad (4\text{-}12)$$

given the confidence coefficient of the parameter.

Response time estimation

Mean response time $\bar{t}$ is obtained from the response time values of each individual process. The response time of a process is computed as the difference between its departure and arrival times. Instead of storing all response time values, the simulation keeps a running estimate of $\bar{t}$, which at any instant is the value of $\bar{t}$ up to that instant [Ferr78].

Let $\overline{t_i}$ denote the response time averaged over the first i processes completely processed by the simulator and let t_j be the response time of the jth process executed. we have

$$\overline{t_i} = \frac{1}{i} \sum_{j=1}^{i} t_j \tag{4-13}$$

and

$$\begin{aligned}\overline{t_{i+1}} &= \frac{1}{i+1}(t_1 + t_2 + \cdots + t_i + t_{i+1}) \\ &= \frac{1}{i+1}(i * \overline{t_i} + t_{i+1}) \\ &= \frac{i}{i+1}\overline{t_i} + \frac{1}{i+1} t_{i+1}\end{aligned} \tag{4-14}$$

Thus, to compute $\overline{t_{i+1}}$, the values of t_1, t_2, ... t_i were not needed. It is sufficient to know $\overline{t_i}$, i and t_{i+1} for given $\overline{t_1} = t_1$.

Similarly, the variance of successive observations about current mean $\overline{t_i}$ may be kept and updated during simulation without storing all the individual response times.

Structure of simulator

The entire simulator consists mainly of two parts. One deals with the general aspects, such as the structure of the event list, the initialization of the parameters, the description of the topology of the network, the definition of the queueing model of a processor, and common procedures for manipulating these data structures. The other part concerns the algorithm to be simulated. The program is separated into two parts so that the first part is suitable for different algorithms, and the second part is modified. The first part is affected by the second part.

For example, the simulation model is determined by the algorithm simulated while the first part must build up the model. A more flexible method of accomplishing a different data structure in one program is to use a separated data file. Two data files are available for this purpose. One is called *datagloba*, the other is called *datastatn*. The "datagloba" file contains global parameters, such as the number of processors in the system, the arrival rate at each processor, etc. The "datastatn" file describes the parameters of each queue and the structure of the queueing network in each processor. Any queueing network must be defined by specifying the predecessor and the successor of each queue. If the number of predecessors is more than one, then a priority is assigned to each predecessor. The various classes of processes go to different successors of a queue via different paths. The simulation model is described by the file and the file is the input of the first part. The second part is dedicated to the algorithm itself. This separation makes the simulator flexible. For different algorithms, the general part is not necessarily modified.

The whole program consists of 12 separate files organized to perform four functions: input/output, job manipulation, common subroutines (including a debugger), and algorithm simulation. Input/output includes initstatn.c, initpara.c, presult.c; job manipulation includes newjob.c, arrive.c, complete.c; common subroutines include event.c, pack.c, common.c, show.c; and algorithm.c is the part to be modified for different purposes.

The main program contains a while loop shown as follows.

```
initialize data structures and parameters;
while (stopping criterion is not satisfied) {
    remove the first event;
    switch (event kind) {
    case JOBARRI :
        switch (queue kind) {
          case SINK :
                statistic procedure;
                break;
          default :
                put job into the queue;
                break;
          }
```

```
                break;

            case JOBDEPA :
                complete a job;
                send the job to the successor queue;
                break;

            default :
                special events;
                break;
            }
    }
```

Figure 4-1 The main() of the simulator

It controls arrival and departure to and from each queue as well as other special events, such as the token arrival in the token ring network, and the minus quantum for the round robin discipline.

4.3. Simulation Model

A distributed computer system consists of N independent processors connected in an arbitrary fashion by a communication network. The topology of the communication network is varied in different studies. In Chapter 5, a point-to-point topology is adopted; in Chapter 6, a token-ring network is chosen; in Chapter 7, a CSMA/CD-based environment is considered. Processors in the system have the same processing capabilities; that is, a process may be processed from start to finish with any processor in the system. Processors employ the same kind of operating system, so they comprise a homogeneous distributed system.

Processes arrive at processor i, $i = 1, 2, ..., N$, according to a time-invariant Poisson process with rate λ_i. The total process arrival rate is denoted by R. A process arriving at processor i may be either processed at processor i or migrated through the communication network to another processor j. A migrant process from processor i to processor j receives its service at processor j

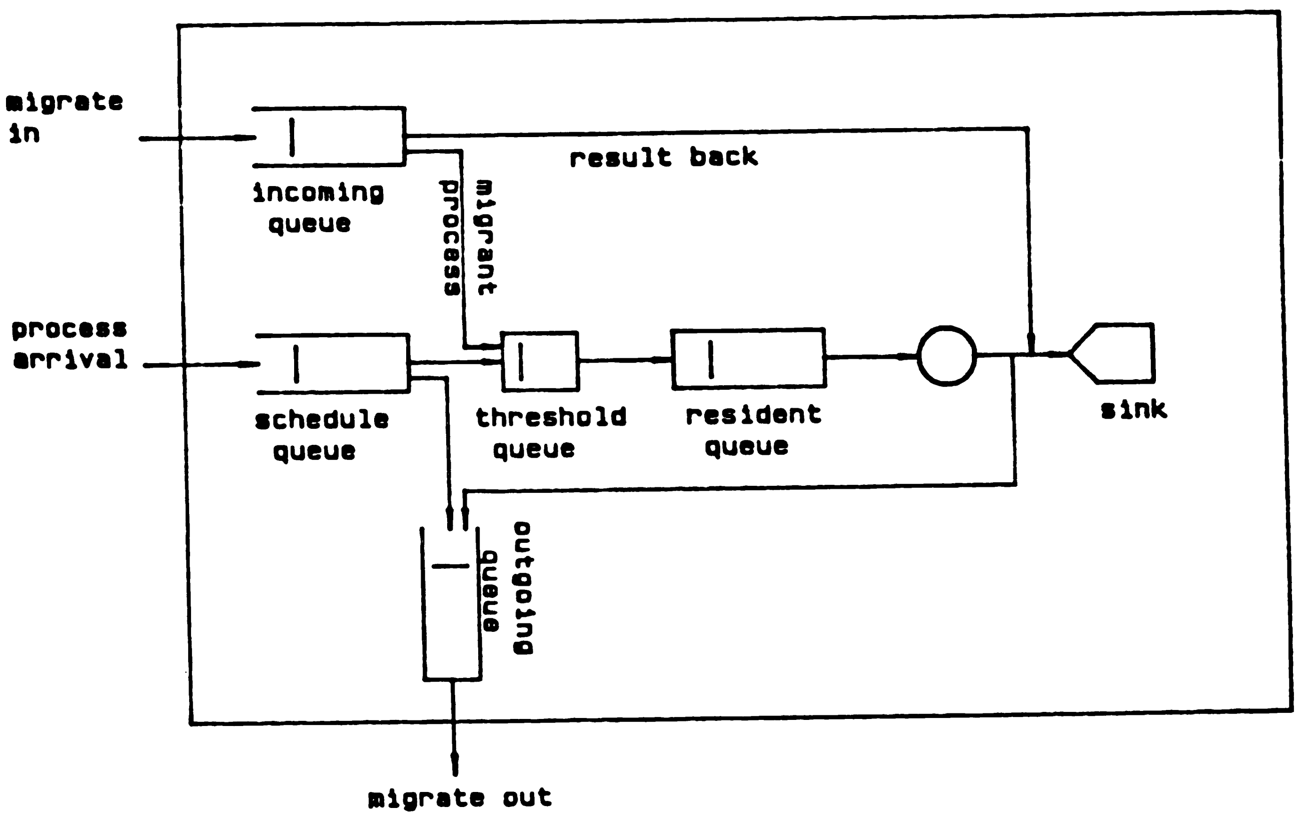


Figure 4-2 A simulation model of a processor

and is not migrated to other processors again. After the process is processed at processor j, a result is returned to the original processor, where no more processing is required. A decision of migrating a process depends on the state of the system as determined by the drafting algorithm to be described in the next chapter.

The processor model used for simulation is shown in Figure 4-2. The open queueing network of the processor has 6 queues and a sink. The schedule queue is the port communicating outside the computer system. All new processes entering the processor come into the queue in question. The incoming queue is the port which communicates with other processors within the system, and is used to buffer migrant processes. The migrated processes may either be the migratable processes or the result migrated back. The former is sent to the threshold queue, the latter is sent to the sink. The outgoing queue is the port that accumulates all messages and migrant processes before they are sent out to other processors. The schedule queue is for queueing processes waiting for scheduling. The processes in the schedule queue may be scheduled to be processed either on the local processor or on the remote processor. Both the threshold queue and the resident queue are finite queues and the length limitations are parameters of the drafting algorithm, which determines the load state of the processor as described in Chapter 3.

CHAPTER 5
A DISTRIBUTED DRAFTING ALGORITHM

Based on the study of design considerations and the processor state, a drafting algorithm with respect to dynamic process migration has been developed. The drafting algorithm has appropriately answered questions raised by the distributed control algorithms mentioned in section 1.2. and gives efficient and effective control of process migration as proved by the results of the simulation of a point-to-point local network.

5.1. Drafting Algorithm Concept

The drafting algorithm is divided into two phases. In the ***coupling phase***, a light-load processor attempts to find a partner (called a couple) from one or more heavy-load processor(s) by mutual selection. In the ***drafting phase***, the heavy-load processor either migrates a most appropriate migratable process to the light-load processor or sends a ***too-late*** message to indicate there is no longer any migratable process when it tries to migrate a process to the light-load processor.

The drafting algorithm is defined based on the following assumptions:

(1) it is executed in a homogeneous distributed system; it allows a process to be migrated only once, and does not allow preemption;

(2) there is no user specified process migration in the system;

(3) lower level communication is reliable;

Coupling phase

Whenever a light-load processor invokes the drafting algorithm, it looks first at the ***load table*** (section 5.2) which contains the most recent information of the load of all other processors in the distributed system to check if there is any eligible processor to participate in the action of load balancing. If such a processor exists, it is called a ***candidate processor***. There may be zero, one or many candidate heavy-load processor(s).

(1) 1-to-0 case (one light-load processor, no heavy-load processor)

Since there is no heavy-load processor, the drafting processor goes to sleep to prevent useless information exchange with respect to migration. When a processor's state becomes heavy, it broadcasts a heavy-load state to other light-load candidate processors. This message wakes up any sleeping light-load processor(s).

(2) 1-to-1 case (one light-load processor, one heavy-load processor)

The light-load processor knows there is only one heavy-load candidate processor. A couple is found between these two processors already. However, it is possible the same heavy-load processor also is the candidate processor of other light-load processors. Therefore, the same heavy-load processor couples with many light-load processors. A N-to-1 (many light-load processors, one heavy-load processor) case results over the whole network. From the light-load processor's point of view, however, the coupling phase has been done.

(3) 1-to-N case (one light-load processor, many heavy-load processors)

A light-load processor may find that there are many heavy-load candidate processors when examining its load table. It must select only one as its couple. It sends out a ***draft request*** message to all heavy-load candidate processors, requiring them to reply their ***draft age***. The light-load processor selects the heavy-load processor with the oldest draft age (heaviest load or process in scheduling queue waiting the logest) as its migrating couple. From the light-load processor point of view, the situation becomes a 1-to-1 case. Its coupling phase is completed. A two-step control message is needed for every algorithm invoked in this case. Of course, from the whole distributed system point of view, the situation may be the more complicated N-to-N case (many light-load processors, many heavy-load processors).

Migrating phase

The drafting algorithm begins its migrating phase immediately following its coupling phase. The light-load processor sends a ***draft standard*** to its couple processor. This represents a range of acceptable draft ages for migratable processes from the heavy-load processor. The heavy-load processor migrates a migratable process to the light-load processor if it finds a migratable process which satisfies the draft standard.

As mentioned above, the 1-to-1 case may represent a N-to-1 situation over the whole distributed system. The selected migratable process in the heavy-load processor may have been served by the local processor or may have been migrated to another light-load processor when the heavy-load processor receives the draft standard message from a light-load couple processor. Then, it either migrates another migratable process which still satisfies the draft standard or sends a *too-late* message to those light-load processors sending the draft standard message. The algorithm is briefly illustrated in Figure 5-1.

5.2. Design Parameters

Many design parameters should be determined for the drafting algorithm. The major parameters are discussed as follows.

The length of the resident queue

The length limitation of the resident queue determines the light load state of the processor, which is an important parameter for triggering the drafting

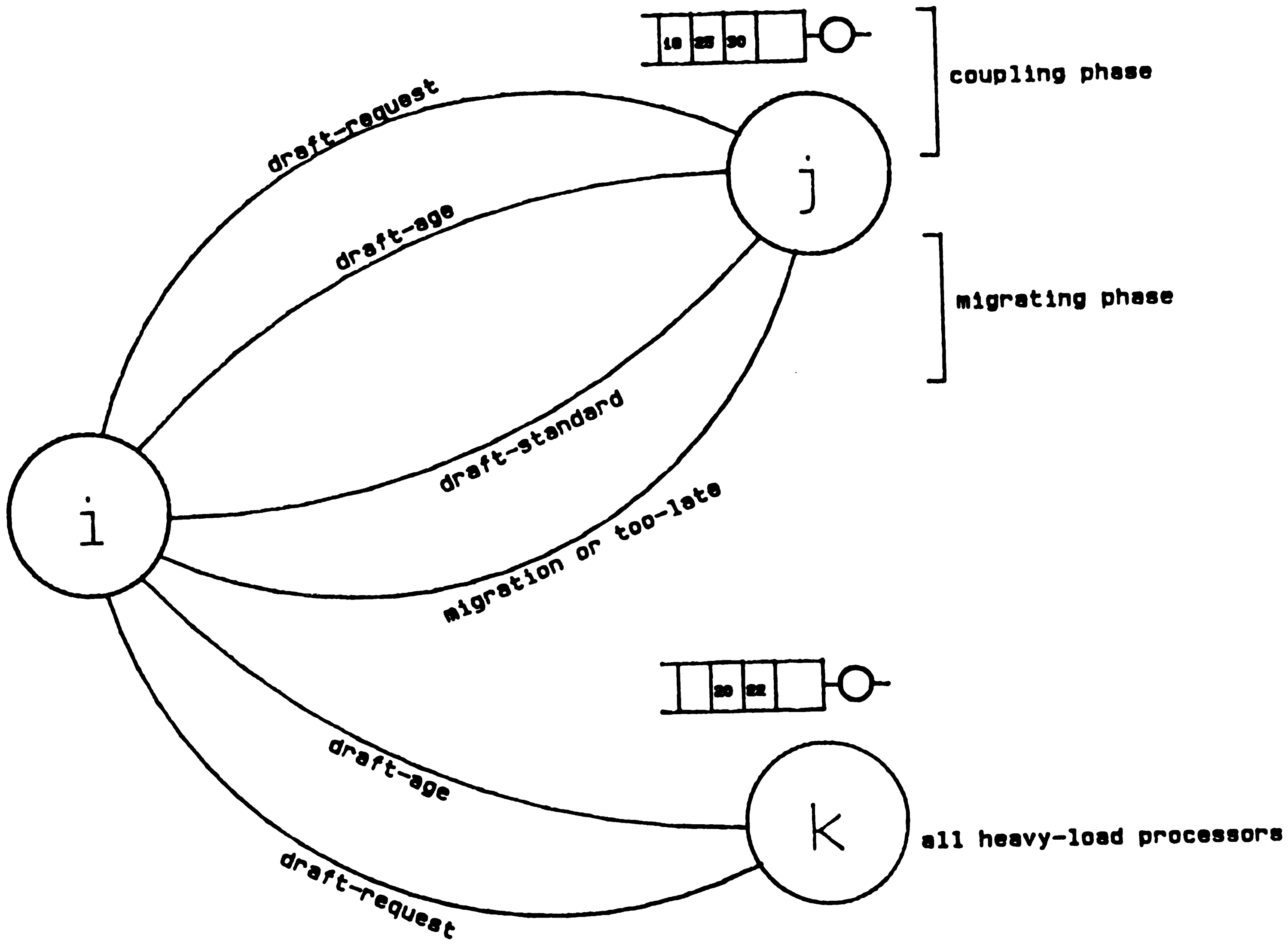


Figure 5-1 A brief illustration of drafting algorithm

algorithm. In single processor computers, the value of 1 with the FCFS discipline, or the degree of multiprogramming with a round-robin discipline, is the measurement of the processing capacity of the computer (processor or memory). This measurement is expressed by symbol L. Therefore, the load situation is called light if the number of waiting processes in the processor is less than or equal to the value of L. The value of L is a parameter which depends on the arrival rate, the departure rate, the discipline of the processor, the size of the physical memory, and the size of the processes as well as the attributes of other resources.

The Q value

The length of the threshold queue is a subtle parameter, as mentioned previously. We select it based on two basic principles. The migration must have some benefit; that is, the migrant process must expect a shorter response time than if it were not migrated. This is called the *time equality principle.* Suppose the migrated process requires more time before completion in the remote processor than if served by the local processor. This migration is not only worthless for the migrant process but also increases channel traffic.

Another principle is that when the migrant process reaches the light-load processor, it must not make the processor become heavily loaded. It is called the *state equality principle.* The value of Q is examined based on the definition of the time equality principle as follows.

The time taken by a typical migration from processor i to processor j includes:

(1) the time spent in making a decision which migratable process would be migrated.

tha

sor

sho

the

the

whe

at w

that

cesso

the r

depa

is

The a

perioc

It is assumed to be ignored because it is very short.

(2) the time period for seizing the communication channel and for the time that the migrant process is queued in the migrant buffer of the remote processor.

For a point-to-point network, the time to seize a channel may be assumed short enough to be ignored. For a broadcast network, however, the length of the time period is the unpredictable length w.

(3) the time spent in the channel.

Under the assumption the channel is not saturated, the time required by the transmission is

$$T_3 = \frac{S_i}{c} \tag{5-1}$$

where S_i is the average size of the migrant process from station i; c is the rate at which the channel transmits messages. Thus, the migration requires time

$$T_{mig} = w + T_3 \tag{5-2}$$

(4) the service time of the migrant process in the remote processor j.

The processor j was in light load at the beginning of the migration; assume that the queue length is L_j. However, during the time period of T_{mig}, the processor j's queue length is changed due to local arrivals and departures. If A is the random variable of the new arrivals, and D is the random variable of the departures, the probability that k processes will arrive in the time length T_{mig} is

$$P_k = e^{-\lambda_j T_{mig}} \frac{(\lambda_j T_{mig})^k}{k!} \tag{5-3}$$

The average number of processes entering the processor j during the time period T_{mig} is

$$E(A) = \sum_{k=0}^{k=\infty} kP_k = \sum_{k=0}^{k=\infty} ke^{-\lambda_j T_{mig}} \frac{(\lambda_j T_{mig})^k}{k!} = \lambda_j T_{mig} \tag{5-4}$$

Similarly, the average number of processes departing from processor j during the time period T_{mig} is

$$E(D) = \mu_j T_{mig} \tag{5-5}$$

When the migrant process reaches the queue of processor j, the queue length of processor j will be

$$L_j + E(A) - E(D) + 1 \tag{5-6}$$

where L_j is the length of resident queue of processor j and 1 represents the migrant process. Its response time, therefore, is

$$T_4 = \frac{L_j + E(A) - E(D) + 1}{\mu_j} \tag{5-7}$$

(5) the time for sending the result back to the originating processor i from the remote processor j depends on the unpredictable time period w and the time T_5. The time period w includes the time period for seizing the communication channel and the time period the result is queued in the migrating buffer of the originating processor. The time period T_5 is the time required by the result transmission which can be expressed as

$$T_5 = \frac{R_j}{c} \tag{5-8}$$

where R_j is average size of the result produced in processor j and being sent back to the originating processor i; c is the rate at which the channel transmits the result.

Therefore, the total time spent in migrating a process to a remote processor from an originating processor is

$$\begin{aligned} T_{rm} &= 0 + w + T_3 + T_4 + w + T_5 \\ &= 2w + T_3 + T_5 + \frac{L_j + \lambda_j T_{mig} - \mu_j T_{mig} + 1}{\mu_j} \end{aligned} \tag{5-9}$$

Now, if the migrant process were instead left in the queue in the originating processor, then it would have to spend the response time (include the service time) before it was completed. The average response time is

$$T_{ro} = \frac{L_i + Q_i + 1}{\mu_i} \tag{5-10}$$

where L_i is the length of the resident queue, Q_i is the length of the threshold queue of processor i and 1 represents the migrant process itself.

According to the time equality principle definition, the following equation exists.

$$T_{rm} \leq T_{ro} \tag{5-11}$$

Substituting the T_{rm} of (5-9) and T_{ro} of (5-10) into (5-11), we have

$$(2+\frac{\lambda_j-\mu_j}{\mu_j})w + (1+\frac{\lambda_j-\mu_j}{\mu_j})T_3 + T_5 + \frac{L_j+1}{\mu_j} \leq \frac{L_i+Q_i+1}{\mu_i}$$

$$Q_i \geq (1+\frac{\lambda_j}{\mu_j})w\,\mu_i + (\frac{\lambda_j}{\mu_j})S_i\frac{\mu_i}{c} + R_j\frac{\mu_i}{c} + \frac{\mu_i(L_j+1)}{\mu_j} - L_i - 1$$

$$Q_i = f\,(w\,,\frac{1}{c},S_i\,,R_j\,,\mu_i\,,\lambda_j\,,\mu_j\,) \tag{5-12}$$

where $1/c$ is contributed by the process migrating along the channel and w is contributed by the process waiting time in the migrant buffer and in entering the channel. Thus, Q is dependent on local arrival rate, local departure rate of the processors, average size of the migrant process, the average size of the result after completing the migrant process, the channel capacity and the channel traffic. Obviously, in the case that the migrant process cannot reach its remote processor until it passes n intermediate processors, the Q value is also a function of n. The unpredictable value of w makes the value of Q intractable. But, the equation (5-12) may be used to estimate an approximate value of Q under some assumptions. For example, if the value of w is assumed constant, the value of Q gives the bottom line of the benefit at which migration is worth carrying out. Only a properly selected value of Q guarantees that the migrant process gains a better response time than if it is executed locally. In the simulation, we consider Q as a parameter to study the effect of the value of Q. The value of Q also may be found based on the definition of the state equality principle.

Draft age

Intuitively, the draft age means, for example, the arrival time of processes. In fact, the age could be any kind of parameters for negotiation purpose, such as waiting time, priority, precedence relationship, special resource requirements, memory requested, etc. Based on these parameters, a light-load processor selects one of the heavy-load processors as its expected couple.

Draft standard

According to parameters specified by draft age and its own capacity, the light-load processor selects an age among all the received ages as the draft standand and sends out it as a response of the negotiation. A draft standard is required to:

(1) ensure that the effort of the information exchanges would not be wasted;

(2) secure FCFS fairness over the whole system

due to dynamically changed system conditions. Otherwise, if a light-load processor only sends a message to inform the originating processor that it has been selected as the couple, the originating processor might not migrate any process because the selected migratable process either has been served by the local processor itself or has been migrated to another processor. It is smart to send an age-range, i.e., a draft standard, to the originating processor, upon which it may migrate another migratable process whose age falls into the draft standard. The wider the draft standard is, the higher probability that another satisfied migratable process is able to be found. However, the wider draft standard does not ensure fairness because the originating processor may migrate a very new arrival still within the age-range and eventually serve it before other processes with earlier arrival times than the "very new arrival". Thus, the FCFS is destroyed.

The draft standard is an important parameter for the drafting algorithm. A simple example is illustrated in Figure 5-1. The example indicates two queues. One of them is in processor j, another is in processor k. In the queue of the processor j, there are three migratable processes. The first migratable process has a waiting time of 30, the second migratable process has a waiting time of 25 and the third has a waiting time of 18. In processor k's queue, there are two migratable processes. They have the waiting times of 22 and 20, respectively. Processor j is the couple of the processor i. If the drafting standard is selected as the second oldest age among the received ages, say 22, then the second migratable process of the processor j could be migrated in case the first migratable process has been served. The global FCFS discipline is preserved. If the drafting standard is picked wider, say 14, thus the third migratable process of the processor j might be migrated. Consequently, the global FCFS is destroyed because the third migratable process of the processor j is younger than the two migratable processes in the processor k. Therefore, the selection of the draft standard is important and it may only partially preserve the global FCFS discipline if its range is wide.

Load table and its updating strategies

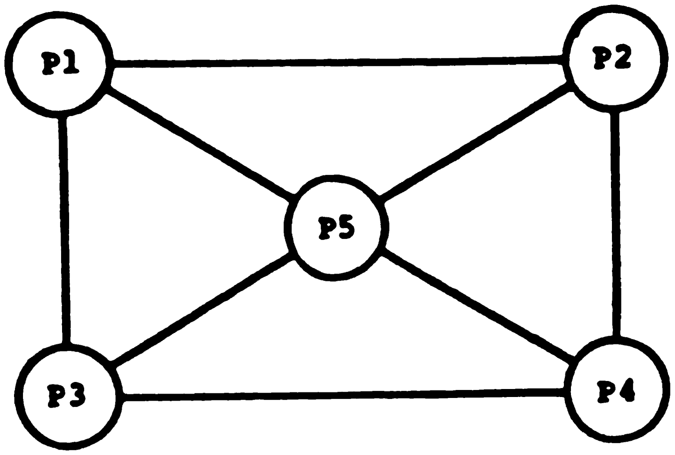
When a processor is in the light-load state, it invokes the process migration algorithm. Without having current load information of other processors, the light-load processor would have to send a migration request message to all candidate processors. All candidate processors would reply with their current load states. The light-load processor could select the one with the heaviest load state as the migration source processor.

However, this approach will cause some problems. First, it creates too much traffic (one-to-all communication every time). Second, after sending messages out, the light-load processor must wait for the replies from the heavy-load processors. However, it does not know how many heavily loaded processors exist and how long it must wait, especially, if all candidate processors are in light-load. The only way to end the waiting is to rely on a specified time interval. To specify a time-out period delays the migration.

To alleviate this problem, every processor has an associated load table. Every entry in the load table records the load state and other parameters of a candidate processor. What parameters should be designed into the load table depends on the implementation of the process migration algorithm and the configuration of the distributed system. An example of a load table is shown in Figure 5-2.

However, updating the load table creates more traffic. Communication along channels may fall into two different types. One is called a *single message*, such as the state broadcast messages. The processor sending a single message out can immediately execute any other instructions without waiting for reply messages from the receiving processors. The second is called a *chain message* in which the sending processor expects the reply messages to return before it executes other related instructions. The single message obviously takes less time and produces less traffic than the chain messages because a chain message may be postponed by other unrelated actions using the channel.

Even though the broadcast messages belong to the single message category, the broadcast of load state messages must be organized in a well-defined



(a) A sample network topology

PROCESSOR	LOAD	ATTRIBUTES
P1	H	. . .
P2	N	. . .
P3	L	. . .
P4	H	. . .

(b) A sample load table of P5

PROCESSOR	LOAD	ATTRIBUTES
P2	L	. . .
P3	L	. . .
P5	H	. . .

(c) A sample load table of P1

Figure 5-2 An example of load table

manner, instead of broadcasting every state change, in order to reduce channel traffic. After a processor changes its load state from light to normal states, its state may be further changed to light again or to heavy, or may remain normal. Similarly the state changes from heavy to normal. To avoid state woggling, the normal state is not broadcasted immediately. It may cause ***load table inconsistency.*** The inconsistency means that the state recorded in the load table is different from the current state of the corresponding processor. For example, when a processor is heavily loaded, it broadcasts 'H' to all its candidate processors. When it becomes normally loaded, it does not broadcast the normal state. The state in the load table of other processors is still 'H'. A lightly loaded processor may send a draft request message to it. Even all the state transitions are broadcasted, due to communication delay, the inconsistency may still exist. Thus, the problem becomes how to reduce the effect of inconsistency.

The main purpose of the load table is to provide information for a L-load processor to find H-load candidate processors. To avoid state woggling between N-load and L-load, a load-change message carrying L-load is broadcast to all candidate processors only if there is a N-L load-change and the processor's previous state before N-load was H-load. Thus, a L state in a load table indicates the corresponding remote processor's load is either light or normal. In either case, the remote processor will not be drafted. In other words, the broadcast rule does not cause any trouble for the L-load processors.

To deal with state woggling between H-load and N-load, a load-change message carrying "H-load" is broadcast to all candidate processors if there is a N-H load-change and the processor's previous load before N-load was L-load. However, this may inhibit H-load processors from being drafted. Imagine that if all the H-load candidate processors change their load from H-load to N-load, the drafting processor will fall sleep until any candidate processor becomes heavy again. If the algorithm does not broadcast the N-H load change, the drafting processor will wait forever. Thus, whenever there is a N-H load-change, the candidate processor must inform the drafting processor regardless of what the candidate processor's previous load state before N-load could be. Three approaches are considered below to handle the N-load carefully.

(1) Always Broadcast. In this approach, a processor always broadcasts N-H load-change and H-N load-change to all its candidate processors. In this approach, all load tables are consistent in ignoring the case caused by the communication delay as mentioned above.

(2) Piggybacking. A N-H load-change is broadcast to all candidate processors if the processor's load before N-load was L-load. A H-N load-change will be piggybacked with the draft age message to the drafting processor. In the meantime, a one bit PG-field is needed in the load table. 1 in the PG-field of the load table indicates the corresponding candidate processor has a N-load state; otherwise, it is 0. Whenever there is a N-H load-change, a load-change message which carries H-load information is sent to those candidate processors with the corresponding PG-field being 1.

By piggybacking with the draft age message, an extra load-change message is eliminated. Since the draft age message is sent in a one-to-one fashion, an inconsistency among different load tables may occur. However, this inconsistency is not important as far as drafting is concerned. One may argue that the PG-field in other load tablne is redundant because only an L-load processor may ask for draft age. Thus, whenever there is a N-H load-change, a load-change message is sent to those candidate processors with L-load in the load table. However, this may cause a serious load table inconsistency. For example, a processor Pl in the L-load is informed via piggybacking that processor P2 is in the N-load. Pl then changes its load to N-load and then to H-load. Through broadcasting, P2 records that Pl is in H-load. When P2 has N-H load-change, it does not inform Pl. When Pl changes its load to L-load again, it thinks that P2 is still in N-load and excludes P2 from the drafting list.

To compare always-broadcast and piggybacking approaches, we assume a particular processor has c candidate processors. Among these c candidate processors, m of them are in L-load. For a certain time interval between two consecutive appearances of the L-load, an original heavy-load processor has alternately k H-N load-changes and (k-1) N-H load-changes. For always broadcast, the processor must transmit (2k-1)c load-change messages. For piggybacking, without broadcasting the H-N load-change, the processor receives a draft-request message from a L-load processor and replies with a draft age message piggybacked with its current N-load. Assuming that, during a N-load period, the processor receives, on the average, m draft-request messages from those L-load processors, k(2m) messages will be generated. Also assume there are m 1's in the PG-field. Thus another (k-1)m N-H load-change messages will be generated. Consequently, the always broadcast strategy is better if the following inequality is satisfied.

$$\frac{(2k-1)c}{2km+(k-1)m} < 1$$

$$\frac{(2k-1)c}{(3k-1)m} < 1$$

Ignoring both the 1 in the numerator and the denominator, the approximate calculation shows that always broadcast strategy is better if m/c is greater than 2/3; otherwise, piggybacking is better. As a result, a csombination of both strategies is suggested.

(3) Mixed Updating. When a processor has a L-N followed by a N-H load-change, use an always broadcast approach if its current value of m/c is greater

than 2/3; otherwise, adopt the piggybacking approach. The processor must keep that approach until the next L-N followed by a N-H load-change.

In a broadcast network, such as Ethernet, a one-to-many transmission takes only one message and all processors in the network are neighbors to each other. In this situation, the always broadcast strategy should be adopted.

The mechanism used to implement the updating of the load table includes a *last-broadcast-load* (LBL) field in the load table. When piggybacking to the normal state information 'N' to the light-load processor to change the state recorded in the light-load processor's load table, the last-broadcast-load (LBL) field of the corresponding light-load processor in the load table of the normal-load processor also changes to 'N'. If the state of the current normal-load processor is changed back to heavy again, the normal-load processor needs only to send 'H' messages to those processors whose LBL is 'N' in the load table. Piggybacking can both 'broadcast' state message and save traffic. Thus, the probability of the state woggling is not only reduced but also load table inconsistency is prevented. Whenever a processor broadcasts its current state, it also records the broadcasting state in the LBL of its load table. If the load state changes from normal to light (or heavy), the processor checks the LBL field. If the LBL is different from the current load state, then the current load state is broadcasted to all candidate processors and the LBL is changed to the current load state. Otherwise nothing is done.

5.3. Simulation Results on Point-to-Point Networks

There are many parameters considered in the selection of design parameters (section 5.2.). We selected them in the simulation as follows.

length of the resident queue	1
length of the threshold queue	a parameter to study
draft age	oldest age
draft standard	second oldest age

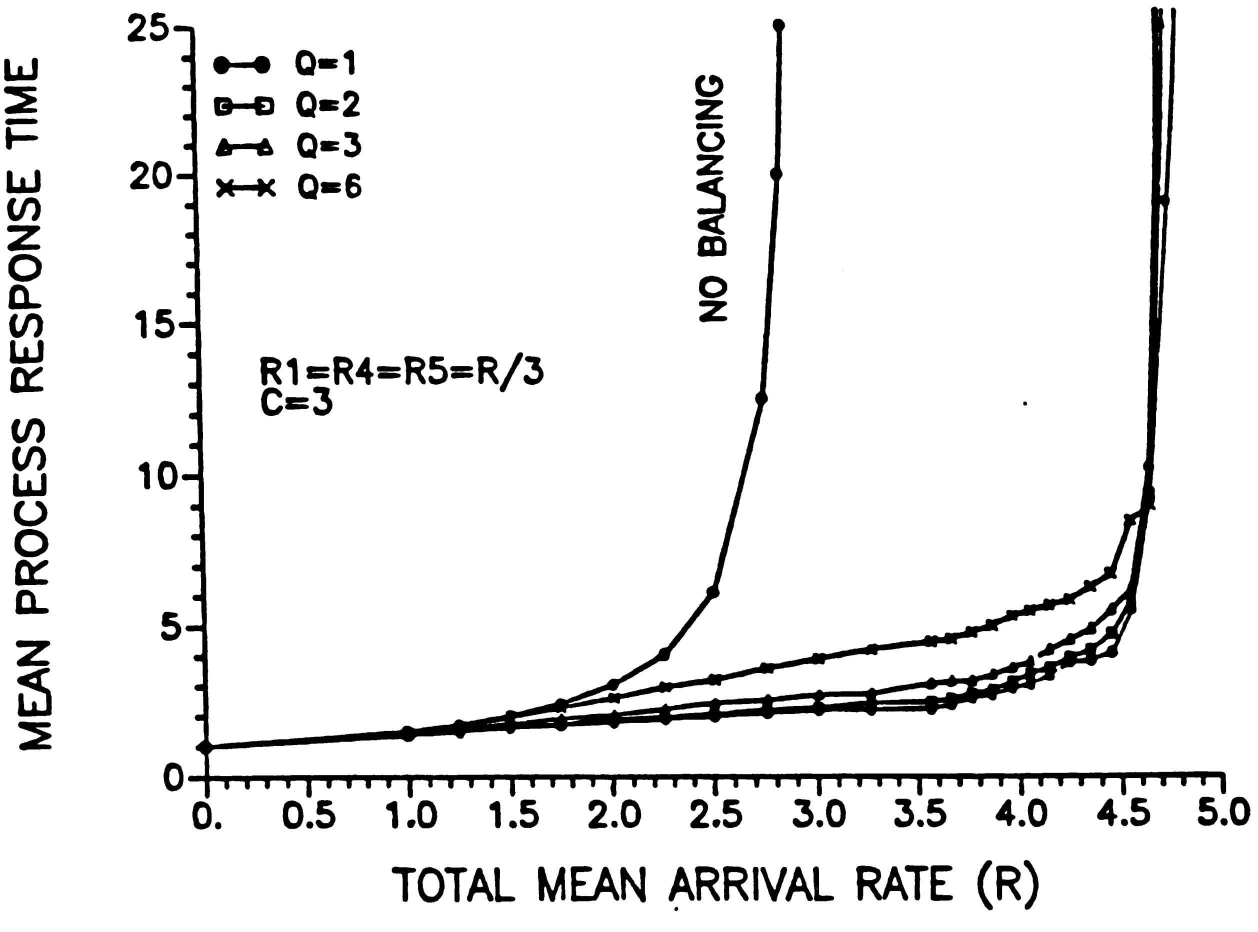


Figure 5-3 The effect of the arrival rate

arrival rate λ_i	the argument of the simulation
service rate μ_i	normalized as 1

The results of simulation provide a comparison of different parameters with and without load balancing.

The effect of different arrival rate λ_i

Two different arrival rate configurations were examined. One is $\lambda_1 = \lambda_4 = \lambda_5 = R/3$ and $\lambda_2 = \lambda_3 = 0$ which is known as *non-probabilistic balancing.* The other is $\lambda_i = R/5$, i = 1,2,3,4,5 which is known as *probabilistic balancing.* For two configurations different values of λ_i were studied. The results shown in Figure 5-3 and Figure 5-5 reveal that the improvement is greater in the former case than in the latter.

The effect of channel bandwidth c

The influence of the channel bandwidth lies beneath the surface. The process migration for load balancing is a time critical action. The transmission rate is usually a bottleneck. The ratios of c/μ_i and c/λ_i are important. The values of the ratios not only influence the time delay of the process migration but also complicate the state transitions. This is why some output of the response time corresponding to some values of λ_i looks strange.

Increasing the channel bandwidth will decrease the average process response time as shown in Figure 5-4. In the figure, for each channel transmission rate, c, the corresponding optimum value of Q is used for providing a fair comparison. When c is large enough, the drafting processor is likely to stay in the light load state upon the arrival of the migrant process. If c is small, the drafting processor may be no longer in the light load state due to its local arrivals. Theoretically, if c approaches infinity, then the optimum value of Q will be zero and the system performance will be very close to the result of an

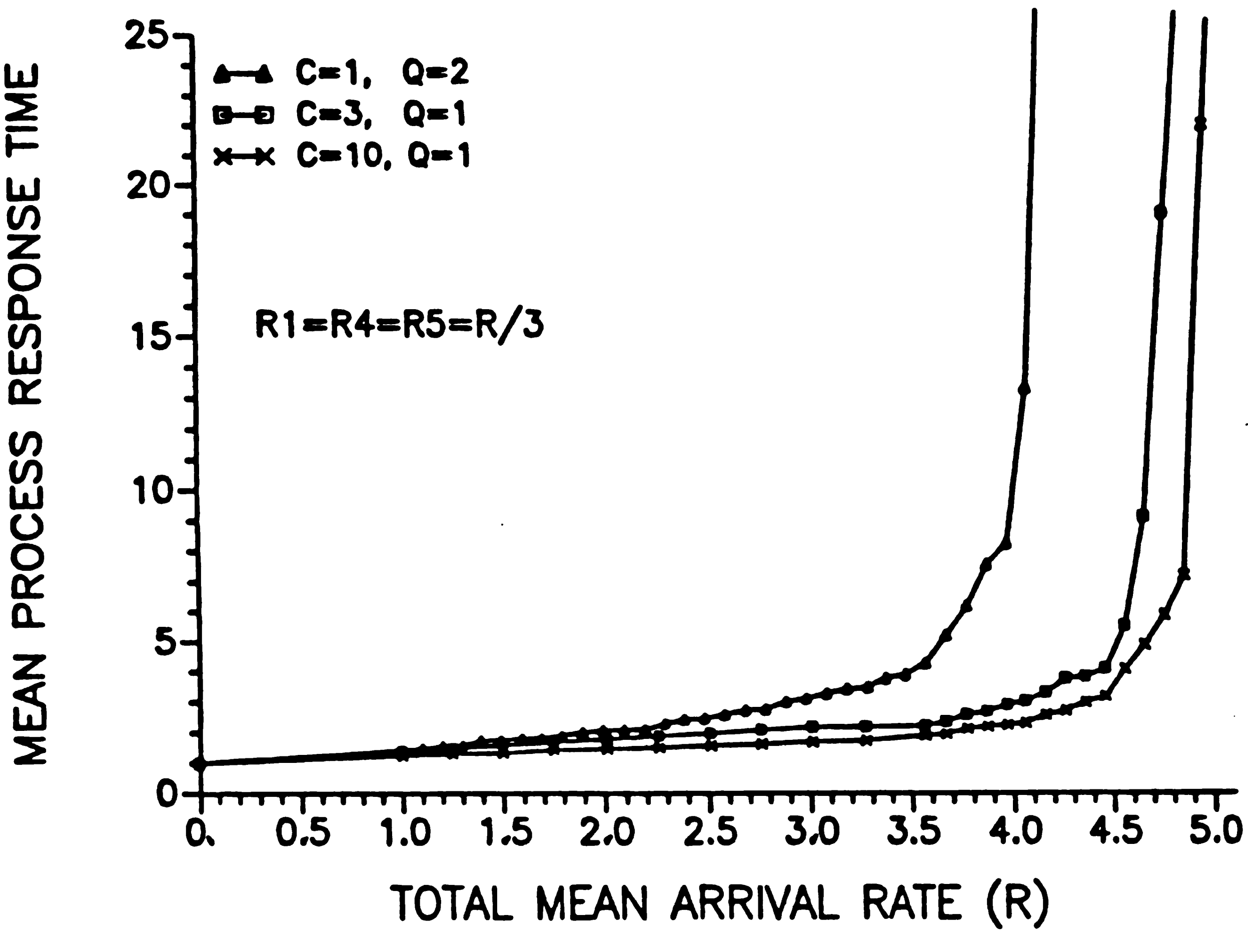


Figure 5-4 The effect of the channel bandwidth

M/M/N queueing system. Also, the number of too-late messages will be significantly decreased and the number of control messages will become smaller as c increases. This is due to the fact that the message or state information is more current. Consequently, the lower channel bandwidth produces worse performance. Therefore, dynamic process migration is suitable for local area networks rather than wide area networks.

The effect of Q values

The value of Q determines the point at which process migration starts. The bigger the value of Q, the longer the queue length at which process migration starts. When Q is sufficiently large, the system behavior will approach the case of no load balancing effort. When Q is small, the processor will have a higher probability to reach heavy-load state and generate more migrant processes. However, there may be many unnecessary migrant processes. As Figure 5-5 shows, the simulation result reveals that as long as Q is not too far from its optimum value, the performance difference is not significant. In Figure 5-5, the optimum value is Q=1. But the performance of Q=2 and Q=3 are very close to that of Q=1. With Q=6, the performance grows worse. Since the value of Q is closely related to the processor load, this implies that the definition or measurement of processor load will affect the system performance, but does not require much accuracy. It is, therefore unnecessary to find a very accurate method of measuring processor load.

The difference between static balancing and dynamic balancing

With the same network topology, it is assumed that all processors have the same arrival rate ($\lambda_i = R/5$ for all i). In this case, the system is statically (or statistically) balanced. From this point of view, there is little migration. However, in the dynamic balancing, process migration is still possible because the

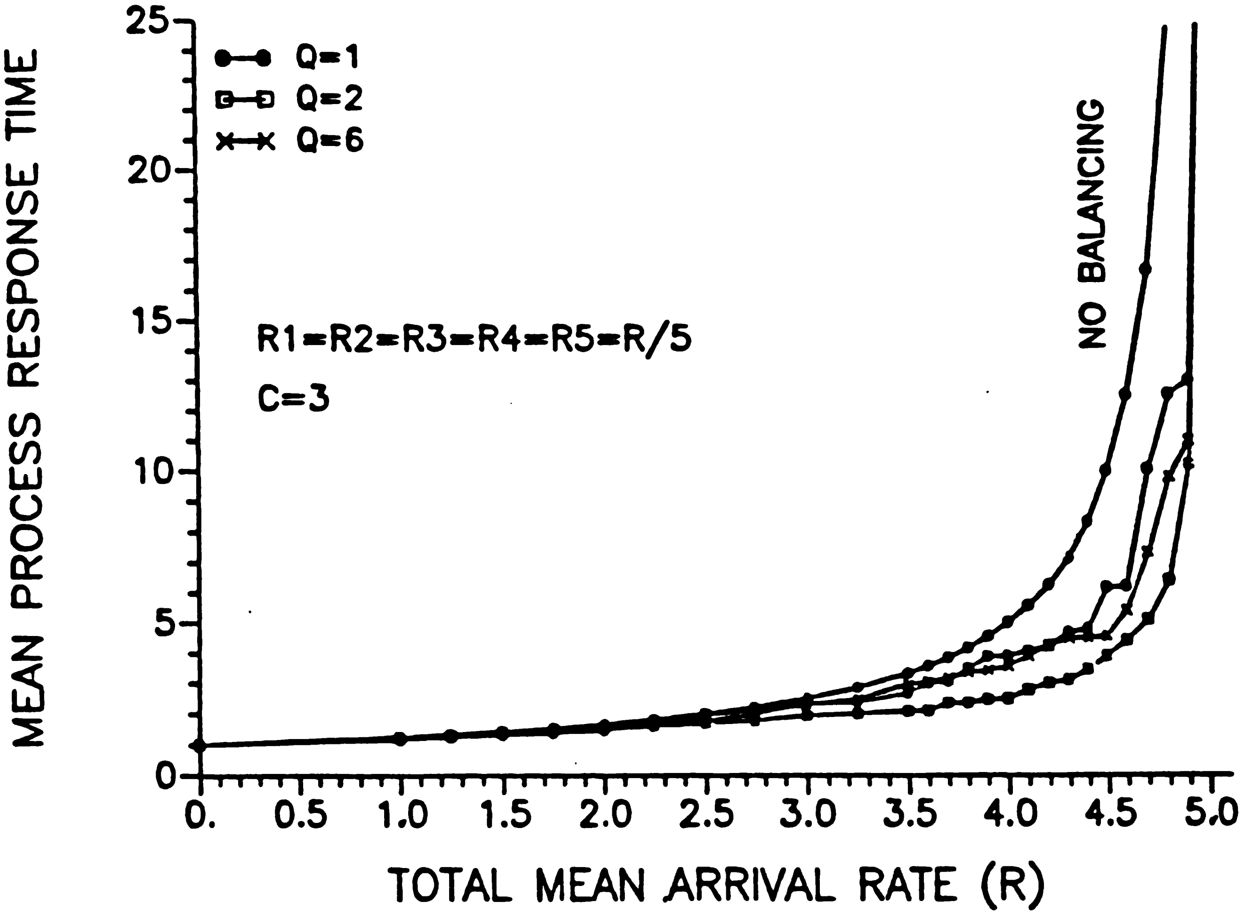


Figure 5-5 The difference between static and dynamic balancing

number of processes in each processor is not dynamically the same. In the case of c=3, Q=2 provides the best performance, as shown in Figure 5-5. When Q=1, there are too many unnecessary migrant processes. Furthermore, these migrant processes are further delayed due to local arrivals at the remote processor. When R is large, even Q=6 has a better performance than that of Q=1. When R is even greater, all processors are likely to reach the heavy-load state and the number of migrant processes decreases. The number of control messages generated is also less than the previous workload pattern.

5.4. Formal Description of the Drafting Algorithm

Three major concurrent processes are described in a Pascal-like notation. Other concurrent processes, such as updating the load table and load measurement, are detailed in section 5.2. and not included here. While most of the statements are standard constructs we do need a couple of additional constructs in order to describe the algorithm. These constructs are the event variables and multicasting.

An event variable (EV) is a variable which is used to notify a process of a state change or is interrogated by a process to determine if a state is in effect. Each EV variable takes one of two values, CURRENT or NOTCURRENT. An EV variable is manipulated by the following procedures:

wait(EV); (* If EV is NOTCURRENT, then the process is blocked until EV becomes CURRENT. This operation does not change the value of EV *)

signal(EV); (* Set EV to CURRENT *)

clear(EV); (* Set EV to NOTCURRENT *)

current(EV); (* A function which returns TRUE if EV is CURRENT, otherwise FALSE is returned *)

Multicasting provides a way of indicating 1-to-many communication. Three multicasting primitives are used.

multi_cast_and_wait(proc_list, message, reply_list, status); (* All processes whose IDs are listed in the ProcList are sent the message. The sending process is blocked until all processes reply or until a timeout occurs *)

receive_any(source, message); (* The process which executes this statement is blocked until a message is received. Source is a variable which receives the ID value of the message-sending process *)

multi_cast(proc_list, message); (* All processes whose ID's are listed in the ProcList are sent the message. The sending process is not blocked *)

In the following description, the purpose of some procedures are apparent from their names. Some event variables, such as L_load, H_load, N_load, load_change, heavy_msg_arrive, and load_measure_event, may be signaled or cleared by other concurrent processes, such as load measurement and load table updating. These event variables as well as the load_table are accessed by all processes.

Each processor waits until the load state becomes light, then the processor executes the process send_draft_request to couple with a heavy-load processor.

```
process send_draft_request;
var
  num_heavy : integer;
  process_id : process_id_type;
  standard : draft_standard_type;
  status : (fail, ok, timeout);
  proc_list : ptr_of_process_list;
  age_list : ptr_of_age_list;
  mig_reply : migration_reply_record;
begin
  loop
    wait(light_load);
    repeat
      any_heavy(num_heavy, proc_list);
      if num_heavy = 0 then
        wait(normal_load or heavy_msg_arrive);
        clear(heavy_msg_arrive);
        if current(N_load) then goto L1
        else goto L2;
        endif;
      endif;
      if num_heavy > 1 then
        multi_cast_and_wait(proc_list, "draft_request",
                            age_list, status);
        if status <> fail then
          calc_draft_standard(proc_list, age_list, standard, process_id,
                              found_heavy, status);
          if not found_heavy then (* load changed *)
            wait(N_load or heavy_msg_arrive);
            clear(heavy_msg_arrive);
            if current(N_load) then goto L1
            else goto L2;
            endif;
          endif
        else (* fail *)
          exception_handling;
        endif
      else (* num_heavy = 1 *)
        standard := 0;
        process_id := first(proc_list);
      endif
      if NOT current(L_load) then goto L1;
      endif;
      (* send draft_standard message *)
      send_and_wait(process_id, standard, mig_reply, status);
      if status = ok then
        if mig_reply.kind = process then (* receive a process *)
        load_migrate_queue(mig_reply.pcb);
        else (* receive a too_late message *)
          update_load_table(process_id, mig_reply.load);
        endif;
      else (* status is fail or timeout *)
        exception_handling;
      endif;
    L2 : until (NOT current(L_load));
    L1 : forever;
  end_process;
```

The processors in the heavy load state execute the following two processes to respond to the drafting request and the drafting standard.

```
  process respond_draft_request;
  var
    source : process_id_type;
    age : draft_standard_type;
  begin
    loop
    receive_any(source, "draft_request");
    calc_draft_age(age);
    send(source, age);
  forever
end_process;


process respond_draft_standard;
var
  source : process_id_type;
  pcb : process_control_block;
  mig_reply : migration_reply_record;
  standard : draft_standard_type;
  found : boolean;
begin
  loop
    receive_any(source, standard);
    select_process(standard, found);
    if found then
      mig_reply.kind := process;
      mig_reply.pcb := pcb;
    else
      mig_reply.kind := too_late;
      mig_reply.load := get_current_load;
    endif;
    send(source, mig_reply);
  forever
end_process;
```

5.5. A Comparison between Bidding and Drafting Algorithms

Both the drafting algorithm and the bidding algorithm are distributed control algorithms. As mentioned in chapter 1, a load balancing algorithm should consider : (1) when a waiting process can be migrated; (2) which waiting process should be migrated; (3) from where to where the migration will take place; (4) and how to gather the global status information. Table 5-1 summarizes the differences between the bidding algorithm and the drafting algorithm.

As illustrated, the bidding algorithm triggers a migration when a new arrival enters a processor in the system, regardless of how busy the processor is. Suppose there are N processors in the system. When a processor, processor i, receives a new arrival, it issues an (N-1) bid request messages to all other pro-

Table 5-1 A comparison of bidding and drafting algorithms

	bidding	drafting
when	a new arrival	light load
which	the new arrival	a selected migratable process
where	from the tail of a waiting queue to the tail of another queue	from the first in schedule queue to the tail of the threshold queue
how	gathering information instantaneously	by the load table

cessors in the system. Each of the (N-1) processors must respond one bid message back to processor i. After processor i selects winning processor j, processor i sends one winner notification message to processor j. Then processor j issues one accept or reject message to processor i. Consequently, every new arrival causes 2(N-1)+2 information exchanges.

On the other hand, the drafting algorithm triggers a migration when a processor is in the light-load state. Suppose there is an H heavy-load processor in the system at the moment when a light-load processor, say processor i, attempts to issue the drafting request. It then must issue an H drafting request message, and each of the H heavy-load processors must respond one draft_age message back to processor i. Processor i sends one draft_standard message to its couple processor. The total number of message exchanges is 2H+1. Since the value of H and the number of light-load processors are dynamically changed variables, the number of control messages generated by the drafting algorithm can only be measured by experimental simulation. For a point-to-point distributed system consisting of 5 processors, the number of control messages generated by the drafting algorithm for completing 10000 jobs is a few thousand for low arrival rates (0.5 to 2.5) and 10,000 to 32,000 for the moderate to high arrival rates (2.5 to 5.0). However, under the same condition the bidding algorithm must issue (2(5-1)+2)*10,000 = 100,000 control messages.

The bidding algorithm migrates the new arrival from the original processor to the remote processor. When the external load fluctuation is serious, the migration of the new arrival causes processor thrashing and unnecessary migrations, and cannot reach fairness in the global sense. The drafting algorithm migrates a migratable process which satisfies the drafting standard only. This kind of migration achieves fairness at the system level.

The bidding algorithm migrates a process from the tail of a waiting queue to another tail of a waiting queue. It may migrate a process from a heavily loaded processor to a less heavily loaded processor or from a lightly loaded processor to a more lightly loaded processor. The drafting algorithm usually migrates the first process in the schedule queue to a light-load processor; the value of Q (section 5.2.) guarantees the migrant process gaining a better response time than if executed locally.

Finally, the bidding algorithm gathers the global information instantaneously whenever a new arrival enters a processor. The drafting algorithm uses a load table to keep track of system status information. Each processor broadcasts its current state in a well-organized manner (section 5.2.). The piggyback further considerably reduces this overhead.

These features make the drafting algorithm efficient and effective. On the other hand, they cause the design of the drafting algorithm to be complicated.

CHAPTER 6

DYNAMIC PAIRING IN TOKEN RING NETWORKS

In a token ring network, all stations form a physical ring topology. Each station connects to the communication channel through a *ring interface processor* (RIP). The RIP has four communication ports: one connects to the input channel, one connects to the output channel, one passes information to the station, and one accepts information from the station, as depicted in Figure 6-1. Detailed operations of token ring networks can be found in [LiRo84, IEEE84]. In this chapter, a description of the token ring network operational principles and characteristics necessary for understanding our proposed distributed pairing algorithms for load balancing will be provided.

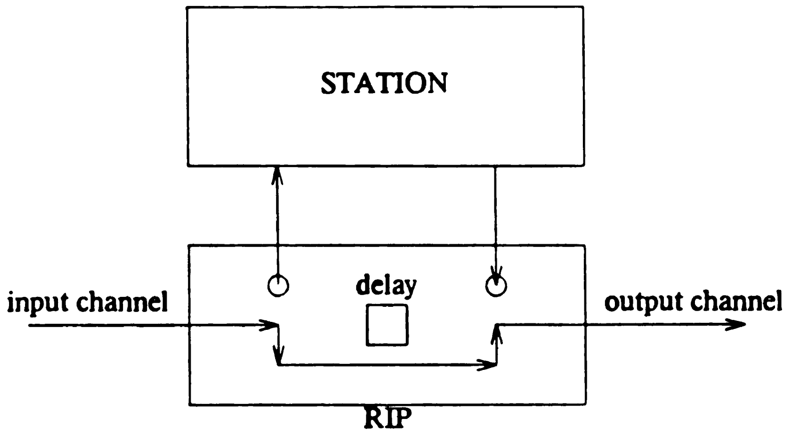
In order to avoid channel access conflict, a physical *token* circulates around the ring. The token is either *free* or *busy*. A station must seize a free token and then transmit the information frame. The token is not removed from the ring; rather, its state is converted from free to busy by the token holder and the frame follows the busy token circulating around the ring. A non-token holder station will copy the frame as it passes by the RIP if the station is the destination station. The frame is removed by the sending station (token holder) and the token holder also converts the token from busy to free. The next station downstream from the ring becomes the new token holder upon receiving the free token.

The RIP is an active device and plays an important role in a token ring network. The RIP has three operational modes. If the station is in failure or does not want to join the ring, the RIP will be in the *bypass mode*, directly connecting the input channel to the output channel (see Figure 6-1(a)). If the station is the token holder, the associated RIP will be in *transmit mode*. In this case, the input channel is connected to the input port of the station and the output channel is connected to the output port of the station (see Figure 6-1(b)). For a non-token holder station, its RIP will be in the *listen* mode. In this case, the RIP buffers the frame received from the input channel and reproduces the frame to put on the output channel (see Figure 6-1(c)). If the station is the destination station, a copy of the input frame is sent to the station. In the listen mode, the frame is delayed for at least one bit-time.

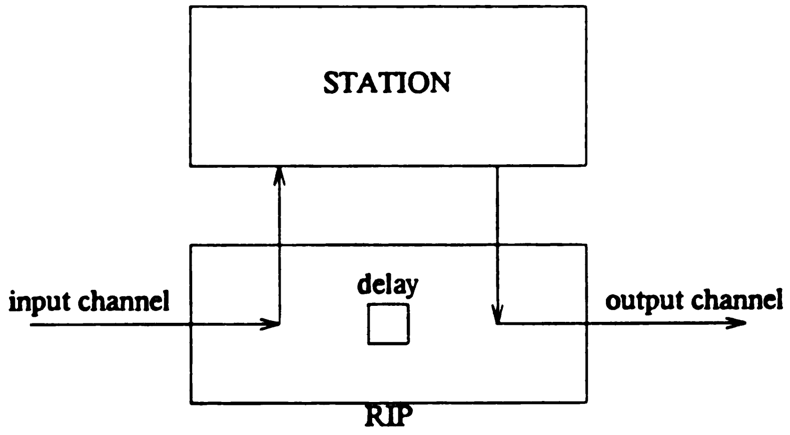
6.1. Characteristics of Token Ring Networks

In a token ring network, the time period between two consecutive free token visits to a given station is called ***token cycle time.*** The token cycle time is an important distribution function used to measure the performance of token ring networks [Rego85]. The time period during which the free token is passed from one station to the next station is called ***token passing time.*** This value is deterministic and is dependent on physical distance between these two adjacent stations, the transmission rate, and the channel propagation delay. The interval from the instant that a station receives the free token until it passes the free token to the next station is referred to as ***token holding time.*** Note that if the outgoing queue is empty, the token holding time is zero. Obviously, if there are N stations on the network, the token cycle time is the sum of token passing time and token holding time of all these N stations.

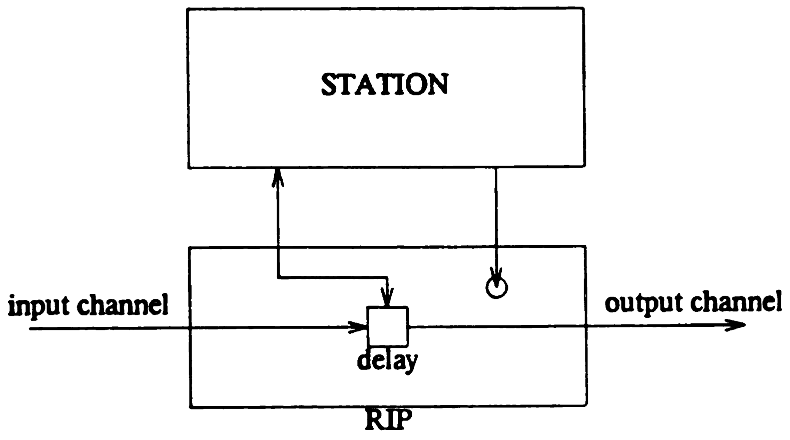
The distribution of token holding time is mainly determined by the number of ready-to-transmit frames in the station and the queue emptying discipline. The *queue emptying discipline* is defined as the maximum number of frames transmitted per free token visit and is classified into two categories. An *exhaustive* discipline allows each station to transmit all its ready-to-transmit frames before passing the token to the next station. This discipline results in a



(a) Bypass mode



(b) Transmit mode



(c) Listen mode

Figure 6-1 Architecture and operational modes of the RIP

long token cycle time and is not fair to those stations with a small number of frames. A *nonexhaustive* discipline allows the token holder to transmit at most n frames per token visit. In the standard token ring network, the value of n is one [IEEE84]. This provides a more equitable service to stations. The time period from when the token holder puts a frame on the channel and the token holder removes that frame from the channel is referred to as *frame cycle time.* This value is dependent on the physical length of the whole ring, the delay caused by each RIP, the number of stations, the channel propagation delay, the transmission rate, and the length of the frame.

Since a transmitted frame is removed by the sending station (the token holder), every station on the network is able to listen to the ongoing transmission. This makes the implementation of *multicast* (one-to-many) and *broadcast* (one-to-all) easy.

Another unique characteristic of the token ring network is its *bitwise changeable* feature. Every bit received by the RIP from the input channel may be changed and forwarded to the output channel. Communication traffic can be reduced by using this feature. For example, a separate acknowledgment frame must be transmitted by the receiving station for the purpose of a reliable communication in most of the network architectures. In the token ring network, an extra ACK bit is placed at the tail of a frame. The receiving station flips the ACK bit if the received frame is receives correctly. When removing the frame from the ring, the sending station examines this ACK bit to decide whether the frame was correctly received.

The third feature of the token ring network is its *sequential passing* property. All stations must take turns becoming the token holder and all frames must pass through each station in a fixed order.

A *priority reservation* mechanism implemented on the token ring network is based on the above features [IEEE84]. In order to efficiently handle frames with different priority levels on a token ring network, a priority reservation (PR) field (3-bit) is defined in the frame format. When the frame passes through a non-token holder which is in listen mode, the non-token holder reads the current PR value, stores it in the inbuff (delay element in the RIP), compares it with its own priority, and writes its own priority to the PR field if the priority is higher than the reserved one. When the frame is retured to the token holder, the PR field carries the current highest priority among all stations. The token holder then raises the priority of the free token and passes the token to the next station. The station whose priority is no less than the token priority is eligible to become the token holder.

Note that the above read-compare-modify procedure is done in one bit-time for each incoming bit of information. The RIP does not hold (delay) the whole PR field and then make a reservation. Figure 6-2 shows a procedure to demonstrate the priority reservation process, which can be done by having one bit delay in the delay buffer.

6.2. Load Balancing Design Considerations

The goal of this research is to find an efficient load balancing strategy for token ring networks. In a token ring network, the single communication channel is shared by all stations. Directly implementing bidding or drafting algorithms without taking advantage of those special features provided by the ring network is very inefficient.

Both bidding and drafting algorithms have two operating phases. During the first phase, *coupling phase*, a remote processor is selected for either immigrating a process or migrating a process. The second phase is *migrating phase* which

cessor

proces

long c

change

is esse

mechan

```
(* the priority takes three bits [0..2] *)
(* oldpr[0..2] -- old priority received from the channel *)
(* newpr[0..2] -- new priority put on the channel *)
(* pcp[0..2] -- priority of the station *)
(* inbuff -- one-bit buffer *)

procedure priority_reservation;
   begin
      flag := unsolved;
      for i := 0 to 2 do (* take three bit-time *)
         begin
            inbuff := oldpr[i];
            case flag of
               unsolved:
                  begin
                     if (pcp[i] > inbuff) then
                        begin
                           newpr[i] := pcp[i];
                           flag := keepnew;
                        end
                     else if (pcp[i] < inbuff) then
                        begin
                           newpr[i] := inbuff;
                           flag := keepold;
                        end
                     else
                        newpr[i] := inbuff;
                  end;
               keepnew : (* station pr > old pr *)
                  newpr[i] := pcp[i];
               keepold : (* old pr >= station pr *)
                  newpr[i] := inbuff;
            end;
         end;
   end;
```

Figure 6-2 The process of priority reservation on token ring networks

which performs the migration of the process. During the coupling phase, a processor has to negotiate with those related processors. These negotiation processes create extra communication traffic and introduce extra time delay. A long coupling phase also increases the probability that a station may have changed its state of load. Thus, an efficient implementation of coupling phase is essential to the system performance. Also, an efficient synchronization mechanism is needed to ensure that all stations will notice the transition from coupling phase to migrating phase.

In the drafting algorithm, the light-load station sends a draft-request message to all heavy-load stations. In the token ring network, this can be achieved by sending a broadcast message to all stations. Those light-load or normal-load stations may ignore this message. The heavy-load station must reply with a draft-age message to the drafting station. In order to do so, the heavy-load station must wait until it receives the free-token. Thus, before the next token visit, the drafting station must receive all draft-age replies. If there are K heavy-load stations among N total stations, it takes at least $hK+pN$ for the drafting station to receive all replies if the rest of the $N-K$ stations don't transmit anything, where h is the average token holding time of those stations whose outgoing queue is noe-empty, and p is the average token passing time. The sending station then sends a draft-standard to the drafted station.

The original definition of the bidding algorithm requires all stations in the network to reply with a bid. In the token ring network, the bidding station has to wait at least $(p+h)N$ to complete the coupling phase.

The environment of the token ring networks strongly affect the implementation of load balancing algorithms. By taking advantage of these unique features mentioned in the previous section, the coupling phase can be efficiently implemented by defining a *writable message.*

The broadcast feature allows every processor to collect information carried in the message sent by any other processor. The bit changeable feature gives every processor a chance to compare each passing bit in the message and change its content if necessary. And the sequential passing fashion forces the serialization of actions taken by processors. These three features allow the proposal of a *writable message format* which makes it possible for the message contents to be changed while the message passes around the ring. This vehicle carrying the negotiation information enters a processor, carries the response of the processor, and goes to the next processor. The negotiation will be done by visiting every processor in the ring in one run. The negotiation result is either known by the related processor immediately or known by the token holder when the message finally returns. The writable message is a convenient data structure for the negotiation and allows the communication to obtain more hardware support from the RIP of a host processor. Thus, the RIP is more intelligent and shares the burden with the host computer to make the communication faster.

The writable message format adds one field, called the <specifier> into the head of the original message format designed for the ring network [Dixo82]. It is described by the BNF as follows.

```
<specifier>          ::= 0 | 1 <task_spec> <writable_field>
<task_spec>          ::= 00 | 01 | 10 | 11
<writable_field>     ::= <w_tag> <w_sid>
<w_tag>              ::= 0 | 1
<w_sid>              ::= integer
```

Figure 6-3 The <specifier> field of the writable message format

where, the <0> or <1> is called <ident_bit> which indicates whether the <specifier> field exists or not. The <task_spec> tells what kind of task will be accomplished. The <writable_field> is dedicated to the message the processors want to modify. Its format depends on the algorithm employed. If the <specifier> field exists, then <ident-bit> is set as 1. Otherwise, it is set as 0. In this way, the control message is piggybacked on a data message to allow the control message go through as quickly as possible. The structure of the field varies with the algorithm employed for load balancing.

6.3. The Concept of Dynamic Pairing

Because of the writable message, every processor is able to obtain the current global information about the ring and finish the negotiation within a message cycle time. Specifically, the control message communication of the coupling phase in the load balancing design becomes "real time".

As mentioned earlier, the algorithms of load balancing are classified as heavy-initial and light-initial [WaMo85]. The bidding algorithm belongs to the former, the drafting algorithm belongs to the latter. More important, the drafting algorithm introduces the internal states and the fairness concepts to make the algorithm tend toward the performance of an M/M/N multiple-server queueing system with a global first-come-first-served scheduling discipline. However, the sequential passing fashion in the ring sacrifices the parallelism existing in the environment of networks. Each time a processor becomes the token holder, it should not miss a chance to trigger a migration if conditions are appropriate. Therefore, a mixed algorithm called *dynamic pairing* is proposed to take advantage of both bidding-type and drafting-type concepts.

The benefits of adopting the bidding concept are two fold. First, the heavy-load token holder can also trigger a migration by sending a bid-request message to other processors in the ring. Second, a light-load processor is put to sleep when it figures out there were no heavy-load processors in the ring. Whenever a heavy-load processor appears, it can employ the bidding algorithm to awaken the sleeping light-load processors. Consequently, each token holder may trigger a migration in both light-load and heavy-load states. This idea can be realized due to the design of the writable message containing all the information needed by the negotiation. The message cycle time automatically synchronizes the different phases of the algorithm in the global sense but it is still difficult to synchronize global actions with local actions.

The concept of dynamic pairing in the coupling phase is to have a heavy-load processor pairing with a light-load processor and a light-load processor pairing with a heavy-load processor. The pairing activity can be initiated by either a heavy-load processor or a light-load processor. After a pair of processors have formed, process migration is then initiated between these two paired processors. There are several ways to form a pair. A good pairing algorithm has to consider the tradeoff between communication complexity, efficiency, and implementation issues. In the following sections, a first-match algorithm and a min/max algorithm are introduced. Detailed specifications and design alternatives of these approaches are discussed. A comparison study as well as the simulation results of these algorithms are covered in the next chapter.

6.4. The First Match Algorithm

The *first match* algorithm provides the simplest way to form a pair. While the draft-request message sent by a light-load processor goes around the ring, the first heavy-load processor downstream in the ring responds to the draft-request message and forms a pair. Other heavy-load processors are then prohibited to form another pair with that light-load processor. Similarily, the first light-load processor will respond to a bid-request message.

6.4.1. Definition of writable message format

The commands of the first match algorithm will be carried by a writable message format shown in Figure 6-3. All the commands for the first match algorithm are coded in the <task_spec> field as summarized below.

Implementation of the first match algorithm needs the following five parameters: current load state of the processor; number of migratable processes; L-couple buffer to record the paired light-load processor ID; H-couple buffer to

<task_spec>	command	sender	receiver
00	draft request	light-load processor	heavy-load processor
01	cancel	token holder	paired processor
10	bid request	heavy-load processor	light-load processor
11	migrate	heavy-load couple	light-load couple

record the paired heavy-load processor ID; and the task specification field of the messages passing. By designing one slot for the H-couple buffer, it is guaranteed that the light-load processor will accept at most one migrant process. Thus it prevents the light-load processor from being overloaded. Two phases of the dynamic pairing algorithm employing the first match strategy are described as follows.

6.4.2. The coupling phase

The coupling phase starts when the processor receives a token. An *ECA (Event-Condition-Action) tabulation* method is proposed to formally specify the algorithm. Whenever an event occurs, the processor has to check some conditions. A set of actions is performed only when all conditions are satisfied. The ECA table provides a clear, concise, and elegant way to specify a protocol, compared with state transition diagrams or standard language specifications. Each column in the table corresponds to one transition due to the occurrence of an event and the satisfaction of the corresponding conditions. Each row belongs to one of four sets. The first set (E) represents the event. The second set (C) lists a set of conditions. The corresponding actions are fired if all the conditions are satisfied. A null entry in a condition implies a don't care condition. The third set (A) indicates the actions to be performed. The last set is a transition indexing for reference purpose. Note that if the table is very wide, it may be partitioned at the boundary of different conditions. The symbol % in the leftmost column represents a place-holder; its content is listed in the corresponding row entry.

Processor to processor communication is via ***send(destination,message)*** command. If the destination is *all*, then it is a broadcast message. An ***L-couple*** or ***H-couple*** in the destination field indicates its paired light-load processor or paired heavy-load processor, respectively. The message field is indicated by one of the four commands specified in the <task_spec> field defined earlier.

Table 6-1. The ECA table of a light-load processor receiving the token

E	event	receive a free token				
C	load state	light	light	light	light	light
	H-couple=empty?	yes	yes	yes	no	no
	L-couple=empty?	yes	yes	no	yes	no
	Lsleep=TRUE?	no	yes			
A	auto-check				yes	yes
	send(all, %)	draft				
	send(H-couple,%)			cancel*		cancel
T	Transition	1	2	3	4	5

*Cancel has higher priority than request because the consistency is more important than migration

Table 6-1 lists all actions taking place when a light-load processor becomes the token holder. The state of light-load and the empty of its H-couple buffer and L-couple indicates it can accept a migrant process (transition 1). However, if all processors are in light-load state, no process migration should be triggered as indicated by the Lsleep flag (transition 2). The request is either piggybacked on the first message already in the outgoing queue (if the first packet has not carried any control message in it), or sent in a newly created message. In this section, we only discuss normal cases. Those abnormal cases (transitions 3-5) will be detailed in the next section.

While the draft request passes a non-token holder processor in the ring, the actions listed in Table 6-2 take place according to the processor's current status. The non-token holder processor learns that the message is a draft request by reading the task specification code. The processor fills its ID into the writable field if the processor is in the heavy load state, it has not yet paired with another L-couple, and no other processor has made pairing reservation with the requesting processor (transition 8). Note that the latter case can be detected if the <w_tag> field was on. It then turns on the prohibition tag bit in <w_tag> to prevent refilling from other heavy-load processors and puts its address in the <w_sid> field. Meanwhile, the heavy-load processor records the ID of the light-load drafting processor (drafter address) into its L-couple buffer.

Eventually, the draft request message comes back to the token holder initiating the draft request. The token holder records the processor ID carried by the message field <w_sid> into its H-couple buffer (transition 12 of Table 6-3). If there is not any heavy-load processor in the ring, the token holder goes to sleep by setting its Lsleep flag to true (transition 11 of Table 6-3). The coupling

Table 6-2. The ECA table of a non-token holder receiving a draft request

E	event	receive a draft request				
C	token holder?	no				
	load state	light	normal	heavy		
	L-couple=empty?			yes		no
	reserved?			yes	no	
A	Hsleep $\leftarrow$ %			FALSE		
	make reservation			yes		
	L-couple $\leftarrow$ %			drafter addr		
T	transition	6	7	8	9	10

phase is done.

Table 6-3. The ECA table of the token holder receiving a draft request

E	event	receive a draft request			
C	token holder?	yes			
	load state	light		normal	heavy
	reserved?	no	yes		
A	Lsleep $\leftarrow$ %	TRUE			
	H-couple $\leftarrow$ %		<w_sid>		
T	transition	11	12	13	14

Similarily, once the token holder is in the heavy load state, it then sends a bid-request message trying to pair a light-load processor as shown in Table 6-4 (transition 15). If the token holder's Hsleep flag is true indicating all processors are in heavy-load state, then no process migration action will be triggered (transition 16). If the heavy-load token holder has a paired L-couple, it will migrate a process to the paired processor (transition 17). If it had paired a heavy-load processor, it will cancel the pairing relationship with that heavy-load processor (transition 18).

Table 6-4. The ECA table of a heavy-load processor receiving the token

E	event	receive the free token			
C	load state	heavy			
	H-couple=empty?	yes			no
	L-couple=empty?	yes		no	
	Hsleep=TRUE?	yes	no		
A	send(all,%)		bid request		
	send(L-couple,%)			migrate	
	send(H-couple,%)				cancel
T	transition	15	16	17	18

When the bid-request message passes through the first light-load non-token holder downstream, the light-load processor will turn its Lsleep flag off if the flag has been true. It then fills its processor ID into the writable field, marks the prohibition tag bit as 1 and records the bid sender's ID into its H-couple buffer as described in Table 6-5.

Table 6-6 illustrates a case where the bid command returns to the token holder sending the bid request message. The token holder then records the processor ID carried by the message field <w_sid> into its L-couple buffer (tr

pro

tio

Table 6-5. The ECA table of a non-token holder receiving a bid request

E	event	receive a bid request				
C	token holder?	no				
	load state	light			normal	heavy
	H-couple=empty?	yes		no		
	reserved?	no	yes			
A	Lsleep ← %	FALSE				
	make reservation	yes				
	H-couple ← %	bidder addr				
T	transition	19	20	21	22	23

(transition 27). If no processor made pairing reservation, it indicates that all processors are in heavy-load state. The Hsleep flag is then set to true (transition 26).

Table 6-6. The ECA table of the token holder receiving a bid request

E	event	receive a bid request			
C	token holder?	yes			
	load state	light	normal	heavy	
	reserved?			no	yes
A	Hsleep ← %			TRUE	
	L-couple ← %				<w_sid>
T	transition	24	25	26	27

ma

th

mi

ho

ces

anc

abl

the

the

con

<w

pro

cha

to

pos

the

in

add

the

6.4.3. The migrating phase

The migrating phase starts whenever the heavy-load processor which has made a migration reservation becomes the token holder. If the load states of the coupled processors remain original, as in the coupling phase when the migrating phase starts, the migration continues normally. The heavy-load token holder will send a migrate command message associated with the migrant process to the paired destination processor as shown in Table 6-4.

There are two kinds of migrate commands. One is for migrate only, another is for both migrate and bid, depending on the number of the migratable processes currently existing in the heavy-load processor. At the time when the heavy-load processor migrates the migrant process, if it finds the number of the migratable processes is greater than one, it will send the migrate and bid command. This can be achieved by issuing a migrate command and having <w_tag>=0 (enable bidding procedure) or <w_tag>=1 (disable bidding procedure). Upon receiving the migrant process, the destination processor changes the task specification field of the message from the migrate command to the bid command (transitions 28 and 29). The light-load processors whose positions are behind the destination processor in the ring give its response to the bid command later on. Thus, one message plays a double role as tabulated in Table 6-7. Note that the condition ***address match*** means that the sender's address must be same as the H-couple.

The process migration is a global action which performs in parallel with the local actions taking place in different local processors. When the global

Ta

ac

fi

m

r

A

t

1

Table 6-7. The ECA table of a non-token holder receiving a migrate command

E	event	receive a migrate command				
C	token holder?	no				
	load state	light			normal	heavy
	address match?	yes		no		
	<w_tag>=0 (bid)?	yes	no			
A	receive migrant process	yes	yes			
	change to bid command	yes	yes			
T	transition	28	29	30	31	32

action involves a processor, it is difficult to tell whether a current local state is final or not. For example, at the moment a processor receives a draft request message, it may be in the middle of a migration phase. It responds to the draft request that it has one migratable process before migrating this process out. After a while, it changes its load state from heavy-load to normal-load, due to the fact that it migrated the migratable process. Thus, it leaves an inconsistent state implicit. However, it is impossible to synchronize the global action with the local action. Therefore, it cannot avoid anomalies, especially when the time period between the coupling phase and the migrating phase lasts longer. The anomalies include :

(1) The L-couple processor which initiated the coupling phase changes its state to normal or heavy before the draft request message returns.

The original light-load token holder can ignore the result of the request message immediately; i.e., it breaks the pairing relationship with its H-couple.

When

and

beca

brok

sor,

bac

the

res

mig

cou

rese

cou

be

st

m

no

When the H-couple becomes the token holder, it still starts the migrating phase and sends a migrant process to the destination, namely its L-couple processor, because the H-couple is not informed that the pairing relationship had been broken. At this moment, regardless of the current state of the L-couple processor, it cannot accept the migrant process. Thus the migrant process will come back to the heavy-load processor. The migrant process should be sent back to the schedule queue, and the L-couple buffer of the heavy-load processor will be reset.

(2) The L-couple processor changes its state to normal or heavy before the migrant process reaches it.

The H-couple processor migrates the migrant process, but the original L-couple processor cannot accept the migrant process. The L-couple processor reset its H-couple buffer. The migrant process will be returned back to the H-couple processor and sent back to the schedule queue.

(3) The original H-couple processor changed its state to normal or light before the moment of migrating.

When the H-couple processor becomes the token holder, and if its own state has been changed to normal or light load state, it sends a cancel command to cancel the result of the coupling phase.

(4) The heavy-load processor which initiated the bid command becomes normal or light state before the bid command returns back.

The original H-couple processor can ignore the result of the bid command immediately, but the L-couple processor had set up the pairing relationship with the H-couple processor. Suppose that the L-couple processor will stay in the light state forever, then it will wait for the impossible migration forever, i.e., it will never participate the drafting algorithm again. Therefore, we need a special treatment, called *auto check*, for this case. When a processor gets the token and detects its H-couple buffer, if the buffer has the same processor ID in it for three times, it will reset the buffer to 0 automatically to cancel the coupling result. This case is reflected in Table 6-1 which indicates the processor has light load state but its H-couple buffer is not empty.

Table 6-8. The ECA table of a normal-load processor receiving the token

E	event	receive the free token			
C	load state	normal			
	H-couple=empty?	yes		no	
	L-couple=empty?	yes	no	yes	no
A	send(L-couple,%)		cancel		
	send(H-couple,%)			cancel	cancel
T	transition	33	34	35	36

The typical cancel situations happen in the cases listed in Table 6-8. Other cases are spread in Tables 6-1 and 6-4. Note that in transition 36, both L-couple and H-couple should be cancelled. However, cancelling H-couple has higher priority than cancelling L-couple because L-couple means nothing to the normal state but the H-couple may still migrate a process to a normal processor.

Note the <w_tag> is used to indicate whether the L-couple or the H-couple is to be cancelled. When the cancel command passes a non-token holder, the processor checks its processor ID; if it matches the ID carried by the message, it cleans its H-couple or L-couple accordingly. When the cancel command returns to the token holder, no more actions are required. State transition tables of the above events are rather simple and are not showned here.

6.4.4. Advantages and disadvantages of the first match algorithm

Obviously, the first match algorithm does not require the load table and it is a simple control strategy. It is, however, unfair in the sense that the first H-couple might not be the current heaviest processor or the first L-couple might not be the current lightest processor in the ring network. Imagine that there are many processors, say 100 in the ring; then only a few processors around a heavy-load processor may couple with the heavy-load processor, and the processors whose locations are "far away" from the heavy-load processor may never get a chance to have a migrant process. This phenomenon is called ***migration starvation*** and will be explained in detail in the next chapter. A more serious problem is that the migrating phase cannot start until the H-couple processor becomes the token holder. The longer the time period between the coupling phase and the migrating phase, called ***state change period***, the higher the probability that the anomalies will occur. If the H-couple processor is located far away from its L-couple processor, then its state change period will be long. Other algorithms must be studied to overcome the disadvantages mentioned above.

6.4.5. Design alternatives

The different token passing disciplines give different alternatives of the algorithm. The major alternatives are

- Single coupling, single migrating -- Each token holder can only make one couple and each heavy-load token holder can only migrate one migratable process if it has made a migration reservation. This alternative makes a fair equal opportunity for every processor to send a message and migrate process.
- Multiple coupling, multiple migrating -- Each heavy-load processor can make multiple migration reservations of n and each H-couple can migrate multiple migratable processes of m. This version makes the load balancing mechanism more complicated than others since it requires list data structure for n migration reservations, more hardware and more complicated exception handling considerations. Meanwhile, choosing a proper value of n and m is a formidable task since it depends on the various parameters of the network.
- Single coupling, multiple migrating -- Each processor makes only one couple but whenever the H-couple becomes the token holder, it is allowed to migrate as many migratable processes as possible. The control is simple. The number of migrations in one token cycle time period is no longer restricted. The token passing time will be longer than that in the first version. This alternative is inferior to the individual heavy-load processor. Especially in the situation where a heavy-load processor stays at the

heavy-load state forever, load balancing is effectively killed.

6.5. The Min/Max Algorithm

The second algorithm is called min/max algorithm. Every migration occurs between the current maximum heavy-load processor and the current minimum light-load processor. Obviously, the algorithm provides better fairness but more complexity.

6.5.1. Definition of the writable message format

The min/max algorithm attempts to couple the min-max pair. The writable message format must have fields to carry the values of the light load and values of the heavy load as a basis of comparisons. The <h_sid> and <l_sid> fields are used to keeping track of the minimum light-load processor ID and the maximum heavy-load processor ID. A <w_tag> is set for distinguishing the ordinary command and the confirm message.

The specification of the <specifier> of the writable message format by BNF expressions is as follows.

```
<specifier>         ::= 0 | 1 <task_spec> <writable_field>
<task_spec>         ::= 00 | 01 | 10 | 11
<writable_field>    ::= <w_tag> <h_load> <h_sid> <l_load> <l_sid>
<w_tag>             ::= 0 | 1
<h_load>            ::= integer
<h_sid>             ::= integer
<l_load>            ::= integer
<l_sid>             ::= integer
```

Figure 6-4. The <specifier> field of the writable message format of the min/max algorithm

All the <task_spec> codes are listed below.

00 -- couple, sent by the current light- or heavy-load token holder
01 -- cancel, sent by the token holder
10 -- confirm, sent by the token holder to confirm the coupling result
11 -- migrate, sent by the heavy couple, received by the light couple

6.5.2. The coupling phase

The draft and the bid commands lose their differences because both commands are looking for the min/max pair. When a processor becomes the token holder, it checks its load state. If the load state is either light or heavy, it sends a couple message into the ring. Every non-token holder processor compares its load with the load value carried by the message. If a non- token holder is lightly loaded and its load is less than the light load value carried by the message, it will fill its load value and its processor ID into the message. Similarly, if a non token holder is heavily loaded and its heavy load is greater than the heavy load value carried by the message, it will fill its heavy load and its processor ID into the message. Thus, the token holder will receive a packet which ran around the ring and contained the min/max information which, however, has not been confirmed since nobody knows which processor eventually is the min or the max except the token holder.

6.5.3. The confirming phase

Therefore, an additional confirming message should be sent out before the token holder releases the token to the next processor. The confirming message carries the min/max information and set the <w_tag> as 1 to indicate that the writable field is not changeable. Each light-load processor compares its processor ID with the <l_sid> and each heavy-load processor compares its processor ID with the <h_sid> in the confirming message. A match indicates that the processor is the minimum light processor or the maximum heavy processor, respectively. All of the no match cases rely on the cancel mechanism.

6.5.4. The migrating phase

The migrating phase does not start until the maximum heavy-load processor becomes the token holder. The normal migrating phase is the same as that of the first match algorithm. The anomaly situations are different.

The state change time period between the coupling phase and the migrating phase is longer than that of the first match algorithm because it requires an additional confirming phase. Thus the probability that the state will be changed is larger than that of the first match algorithm. On the other hand, the channel transmission rate is very high in the local area networks. Even though two runs are needed before the migrating phase, the state change period is still small compared with the processes interarrival time. And the min/max algorithm pairs the maximum heavy-load processor with the minimum light-load processor with more stable status than the couples of the first match algorithm. This fact alleviates the probability of the state change so that the problem is not very serious.

The anomalies are :

(1) The token holder is the original minimum light-load processor or the

original maximum heavy-load processor, but it changed its state before the coupling message returns.

The coupling phase is failed and ignored. There will be no confirming message. The token holder can release the token immediately.

(2) The minimum light-load processor changed its state to normal or heavy or the maximum heavy-load processor changed its state to normal or light after the coupling message returns but before the confirming message reaches it.

The coupling result is ignored. However, either the H-couple has been confirmed or the L-couple has been confirmed. If it is the former, then the H-couple processor will start the migrating phase and send a migrant process to the destination, namely, the L-couple processor. At this moment, regardless of the current state of the L-couple processor, it cannot accept the migrant process. Thus the migrant process will be returned to the H-couple processor, sent back to the schedule queue, and the L-couple buffer will be reset. If it is the latter, the L-couple processor recorded the original H-couple processor ID into its H-couple buffer. Suppose that the L-couple processor will stay in the light state forever. Then it will never participate in the algorithm again. Therefore, the auto check procedure should be applied.

(3) The original L-couple processor changed its state to normal or heavy before the migrant process reaches it.

The H-couple processor migrates a migrant process, but the original L-couple processor cannot accept the migrant process. The migrant process will be returned to the H-couple processor and sent back to the schedule queue.

i

p

e

p

pr

W

to

ma

<l

(4) The original H-couple processor changed its state to normal or light before the migranting moment.

When the H-couple processor becomes the token holder, and if its load state has been changed to normal or light, then it sends a cancel command.

6.5.5. Advantages and disadvantages of the min/max algorithm

The min/max algorithm which always pairs the minimum light-load processor with the maximum heavy-load processor gives the load balancing fairness the avoidance of the "migration starvation" phenomenon as mentioned in the first match algorithm. On the other hand, it requires every processor to take care of the load comparison, which indeed increases the burden of every interface. The confirming phase requires an extra message causing a longer state change time period and more channel traffic. Therefore, we propose an improved min/max algorithm which attempts to eliminate the confirming phase.

6.5.6. Design alternatives

A design alternative to the min/max algorithm only allows the heavy-load processor to send the couple command to couple with the minimum light-load processor but the light-load token holder does not send the couple command. When the message which carried the coupling result back to the heavy-load token holder arrives, the processor checks the result to see if it is the current maximum heavy-load processor in the ring by comparing the value of the <h_load> carried in the message with its own heavy load value. If so, it starts the migrating phase immediately. Otherwise, the result is ignored.

The algorithm, called the *modified min/max algorithm* provides additional features beyond the advantages of the original min/max algorithm. The features are that there is only one cycle before the migrating phase and there is no state change time period. The major problem is that the coupling may never succeed since the token holder may not be the current maximum heavy-load processor and the current maximum heavy-load processor may not be the token holder simultaneously.

A *coupling standard* will be applied to alleviate this problem. The coupling standard is a range of the difference between the current value of the maximum load in the ring and the current load value of the token holder. Suppose the coupling standard is predefined as 2 and the current maximum load value carried by the message is 4 but the load value of the token holder is 3. The difference is 1 which is within the coupling standard. Then the token holder can start the migrating phase even though it is not exactly the current maximum heavy-load processor in the ring. The coupling standard is an important design parameter of the algorithm.

In a situation where more than one processor has the same minimum light-load value, say 0, or more than one processor has the same maximum heavy-load values, it will force the modified min/max algorithm back to the first match algorithm since the first light-load processor in the path will be selected as the L-couple. One way to distinguish them is to use another design alternative of the min/max algorithm, called the statistical min/max algorithm, which requires a load table to record the statistics. The load of a processor may either be measured by the number of the processes in the queue or measured by the time period, named *state counter*, in which the processor remains in one of the three state groups. By using the second measurement, the statistical method attempts to record the state counter in a load table. The bigger the state counter, the longer the processor remains in the state group. When the heavy-load token holder checks the load table and finds it is the current maximum heavy-load processor. It couples the minimum light-load processor from the load table, and starts the migration.

Every processor has its own entry in the load table. An entry includes an L-counter for recording the light-state counter and a H-counter for recording the heavy-state counter. The rules for the counting are defined as

- draft request command increases the token holder's L-counter by one;
- bid command increases the token holder's H-counter by one;
- migrate command reduces the token holder's H-counter by one and the receiving non-token holder's L-counter by one;
- normal state broadcast command resets the token holder's H-counter and L-counter.

Note that the coupling phase is replaced by the message recording. The load table requires extra memory. Thus the message recording may cause a buffer overflow problem. The number of messages is increased because the token holder must send a message per token visit, including broadcasting the normal state information.

d

th

di

ri

th

tu

7.

sin

des

Pa

are

(1)

(2)

(3)

(4)

CHAPTER 7

PERFORMANCE EVALUATION OF DYNAMIC PAIRING STRATEGIES

The performance of the dynamic pairing algorithms are studied by conducting simulation experiments on the token ring network environment. First, the performance of the first match strategy and the min/max strategy is studied. Then, the performance of the drafting algorithm and the bidding algorithm under the same environment are compared. The simulation results reveal the quantitative differences between these algorithms, and provide a clearer picture of the conceptual analysis described previously.

7.1. Performance of Dynamic Pairing Algorithms

The first match and the min/max algorithms have been simulated by a simulator written in the C language. The main points of the simulator were described in Chapter 4.

Parameters for the simulation

For the purpose of performance measurement, the following assumptions are made:

(1) There are N processors in the ring network, where N=10 in our simulation;

(2) The physical distance between two adjacent RIPs is D meters, where D=50 in our simulation;

(3) The transmission rate of the channel is R mbps, where R=10 in our simulation;

(4) The length of the token is T bytes (8T bits), where T=10 in our simulation;

(5) The length of a control message is C bytes (8C bits), where C=10 in our simulation;

(6) The mean length of a migrant process is K bytes (8K bits), where K=1000 in our simulation and the actual length of a migrant process is a random number obtained by calling the exponential random generator aexp(8K);

(7) Every RIP makes 1 bit delay.

A message sent by the token holder and then returning to the token holder takes a period of time called message delay = processor number * (interprocessor propagation delay + 1 bit delay) + message length / channel transmission rate, where, interprocessor propagation delay = D / (2/3 * light speed); 1 bit delay = 1 / channel transmission rate. The message delay time is normalized by using the same unit as the one for normalizing the local processor's service rate. It is assumed the unit is norm_unit = 0.001 second. Therefore,

$$msg\_delay = (N * (3 * D / 2 * light\_speed + 1/R) + msg\_length / R) / norm\_unit$$

where, for control message the msg_length = 80; for migrant process the msg_length = aexp(8K).

The results and analysis

The simulation results based on the comparisons of two algorithms are analyzed.

The performance of the first match and the min/max algorithms under the condition of Q=3 is shown in Figure 7-1 illustrating the curve of the performance divided into 2 parts. In the range of the total arrival rate of 5.0 to 7.5, both the first match and the min/max algorithms have nearly the same mean response time, even though the min/max is little better than the first match. In the lower load situation, the processors are likely lightly loaded. Under the min/max algorithm, a heavy-load processor also chooses the first light-load

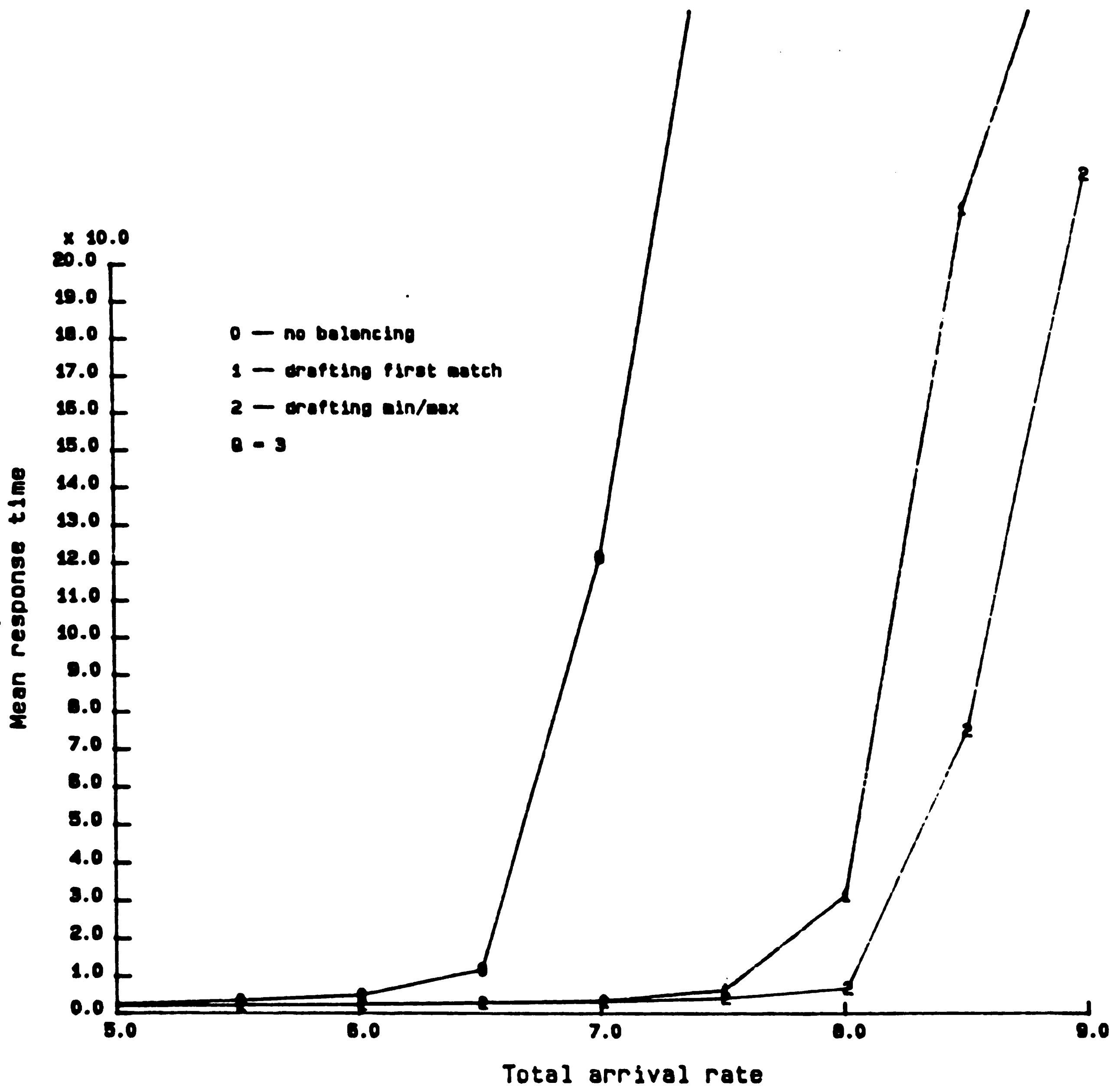


Figure 7-1 Performance of first match and min/max algorithm

processor downstream as the minimum light-load processor since several light-load processors downstream may have the same minimum load value, for example, 0. Therefore, the min/max algorithm behaves like the first match algorithm. When the total arrival rate reaches 7.5 and up, the min/max algorithm exhibits better performance than the first match algorithm.

More detailed data from the simulation results of two algorithms are shown in Tables 7-1 to 7-4. Tables 7-1 and 7-2 refer to the total arrival rate 5.01 corresponding to the lower load situation, while Tables 7-3 and 7-4 refer to the total arrival rate 8.51 corresponding to the higher load situation. The distribution of the arrival rate along the 10 processors is $\lambda_i = 0.147 * \text{total arrival rate}$, for i = 1,2,3; and $\lambda_j = 0.08 * \text{total arrival rate}$, for j = 4,5,6,7,8,9,10. Therefore, processors 1 to 3 form a ***heavy oriented group*** and processors 4 to 10 form a ***light oriented group.*** The results of the first match algorithm possess a "biased property". However, the results of the min/max algorithm show an "even property". This is clearer in the high total arrival rate situation than in the lower total arrival rate situation. From the data listed in the Tables, we can see

(1) The number of migrant processes migrated out from the heavy oriented processor group (the "mig out" row of the tables) is in decreasing order in the first match, but not in the min/max under the higher total arrival rate situation;

(2) The number of the migrant processes entering the light oriented processor group (the "mig in" row of the tables) is in the decreasing order in the first match, but not in the min/max under the higher total arrival rate situation;

(3) In the dynamic process migration, it is possible for a light oriented processor to migrate out some processes. It is also possible for a heavy oriented processor to accept some migrant processes. This is called ***bidirectional***

migration. The bidirectional migration phenomenon of the first match algorithm is more serious than that of the min/max algorithm (the first and the second rows of the tables). The bidirectional migrations contain some unnecessary migrations which degrade the system performance.

(4) Thus, the mean response time of the min/max algorithm are more even than those of the first match algorithm (the "resp. time" row of the tables).

(5) The standard deviation of response times of the min/max algorithm are less than those of the first match algorithm (the "stand. devia." row of the tables). This reflects that the min/max algorithm is more fair than the first match algorithm.

(6) The number of processes left when simulation stopped in the min/max algorithm is less than that in the first match algorithm (the "late msg" row of the tables).

(7) Unfortunately, the number of the too-late messages of the min/max algorithm are greater than those of the first match algorithm because the phase transition period of the min/max algorithm is longer than that of the first match algorithm.

(8) From Tables 7-3 and 7-4 we can see that the mean response time of the light oriented processors are still small, i.e. the light oriented processors still have a potentially large capacity to improve the performance under the higher total arrival rate situation. The queue emptying discipline adopted restricts the number of migrations. Other kinds of queue emptying discipline to reduce this restriction may be employed.

Table 7-1. First match algorithm, Q=3, R=5.01
(r1=r2=r3=0.73, r4-r10=0.40)

id	1	2	3	4	5	6	7	8	9	10
mig out	828	790	716	85	80	63	60	56	70	62
mig in	72	375	512	874	353	188	124	99	86	96
ctrl msg	3497	3834	3920	2628	1762	1481	1446	1342	1347	1340
late msg	35	44	43	11	11	4	0	7	12	4
draft msg	2032	2225	2326	1962	1419	1264	1258	1174	1181	1175
bid msg	557	551	524	68	54	48	54	44	46	46
resp. time	2.39	2.39	2.38	1.72	1.64	1.61	1.56	1.56	1.59	1.64
stand. devia.	2.06	1.98	2.00	1.66	1.65	1.56	1.48	1.51	1.54	1.58
proc. left	5	2	1	0	1	0	0	1	1	0

Table 7-2. Min/max algorithm, Q=3, R=5.01
(r1=r2=r3=0.73, r4-r10=0.40)

id	1	2	3	4	5	6	7	8	9	10
mig out	802	721	717	60	65	69	71	68	71	59
mig in	47	200	320	649	474	303	230	162	133	126
ctrl msg	2208	2323	2430	2288	2061	1995	2028	1998	1969	1910
late msg	34	34	42	30	57	33	36	45	20	19
draft msg	1052	1191	1205	1927	1802	1795	1820	1821	1796	1767
bid msg	498	452	473	51	52	53	55	47	53	47
resp. time	2.33	2.30	2.35	1.65	1.66	1.63	1.57	1.56	1.63	1.64
stand. devia.	1.95	2.05	2.01	1.56	1.56	1.55	1.46	1.50	1.56	1.56
proc. left	0	1	1	0	0	0	1	3	0	1

Table 7-3. First match algorithm, Q=3, R=8.51
(r1=r2=r3=1.25, r4-r10=0.68)

id	1	2	3	4	5	6	7	8	9	10
mig out	1693	1434	1233	169	205	191	255	195	218	158
mig in	0	0	0	862	860	848	674	685	574	549
ctrl msg	308	457	618	881	929	892	850	856	809	857
late msg	0	0	0	115	140	145	165	140	130	126
draft msg	0	0	0	649	719	734	671	711	677	727
bid msg	308	457	618	205	180	133	135	98	89	83
resp. time	346.0	472.5	735.2	3.2	3.1	2.9	3.2	3.0	3.1	2.7
stand. devia.	136.8	153.3	230.1	2.7	2.8	2.5	2.8	2.5	2.6	2.3
proc. left	784	899	1483	1	0	4	4	1	3	4

Table 7-4. Min/max algorithm, Q=3, R=8.51
(r1=r2=r3=1.25, r4-r10=0.68)

id	1	2	3	4	5	6	7	8	9	10
mig out	1968	1699	2171	27	24	39	31	35	43	46
mig in	0	0	0	666	671	738	730	765	685	634
ctrl msg	1184	1372	1171	1962	1751	1720	1707	1594	1603	1449
late msg	0	0	0	220	230	276	243	251	188	200
draft msg	0	0	0	602	572	558	582	558	542	486
bid msg	720	795	465	474	379	358	388	337	451	441
resp. time	212.2	215.7	220.3	3.5	3.2	3.3	3.3	3.0	3.5	3.3
stand. devia.	63.4	61.8	67.1	3.1	2.8	2.9	2.9	2.6	3.1	2.8
proc. left	447	447	454	1	1	0	0	1	0	3

7.2. Performance comparison of bidding and drafting algorithms

The drawbacks of the bidding algorithm have been analyzed and overcome in the design of the drafting algorithm. However, a comprehensive quantitative comparison between the drafting algorithm and the bidding algorithm has not been given. Based on the same token ring network environment, we conducted a series of simulation experiments for both the drafting algorithm and the bidding algorithm. Here, the drafting algorithm is one which differs from the dynamic pairing algorithm investigated in Chapter 6 in the sense that it does not employ the heavy-initiated method. Consequently, the performance of the drafting algorithm is a little worse than that of the dynamic pairing algorithm as shown in Figure 7-3.

Simulation models

The simulation model of the drafting algorithm is the same as shown in Figure 4-2. The bidding algorithm does not have the concept of the internal load states. The model is simple as shown in Figure 7-2. The system parameters are the same as described in Section 7.1.

In the model, new arrivals enter the waiting queue of a processor. Migrant processes enter the incoming queue. All processes which will be migrated and other messages are accumulated in the outgoing queue. The migratable processes are the new arrivals in the waiting queue. On the token ring network, only the token holder has the authority to use the channel. The token holder may not have a new arrival, while the processor with a new arrival may not be the token holder. In order to overcome this difficulty, a rule is employed, which says that the token holder can send out a bid request if it had at least one new arrival during the previous token cycle time, that is, the period of time after it released the token until it receives the token again. The load difference between the source processor and the destination processor is selected and given the same value as the Q value in the drafting algorithm, the threshold value between the heavy load state and the light load state.

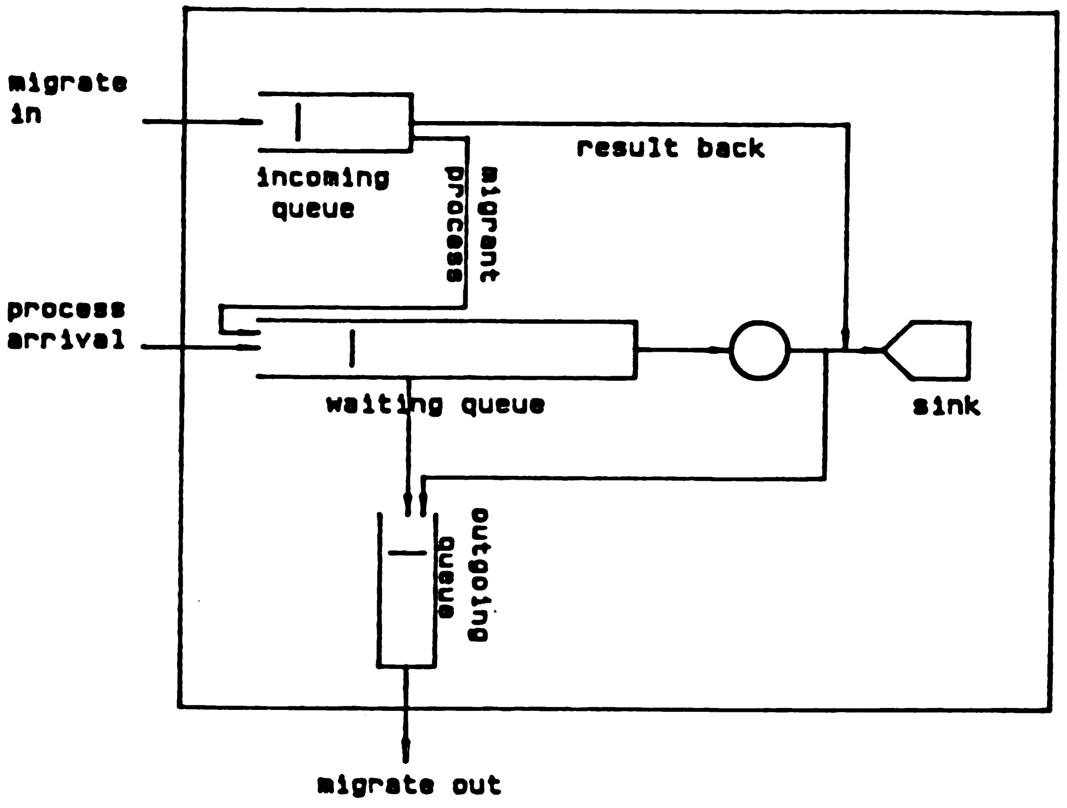


Figure 7-2 The simulation model of bidding algorithm

Simulation results

Figure 7-3 shows the performances of the bidding algorithm and the drafting algorithm. It is clear the drafting algorithm has higher performance than that of the bidding algorithm. Tables 7-5 and 7-6 list more detailed data from the two algorithms under the same parameters.

The following conclusions may be drawn from the tables:

(1) The bidding algorithm does not have the concept of internal load states. Whenever a processor has a new arrival, it sends a bid request no matter how heavy its load is. This creates a lot of control messages even on the token ring network environment (the row "cntl msg" of the tables). The drafting

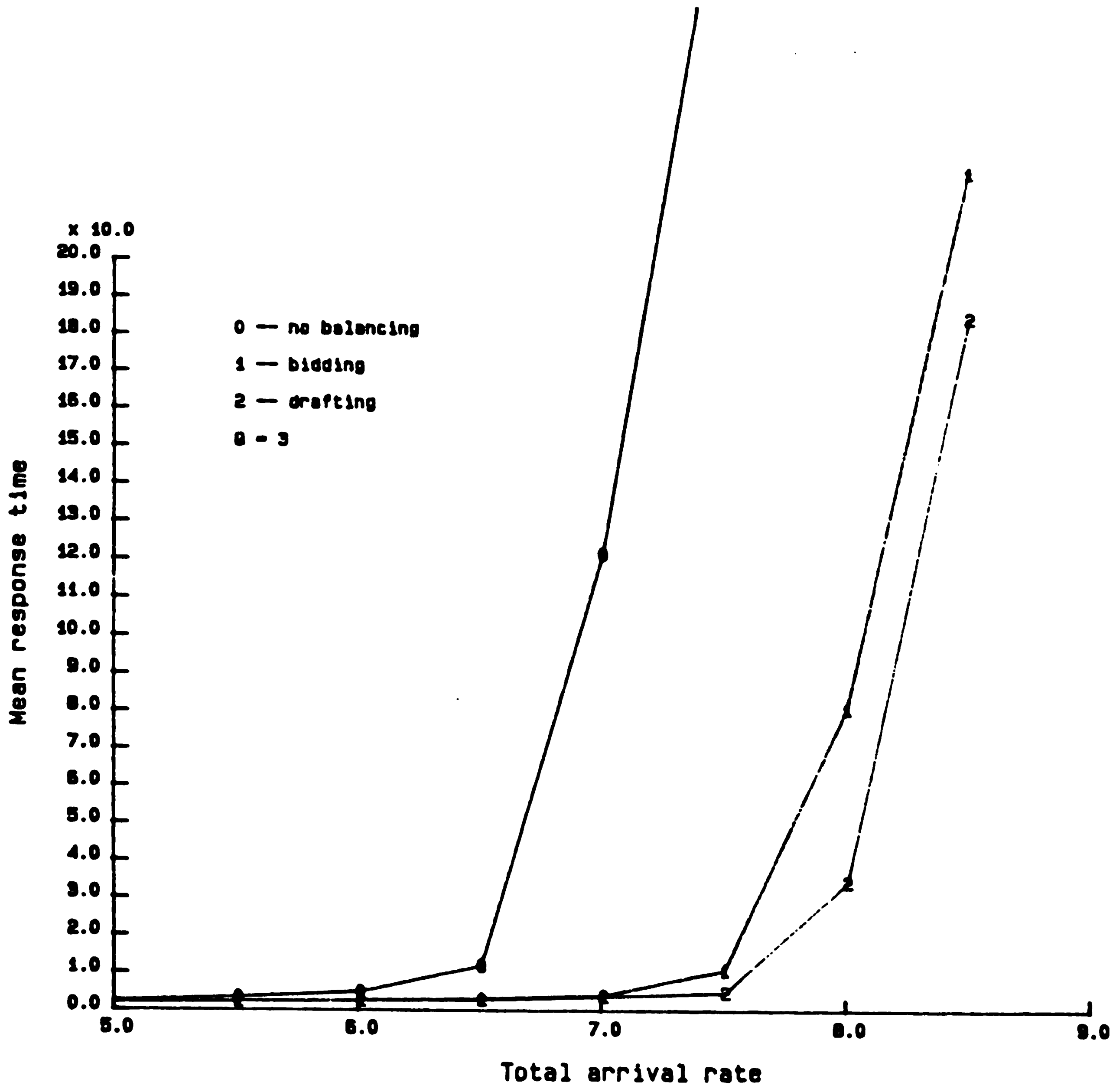


Figure 7-3 Performance of bidding and drafting algorithm

Table 7-5. Bidding, first match algorithm, Q=3, R=8.51
(r1=r2=r3=1.25, r4-r10=0.68)

id	1	2	3	4	5	6	7	8	9	10
mig. out	1308	1263	1249	148	140	133	129	112	118	120
mig. in	0	2	5	766	748	640	546	484	428	387
ctrl msg	1803	1828	1826	1594	1530	1576	1611	1550	1580	1511
late msg	0	0	0	274	247	253	232	209	176	143
draft msg	0	0	0	0	0	0	0	0	0	0
bid msg	1803	1828	1826	1489	1429	1441	1471	1422	1472	1414
resp. time	518.3	590.1	481.0	3.3	3.1	3.0	3.2	2.7	2.8	2.8
stand. devia.	262.3	325.7	241.7	3.0	2.8	2.5	2.7	2.3	2.6	2.5
proc. left	942	1040	938	1	8	2	4	8	0	4

Table 7-6. Drafting, first match algorithm, Q=3, R=8.51
(r1=r2=r3=1.25, r4-r10=0.68)

id	1	2	3	4	5	6	7	8	9	10
mig. out	2081	1465	937	51	120	191	201	191	239	217
mig. in	0	0	0	721	740	738	709	706	631	594
ctrl msg	0	0	0	893	961	944	971	1006	988	1044
late msg	0	0	0	109	145	168	200	175	163	152
draft msg	0	0	0	876	940	910	932	951	938	1002
bid msg	0	0	0	0	0	0	0	0	0	0
resp. time	208.4	420.1	713.2	3.8	3.0	3.1	3.1	2.9	3.0	2.7
stand. devia.	61.6	162.2	208.9	3.7	2.4	2.7	2.7	2.5	2.5	2.3
proc. left	289	794	1356	0	2	0	1	1	1	1

algorithm sends out 6549 drafting requests, and ends up with 5693 migrations; The bidding algorithm sends out 15595 bid requests, and ends up with 6028 migrations. The number of bid requests is about twice as much as the number of drafting requests for nearly the same number of migrations.

(2) The bidding algorithm migrates the new arrivals from the tail of a waiting queue in the original processor to the another tail of the waiting queue in the remote processor. In heavy load situation, migrating the processes from the tail of the waiting queue does not give a considerable contribution to the response time. Therefore, the average response time of the bidding algorithm is 223.51, rather than 184.50 of the drafting algorithm when the total arrival rate is 8.51. However, the response time distribution is more even in the bidding algorithm than in the drafting algorithm. The reason is that the number of light oriented processors is greater than the number of heavy oriented processors in this simulation. It is easier for heavy-load processors to find a couple in the bidding algorithm than for the light-load processors to find a couple in the drafting algorithm. This leads to the result that the processor 1 migrated more processes than the processor 3 in the drafting algorithm, so that the response time of processor 3 is much longer than that of processor 1.

was

tok

The

cha

8.1

nor

to a

pert

data

Whe

med

Secc

whe

eith

term

(not

wher

delay

delay

termi

CHAPTER 8
DISCUSSION OF IMPLEMENTATION ISSUES

The implementation of the drafting algorithm on the token ring networks was considered in Chapter 6. The other two standardized local area networks, token passing bus and CSMA/CD networks, provided different features [Stal84]. The implementation issues on these two networks are further discussed in this chapter.

8.1. Implementation Discussion on Token Passing Bus

The token passing bus physically is a bus, but logically is a ring. Under normal (error-free) conditions, the operation of this type of network is similar to a token ring network. However, the bus network has two special basic properties which are different from the ring network. First, with a bus network, all data terminal equipments are connected directly to the transmission medium. When a data terminal equipment transmits (broadcasts) a message on the medium, it is received by all active data terminal equipments in the network. Second, there is a maximum time requirement for a data terminal equipment when it waits for a response to a transmitted message before it assumes that either the transmitted message was corrupted or the specified destination data terminal equipment was inoperable. This time is known as the ***slot time tau*** (not the same as used in a CSMA/CD bus) and can be defined as :

$$\tau = 2(\textit{transmission path delay} + \textit{processing delay})$$

where the transmission path delay is the worst-case transmission propagation delay going from any transmitter to any receiver in the network and processing delay is the maximum time for the medium access control unit within a data terminal equipment to process a received frame and generate an appropriate response. On receipt of a valid token frame, a data terminal equipment may transmit a number of frames it has in the waiting queue. It then passes the token to its known successor. After sending the token, the data terminal equipment listens to any subsequent activity on the bus to make sure its successor is active and has received the token. If it hears a valid frame being transmitted, it knows that its successor has received the token correctly. If it does not hear a valid frame being transmitted after the slot time interval, it must take corrective actions.

The broadcast feature is good for gathering global status information, but it makes the implementation difficult since it can not take advantage of the writable message format to establish the coupling phase of the drafting algorithm as done previously on token ring networks. That is, the dynamic pairing strategies can not be applied to the token passing bus directly even though the token passing bus forms a logical ring. Therefore, the light-load processor issuing a draft request must wait until all heavily loaded processors in the network become token holders in turn, then respond with the draft-age messages to the light-load processor. For a token ring network consisting of N processors, suppose that the interprocessor propagation delay is T. The total time needed for the message to go around the ring is $N(T+1) + L/C$, where 1 is the 1 bit delay made by every token ring interface, L is the length of the message, and C is the channel tranmission rate. The second item in the above equation may be ignored since L is usually small, while C is large. Thus, the message delay time is approximately

$$N(T+1) \tag{7-1}$$

In the token passing bus, it takes $\tau + L/C$ for a light-load processor to issue a draft request. Then it needs another $\tau + L/C$ to pass the token to its successor. In the worst case where other processors are all heavily loaded, the time interval from a light-load processor issued a draft request until it receives all draft-age messages (under the assumption there is no other message transmission) is $2N(\tau + L/C)$. In the best case where only one heavy-load processor is in the network, the time interval is $(2+(N-2)+2)(\tau + L/C)$. The average time interval is

$$(1.5N+1)(\tau + L/C) \tag{7-2}$$

Consequently, the performance of the drafting algorithm on the token passing bus with respect to that on the token ring network is determined by the value of

$$\frac{N(T+1)}{(1.5N+1)(\tau+L/C)} \tag{7-3}$$

In case the ratio is close to 1, direct implementation of the drafting algorithm on the token passing bus networks can be made, achieving the same level of performance as that on the token ring networks. However, the value of τ is usually greater than T since the propagation delay τ deals with the entire length of the bus, but the propagation delay T only relates to the interprocessor distance. Therefore, the performance of the drafting algorithm on the token passing bus is worse than that on the token ring networks.

8.2. Implementation Considerations on CSMA/CD Networks

The DCRLab (Distributed Computing Research Laboratory) at Michigan State University has a number of Sun workstations running UNIX 4.2 BSD and interconnected through a CSMA/CD Ethernet. The system is facilitated with the primitives for interprocesses communication. These primitives provide the possibility of implementing the drafting algorithm.

Protocol implementation and operating system

To implement an algorithm on an existing computing system is to add a protocol to an operating system. Generally speaking, it may be realized in three ways. The protocol may be in a process provided by the operating system, or it can be part of the kernel of the operating system itself, or it can be put in a separate communications processor or front end machine [Clar82].

The process is the abstraction which most operating systems use to provide the execution environment for user programs. A very simple path for implementing a protocol is to obtain a process from the operating system and implement the protocol to run in it. In this way, kernel modifications are not required, and the job is done by someone who is not an expert in kernel structure. Unfortunately, putting a protocol in a process has a number of disadvantages, related to both structure and performance. First, process scheduling causes a significant time delay which not only decreases the performance of the protocol but also postpones the response to the client. Structurally, the protocol may provide some services. For example, the data streams are normally obtained by going to special kernel entries. It may be impossible to let a program read data from a process. It is usually the case that special kernel modification is necessary to achieve this structure, which defeats the benefit of implementing the protocol in a process.

There are advantages to putting the protocol package in the kernel. It is reasonable to view the network as a device, and device drivers are traditionally contained in the kernel. The process scheduling problem can be alleviated because the code of the protocol is put inside the kernel. However, network protocols have a special characteristic which is different from other devices. The protocols require timeouts to protect themselves from any unpredictable exceptions. The kernel often has no facility to provide timer events. The only available mechanism is the interrupt. From the operating system point of view, it does not require too much time to handle an interrupt. But the protocol may perform some actions which are too complex and time consuming. Thus, in turn, causes a long time interrupt. The system scheduler has no control over the time used by the protocol. If a large number of communications is involved, all of the time may be spent for the interrupts. The system is actually killed. On the other hand, the interrupt handler does not provide a variety of important system facilities. The obscure bugs of the protocol implementation may not only kill the protocol, but also the entire operating system. In addition, the kernel address space is usually too small to contain the protocol code. This constraint does not allow an effective and general implementation of a protocol. A protocol tied to an operating system is a troublemaker whenever the operating system requires change.

The third way is to put the entire protocol package into a separate communication processor. The package is independent of existing operating systems which may differ from host to host in the system. Thus, an easier implemented protocol may be employed by different hosts. Similarly, there are also problems; such as, the need for an extra communication processor, the interface between it and the host as a protocol which still causes problems as mentioned above.

Environment of UNIX 4.2 BSD

The right way to implement a protocol, therefore, is to layer the protocol along with network layers. In the ISO OSI (International Standard Organization's Open System Interconnection) model, each layer provides many facilities supporting the higher layer(s) and using the services provided by the lower layer(s). The bottom-most layer is within the kernel, starting with a device driver or local network driver, then IP (Internet Protocol) and TCP (Transmission Control Protocol), eventually reaching the user. These existing system facilities may be used to implement a new protocol.

Berkeley UNIX 4.2 BSD offers extensive network services for doing IPC (InterProcess Communication) [KiNi86], a comprehensive tool for processes at different processors of the network to be able to communicate and cooperate with each other. Their native network transport facility is the DOD TCP/UDP/IP protocol family. The standard interface between application programs and transport protocols allows implementation of the protocol into processes obtained from the operating system, which are distributed on different processors. The network facilities correspond to a portion of the session layer and all of the transport and network layer of the OSI model. The transport layer normally includes the aspects of reliable transfer, data sequencing, flow control, and service addressing. Reliability is usually reached by explicit acknowledgement of data delivered. Sequencing may be handled by attaching a sequence number to each message. The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer.

In UNIX 4.2 BSD, the process is represented by three memory segments: the ***text segment*** containing code and constant data, the ***data segment*** containing variables, and the ***stack segment*** holding the process stack. Each process has a private ***descriptor table*** through which it accesses the rest of the system. Each descriptor is a handle allowing the process to reference objects such as files, devices, and communication links (sockets). The basic method to create a process in UNIX is to invoke the system call *fork()*.

The IPC facilities provided in 4.2 BSD have been designed as totally independent subsystems. The IPC allows processes to rendezvous in many ways. The basic building block for communication is the *socket*. A socket is a bidirectional endpoint of communication to which a name may be bound. To create a socket, a call is invoked:

```
sk = socket(domain, type, protocol);
```

requiring three parameters. The domain refers to the area the socket works on. For example, UNIX domain (specified by AF_UNIX) is for socket working on one host. Internet domain (specified by AF_INET) is for the communication between hosts. Each socket in use has a *type* and one or more associated processes. Three types of sockets are available. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. A *datagram* socket supports bidirectional flow of data with record boundaries, which is not sequenced, reliable, or unduplicated. A *raw* socket is normally datagram oriented. The exact characteristics are dependent on the interface provided by the protocol. A raw type socket is not used. A socket, also, associates with a particular protocol. Either the user specifies the protocol or the system selects an appropriate protocol (put a value 0). This call returns a value of sk which is an index pointing to an entry in the caller process's descriptor table.

A socket is created without a name. A name is an address. In the internet domain, a address contains an internet host's address and a port number. The name can be specified by user :

```
struct sockaddr_in name
name.sin_family = AF_INET;
name.sin_addr.s_addr = host_address;
name.sin_port = port_number;
```

If user knows only the name of the host, then the address of the host may be found by the *gethostbyname(host_name)* call. The port_number may be set 0 and the system will assign an available port to the name. The connection between the entry of the descriptor table and the name is established by the call ***bind*** :

```
bind(sk, name, name_length);
```

The name provides a point which may be used to connect the host with other hosts in the system. The communication between two hosts could be based on either connection oriented or connectionless oriented. In the former case, a *connect* call will establish a connection :

```
connect(sk, &remote_name, remote_name_length);
```

Then data transfer is done simply by calling *write* and *read*

```
write(sk, buffer, buffer_length);
read(sk, buffer, buffer_length);
```

or *send* and *recv* instead.

```
send(sk, buffer, buffer_length);
recv(sk, buffer, buffer_length);
```

The connectionless data transfer is usually used for datagram. To send data, the *sendto* primitive is available :

```
sendto(sk, buffer, buffer_length, flags, &des_name, des_name_length);
```

The des_name is the name of the destination host the data is expected to reach. To receive data, the *recvfrom* is usable :

```
recvfrom(sk, buffer, buffer_length, flags, &sou_name, sou_length);
```

There are other primitive alternatives for the different calls [LeJF83], [LeFJ84]. Summarized, the steps needed for data transfer are :

- create a socket
- find a name
- bind the socket and the name
- establish a connection (or use connectionless)
- transfer data

Implementation Considerations

The drafting algorithm balances the work load by using the process migration mechanism based on the predefined internal load states of a processor. In other words, it involves two basic aspects: the definition of the internal load states and the mechanism of the process migration. As mentioned in the previous section, the network communication facilities in 4.2 BSD are normally supported by a portion of the session layer and the entire transport layer. Thus, the drafting algorithm is implemented in the application layer in a user transparent manner. The implementation of the drafting algorithm on the CSMA/CD UNIX 4.2 BSD environment gives rise to several considerations.

The internal load states of the host

The internal states deal with the system kernel. At the first stage of the implementation, it is better to avoid the aspect of touching the kernel. Fortunately, the UNIX command *w* gives the average load value in previous 1, 2 and 5 minutes, respectively. Thus, these average load values are used to determine the internal states of the host without touching the kernel by predefining threshold values of heavy, normal and light load states. However, the migrant process is selected only from the new arrival instead of the first process in the schedule queue since the process table in the kernel may not be accessed. As shown in Figure 7-2, the model of the bidding algorithm is simple. It is possible to implement the bidding algorithm without touching the kernel. From the implementation point of view, a bidding algorithm armed with internal states can be implemented.

The process main()

A process *main()* can be built up on the top of the user login process. It accepts the command that is user typed in from the keyboard, then sends this command to a filter. All possible commands issued by users can be distinguished as migratable or unmigratable. For example, *cp, mv, rm, cd* etc. are locally processed since all resources involved reside only in the local. Commands like *cc, lpr, itroff and a.out* without interactive operations etc. might be migrated and remotely processed. The filter figures out the current load of the host and the migratable possibility of the command. If the current load of the host is heavy and the command is migratable, the main() begins sending a request to the daemons on the other hosts by using datagram facilities. Otherwise, if the command is local processed only, then the filter calls the system subroutine *system()* and takes the command as the parameter to execute the user command locally (see Figure 8-1).

```
main()
{
        get_host_name(my_host_name);
        get_host_addr(my_host_addr);
        open_ctl_sk(my_host_addr); /* a datagram socket */
        open_mig_sk(my_host_addr); /* a stream socket */
        while(TRUE) {
                readmask = 1 << mig_sk | 1;
                select(20, &readmask, 0, 0, &tv);
                /* receive a command from key board */
                if ((readmask & 1) > 0) {
                        read(0, buf, BUFSIZE);
                        if (filter(buf) == REMPROC) {
                                if (couple_remote()) {
                                        mig_command(buf);
                                        return;
                                }
                        }
                        system(buf);
                }
                /* receive the result from the remote processor */
                if ((readmask & 1 << mig_sk) > 0) {
                        recv(mig_sk, buf, BUFSIZE);
                }
        }
}
```

Figure 8-1 The main() of the implementation

where, REMPROC is the state for allowing remote process. The couple_remote() calls a procedure *ctl_comm()* described below and through a socket called ctl_sk opened by the procedure *open_ctl_sk()* to communicate with the remote daemon for coupling a remote host and then migrates the command to it through another socket named mig_sk which is a stream type socket and dedicated to migration. In realizing that the system possessed has a file server which can be accessed from all hosts in the system, the migration is performed by passing the command and the file name (the address of the file) instead of the entire file contents.

The I/O multiplex is chosen to implement the main() to make it powerful and flexible. In this way the main() receives the both command from the user's keyboard and the result sent back from the remote hosts.

A daemon for the control

Every processor has a daemon for the purpose of receiving and handling control messages. The request received by daemons in other processors is sent to the procedure *process_request(request, response)*, then the daemon sends the response back to the host initiating the request. Figure 8-2 sketches the main parts of the daemon,

```
daemon()
{
        gethostname(hostname, &name_length);
        for (;;) {
                recvfrom(daemon_sk, (char *)&request, sizeof(request), 0,
                        &from_addr, &from_size);
                process_request(&request, &response);
                sendto(daemon_sk, (char *)&response, sizeof(response), 0,
                        &request.ctl_addr, sizeof(request.ctl_addr));
        }
}
```

Figure 8-2 The sketch of the daemon()

where request and response are declared in globa.h file. They carry the addresses for communication purpose.

In order to task the control functions, the daemon maintains a double linked list which is a modified load table of the drafting algorithm, several manipulating procedures, and abundant control types to handle all possible controls, such as BROADC_H (broadcast_heavy), ANNOUN_W (announce_winning), LEAVE_C (leave_coupling), DELETE_R (delete_record), etc. The response of the remote host is carried by the data structure *response*; it is sent back by the daemon().

Control communication

A procedure *ctl_comm()* plays the role of interface between the couple_remote() of the main() and the daemon().

```
ctl_comm(des_host_addr, msg, type, response)
{
      msg.type = type;
      daemon.addr.sin_addr = des_host_addr;
      daemon.addr.sin_port = daemon_port;
      sendto(ctl_sk, (char *)&msg, sizeof(CTL_MSG), 0,
                  &daemon_addr, sizeof(daemon_addr));
      recvfrom(ctl_sk, (char *)&response, sizeof(CTL_RESPONSE), 0,
                  &from_addr, &from_size);
}
```

Figure 8-3 The interface procedure ctl_comm()

The procedure sends the request message to the daemon which can either be the daemon in the local host or in the remote site by passing a different destination_host_addr. Then it waits for the response coming back.

Transactions between the couple_remote() and the daemon()

The fundamental transactions between the couple_remote() and the daemon() include :

• The couple_remote() broadcasts the remote processing request to all other processors in the system by calling the ctl_comm() in the type BROADC_H. The broadcasting message contains the load value of the heavy-load processor. The message reaches the other host and is inserted into the double linked list in decreasing order of the load value. This results in the heavist host at the first position of the list. The host receiving the heavy-load message then puts its light load value into the response record if it is in light load state. Otherwise, it does nothing. The response record is returned to the heavy-load host.

• The heavy-load host receives all responses and selects one of them as the winning processor based on load values carried by the response records. Then, call ctl_comm() to send the ANNOUN_W message to the winning processor. The winning host issues LEAVE_C to its own daemon to clean the load table and passes an *accept* or *reject* response according to its current load state. If it is in the light load state, it sends accept and calls listen() on its mig_sk socket waiting for the caller's connection.

• The heavy-load host receives the accept response, initiates connect() at its own socket mig_sk and connects to the winning host, then migrates the command of the remote processing. The reject response causes the function couple_remote() to return a FALSE, then the main() calls system routine system() to execute the command locally. Any exception is handled by calling DELETE_R type of request.

This implementation reveals that the UNIX 4.2 BSD operating system provides a good environment for adding new protocols in the application layer. The difficulty lies in the kernel change. The drafting algorithm is more difficult to implement than the bidding algorithm, and was not implemented. The queue-length-determined internal state mechanism was also not implemented.

CHAPTER 9
CONCLUDING REMARKS AND FUTURE WORK

Load balancing is an important mechanism in distributed systems for improving the performance of the whole system. This study reveals the dynamic characteristics of the distributed systems, discuses the design considerations of the load balancing algorithms, and compares the performance of the various load balancing strategies to investigate an efficient dynamic scheduling scheme, called the distributed drafting algorithm.

9.1. Summary

The following results were achieved through this research:

(1) The unpredictable interprocess message transit delays of the distributed systems force control algorithms to work on an approximation of the global system state. The more accurate the global state gathered, the more communications required, therefore, the stronger effects the unpredictable delays show. The unavoidance of the asynchronization of the global actions produced by the control algorithms with the concurrent local actions causes anomalies. These facts create the difficulty of designing protocols and algorithms. The protocols and algorithms must compromise between two contradictory goals: maximize the processor utilization and minimize the communication overhead, and must consider the complicated exception handling.

(2) Generally speaking, the phenomenon of imbalance exists in distributed systems. This phenomenon degrades utilization of processors, and therefore, the performance of the whole system. The dynamic process migration is a useful tool for load balancing to alleviate the unbalancing problem. The distributed drafting algorithm is an efficient load balancing algorithm which is network

topology independent and accommodates dynamically changing system behavior. It satisfies the design requirements of the dynamic protocols mentioned in (1).

(3) Figures 5-3 and 5-5 showed simulation results in the point-to-point network consisting of 5 processors under the nonprobabilistic balancing situation and the probabilistic balancing situation, respectively. These figures indicate that the distributed drafting algorithm strongly improves the system performance over the situation without load balancing. The simulation results in Figure 5-4 also show that the drafting algorithm is suitable for the networks possessing the high transmission rate and low error rate.

(4) The drafting algorithm overcomes the drawbacks of the bidding algorithm by introducing internal load states and letting the light-load process initiate migration. The qualitative analyses point out that the drafting algorithm avoid the wait-while-idle, the unfairness and the unnecessary process migrations which are the inherent problems of the bidding algorithm. The quantitative simulation results of the token ring environment indicate that the drafting algorithm reduces the mean response time of the bidding algorithm around a half under the heavy load situation (up to total arrival rate of 8.0 for a token ring network consisting of 10 processors shown in Figure 7-3).

(5) The underlying network topology of a distributed system affects the implementation strategies, design parameters, and performance of load balancing algorithms. On the token ring network, we combined the heavy-initiated concept with the drafting algorithm to improve the performance. On the CSMA/CD network, we used other measures of the internal states and migration only of new arrival processes to avoid touching the kernel of an existing operating system which we used to implement a load balancing algorithm in a CSMA/CD environment.

(6) The dynamic pairing concept comes from the drafting algorithm, and studies the methods used to select a pair among the possible couples by considering the trade-off between the highest performance and the easiest implementation. Based on the features possessed by the token ring network, a writable message format is designed as a convenient vehicle to carry the negotiation information when implementing the dynamic pairing algorithms in the token ring network environment. The comparison of the simulation results (Figure 7-1) shows the mean response time of the min/max algorithm is about one of fourth of the first match algorithm when the total arrival rate reaches 8.0.

9.2. Future Work

We have studied the distributed drafting algorithm in different network environments. The results of the simulation indicate that the drafting algorithm approaches the M/M/N queueing model to a certain extent, and it outperforms the conventional bidding algorithm. However, some problems still remain as open questions.

(1) Adaptive definition of the internal load states

The bidding algorithm triggered by a new arrival introduces the influence of the external work load fluctuation. This fluctuation not only degrades the performance of the system but also brings an unstable factor to the system. The distributed drafting algorithm introduces internal load states so that the triggering of the algorithm depends on the system states. The number of message exchanges is considerably reduced and the stability of the whole system increased. The introduction of internal states appears the essence of the drafting algorithm.

Obviously, the internal states in turn require a quantity that can be used as a measurement of the processor's load in the system. The number of waiting processes in a processor may be selected as an applicable parameter for the definition of the internal states. Unfortunately, this parameter does not take the size of the waiting processes into account. Meanwhile, the predefined three internal states may not be adjusted with the dynamic situation in the system. A more efficient and effective approach for defining the internal load states is expected. Efficient means less overhead; effective means more accurate. The min/max algorithm studied in chapter 6 is a partial solution of the dynamic definition of states because the migration from the heaviest processor to the lightest processor copes with the dynamic situation of the system. But the definition of the load states itself is still static.

(2) Load state of channels

As mentioned in Chapter 3, the load state of the channels is ignored in this study. Actually, the channel traffic seriously affects the control message exchanges and the process migrations. If channel traffic is heavy, it postpones the control message communications and the migrations, and increases the length of the phase transition period that reduces the effectiveness of load balancing. Meanwhile, additional channel traffic introduced by load balancing further increases the burden on the channels. Therefore, a quantity which may be used as a measurement of the channel's load is important. Considering the unpredictable property of the channel communication delay, the difficulty of finding this quantity is clear.

(3) Implementing the drafting algorithm on a UNIX-based system

Some basic implementation considerations of the drafting algorithm have been described in Chapter 8. More research is required on how to modify the kernel for accessing the process table, in order to fully implement the work described here.

(4) Expanding the drafting algorithm to heterogeneous distributed systems

Homogeneous distributed systems provide compatibility between the processors. This environment is easy for discussion of process migrations. Conversely, in heterogeneous distributed systems, the different processors employ different operating systems. Many problems, such as "What kind of processes can be migrated?", "How can a migrant process be migrated?", "After a migrant process has been migrated to a remote processor, how can the correct communication with the original process be kept?" etc., require further investigation.

(5) What if a distributed system consists of thousands of computers?

A light-load processor must broadcast a drafting request to thousands of other computers. A possible solution to reduce communications and migration overhead is to employ a hierarchical structure which groups some processors. Load balancing can be executed both within a group and between groups.

BIBLIOGRAPHY

BIBLIOGRAPHY

[AlLa79] Almes, G.T., and Lazowska, E.D., "The behavior of Ethernet-like computer communications networks," *Proc. of Seventh Symp. on Operating Systems Principles*, Pacific Grove, California, December 1979.

[BaMM76] Balanchandran, V., McCredie, J.W., and Mikhail, V.I., "Models of the job allocation problem in computer networks," *Proc. COMPCON Fall*, 211-214, 1976.

[BaCM75] Baskett, F., Chandy, K.M., Muntz, R.R., and Palacios, F.G., "Open, closed, and mixed networks of queues with different classes of customers," *J.ACM*, Vol.22, No.2, 248-260, April 1975.

[BoBu80] Bowen, B.A., and Buhr, R.J., "The logical design of multiple microprocessor systems," Prentice-Hall, 1980.

[BrFi81] Bryant , R.M., and Finkel, R.A., "A stable distributed scheduling algorithm," *Proc. of the 2nd Int'l Conf. on Distributed Processing*, 314-323, 1981.

[ChAb82] Chou, T.C., and Abraham, J.A., "Load balancing in distributed systems," *IEEE Trans. on Software Enginering*, Vol. SE-8, N0.4, 401-412, July 1982.

[ChAb83] Chou, T.C., and Abraham, J.A., "Load redistribution under failure in distributed systems," *IEEE Trans. on Computers*, Vol. C-32, No. 9, 799-808, September 1983.

[ChKo77] Chow, Y.C., and Kohler, W.H., "Dynamic load balncing in homogeneous two-processor distributed," *Computer Performance, Edited by K.M. Chandy and M. Reiser*, 39-50, North Holland Publishing Co., 1977.

[ChKo79] Chow, Y.C., and Kohler, W.H., "Models for dynamic load balancing in a heterogeneous multiple processor systems ," *IEEE Trans. on Computers*, C-28, No.5, 354-361, May 1979.

[ChHL80] Chu, W.W., Holloway, L.Y., Lan, M.T., and Efe, K., "Task allocation in distributed data processing," *Computer*, Vol.13, No.11, 57-

69, Nov. 1980.

[ClPR78] Clark, D.D., Pogran, K.T., and Reed, D.P., "An introduction to local area networks," *Proc. of the IEEE*, Vol.66, 1497-1517, Nov. 1978.

[Clar82] Clark, D.D., "Modularity and efficiency in protocol implementation," *Network Inform. Center, SRI Int., Rep. RFC 817*, July 1982.

[Come84] Comer, D., "Operating system design -- the XINU approach," Prentice-Hall, 1984.

[Dixo82] Dixon, R.C., "Ring network topology for local data communications," *IEEE Proc. of Computer networks, COMPCON fall*, 591-605, 1982.

[Efek82] Efe, K., "Heuristic models of task assignment scheduling in distributed systems," *Computer*, 50-56, June 1982.

[Ensl78] Enslow, P.H., "What is a distributed data processing system ?," *Computer*, 13-21, Jan. 1978.

[Farb73] Farber, D.J., et al., "The distributed computing system," *IEEE COMPCON Spring*, 31-34, 1973.

[Ferr78] Ferrari, D., "Computer systems performance evaluation," Prentice-Hall, Englewood Cliffs, N.J., 1978.

[FiSH79] Finkel, R.A., Solomon, M., and Horowitz, M.L., "Distributed algorithms for global structuring," *National Computer Conference*, 455-460, 1979.

[FoST78] Forsdick, H.C., Schantz, R.E., and Thomas, R.H., "Operating systems for computer networks," *IEEE Computer*, 48-57, Jan. 1978.

[GaLR84] Gao, C., Liu, J.W., and Railey, M., "Load balancing algorithms in homogeneous distributed systems," *Proc. of the 1984 Int'l Conf. on Parallel Processing*, 302-306, August 1984.

[Garc82] Garcia-molina, H., "Elections in a distributed computing system," *IEEE Trans. on Computers*, Vol.C-31, No.1, January 1982.

[Garg85] Garg, K., "An approach to performance specification of communication protocols," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 10, 1216-1225, October 1985.

[Ozsu85] Ozsu, M.T., "Modeling and analysis of distributed database concurrency control," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 10, 1225-1240, October 1985.

[GoNe67] Gordon, W.J., and Newell, G.F., "Closed queueing systems with exponential servers," *Operations Research*, Vol.15, 254-265, 1967.

[Grau82] Graube, M., "Local area nets : a pair of standards," *IEEE Spectrum*, 54-64, June 1982.

[GyEd76] Gylys, V.B., and Edwards, J.A., "Optimal partitioning of workload for distributed systems," *Proc. COMPCON Fall 76*, 353-357, 1976.

[Hals85] Halsail, F., "Introduction to data communications and computer networks," Addison-Wesley Publishing Company, Inc., 1985.

[Haye84] Hayes, J.F., "Modeling and analysis of computer communications networks," Plenum Press, NY, 1984.

[HeWC75] Herzog, U., Woo, L., and Chandy, K.M., "Solution of queueing problems by a recursive technique," *IBM J. Res. Develop*, Vol.19, No.3, 295-300, May 1975.

[HiLi80] Hillier, F.S., and Lieberman, G.J., "Introduction to operations research," Holden-Day, Inc., Oakland, California, 1980.

[Huan85] Huang, J.P., "Modeling of software partition for distributed real-time applications," *IEEE Trans. on Software Engineering*, Vol. SE-11, 1113-1126, IEEE, 10, 1985.

[Huan85] Huang, J.P., "Modeling of software partition for distributed real-time applications," *IEEE Transactions on software engineering*, Vol. SE-11, pp.1113-1126, IEEE, 10, 1985.

[Hwan82] Hwang, K., et al., "A UNIX-based local computer network with load balancing," *IEEE Computer Magazine*, 55-64, April 1982.

[HwBr84] Hwang, K., and Briggs, F.A., "Computer architecture and parallel processing," McGraw-Hill, Inc., 1984.

[IEEE84] IEEE Standard 802.5, *Token Ring Access Method and Physical Layer Specifications*, Feb. 1984.

[KiNi86] King, C., and Ni, L.M., "Interprocess communication in the UNIX 4.2 BSD environment," *Research Report*, Department of Computer Science, Michigan State University, 1986.

[Klei75] Kleinrock, L., "Queueing systems, Vol.I : Theory," Willey Interscience, NY, 1975.

[KlNO76] Kleinrock, L., Naylor, W.E., and Opderbeck, H., "A study of line overhead in the Arpanet," *Commun. ACM*, Vol.19, 3-12, 1976.

[Klei85] Kleinrock, L., "Distributed systems," *CACM*, Vol.28, No.11, November 1985.

[Lams80] Lam, S.S., "A carrier sense multiple access protocol for local networks," *Gomputer networks*, 21-32, April 1980.

[LaWo81] Lam, S.S., and Wong, J.W., "Queueing network models of packet-switching networks," *Research Report CS-81-06*, Department of Computer Science, University of Waterloo, Feb. 1981.

[LaPS81] Lampson, B.W., Paul, M., and Siegert, H.J., "Distributed systems - Architecture and Implementation," Springer-Verlag, 1981.

[Lann77] Lann, G.L., "Distributed systems - towards a formal approach," *IFIP Congress Proceedings*, 155-160, 1977.

[Lann79] Lann, G.L., "An analysis of different approachs to distributed computing," *First Intl. Conf. on Dist. Comp. Systems*, Huntsville, Oct. 1979.

[LeLe83] Leach, P.J., Levine, P.H., et al., "The architecture of an integrated local network," *IEEE, 0733-8716/83*, 1983.

[LeMu77] Lee, R.P., and Muntz, R.R., "On the task assignment problem for computer networks," *Proc. 10th Hawaii Int'l Conf. System Sciences*, 5-9, Jan. 1977.

[LeJF83] Leffler, S.J., Joy, W.N., and Fabry, R.S., "4.2 BSD network implementation notes," *Research Report*, Department of Electrical Engineering and Computer Science, University of California, 1983.

[LeFJ84] Leffler, S.J., Fabry, R.S., and Joy, W.N., "A 4.2 BSD interprocess communication Primer DRAFT of December 6, 1984," *Research Report*, Department of Electrical Engineering and Computer Science, University of California, 1984.

[Lind76] Lindgren, B.W., "Statistical theory," 3rd edition, Macmillan Publishing Co., Inc., 1976.

[Lium78] Liu, M.T., "Distributed loop computer networks," *Advances in computers*, Vol.17, 163-221, Academic Press Inc., 1978.

[LiRo84] Liu, M.T. and Rouse, D.M., "A study of ring networks," *Ring Technology Local Area Networks*, edited by I.N. Dallas and E.B. Spratt, North-Holland Pub. Co., 1984, pp.1-39.

[LiMa85] Livny, M., and Manber, U., "Distributed computation via active messages," 1985.

[LiMe82] Livny, M., and Melman, M., "Load balancing in homogeneous broadcast distributed systems," *ACM*, 1982.

[MaLT82] Ma, P.R., Lee, E.Y., and Tsuchiya, M., "A task allocation model for distributed computing systems," *IEEE Trans. on Computers*, Vol.C-31, No.1, January 1982.

[MeJS85] Meister, B., Jonson, P., and Svobodova, L., "File transfer in local-area networks: a performance study," *Proc. of the 5th Int'l Conf. on Distributed Computing Systems*, Denver, May 1985.

[MeBo76] Metcalfe, R.M., and Boggs, D.R., "Ethernet : distributed packet switching for local computer networks," *CACM*, Vol. 19, No. 7, 395-404, July 1976.

[NiHw81] Ni, L.M., and Hwang, K., "Optimal load balancing strategies for a multiple processor system," *Proc. of the 1981 Int'l Conf. on Parallel Processing*, 362-367, August 1981.

[NiAb81] Ni, L.M., and Abani, K., "Nonpreemptive load balancing in a class of local area networks," *Proc. of the 1981 Computer Networking Symposium*, 113-118, Dec. 1981.

[NiLi82] Ni, L.M., and Li, X., "Modeling and analysis of computer load estimation methods," *Proc. of the 1982 Int'l AMSE Conf. on Modeling and Simulation*, Paris, France, July 1982.

[Nil82a] Ni, L.M., "A dynamic job migrating strategy for homogeneous local computer networks," *Proc. of the 1982 Computer Science and Technology Conf.*, Newton, Mass., June 1982.

[Nil82b] Ni, L.M., "Load balancing design considerations for distributed systems," *Technical Report*, Department of Computer Science, Michigan State University, October 1982.

[Nil82c] Ni, L.M., "A distributed load balancing algorithm for point-to-point local computer networks," *IEEE 1982 Fall COMPCON*, September 1982.

[NiXG85] Ni, L.M., Xu, C., and Gendreau, T.B., "A distributed drafting algorithm for load balancing," *IEEE Trans. on Software Eng.*, Oct. 1985.

[NiGe86] Ni, L.M., and Gendreau, T.B., "An universal inter-process communication system for distributed computing," Department of Computer Science, Michigan State University, 1986.

[OuSS80] Ousterhout, J.K., Scelza, D.A., and Sindhu, P.S., "Medusa: An experiment in distributed operating system structure," *CACM*, Vol. 23, No. 2, 92-105, Feb. 1980.

[Pawl81] Pawlita, P.F., "Traffic measurements in data networks, recent measurement results, and some implications," *IEEE Trans. on Communication*, Vol. COM-29, 525-535, April 1981.

[Pete81] Peterson, J.L., "Petri net theory and the modeling of systems," Prentice-Hall, Englewood Cliffs, N.J., 1981.

[PoMi83] Powell, M.L., and Miller, B.P., "Process migration in DEMOS/MP," *ACM*, 1983.

[Ramc73] Ramchandani, C., "Analysis of asynchronous concurrent systems by Timed Petrinets," *Ph.D. Dissertation*, Massachusetts Inst. Technol., Cambridge, Sept. 1973.

[SaCh81] Sauer, C.H., and Chandy, K.M., "Computer systems performance modeling," Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Sech84] Sechrest, S., "Tutorial examples of interprocess communication in Berkeley UNIX 4.2 BSD," *Research Report*, Department of Electrical and Computer Science, University of California, 1984.

[Smit80] Smith, R.G., "The contract net protocol : high-level communication and control in a distributed problem solver," *IEEE Trans. on Computers*, Vol. C-29, No. 12, 1104-1113, December 1980.

[Stal85] Stallings, W., "Data and computer communications," 1985.

[Stan81] Stankovic, J.A., "The analysis of a decentralized control algorithm for job scheduling utilizing Bayesian decision theory," *Proc. of the 1981 Int'l Conf. on Parallel Processing*, 333-340, August 1981.

[StSi84] Stankovic, J.A., and Sidhu, I.S., "An adaptive bidding algorithm for processes, clusters, and distributed groups," *Proc. of the 4th Int'l Conf. on Distributed Computing Systems*, 49-59, May 1984.

[Stan85] Stankovic, J.A., "An application of Bayesian decision theory to decentralized control of job scheduling," *IEEE Trans. on Computers*, Vol.C-34, No.2, February 1985.

[StBo78] Stone, H.S., and Bokhari, S.H., "Control of distributed processes," *Computer*, Vol.11, No.7, 97-106, July 1978.

[Ston78] Stone, H.S., "Critical load factors in two-processor distributed systems," *IEEE Trans. on Software Engineering*, Vol. SE-4, No.3, 254-258, May 1978.

[Tane81] Tanenbaum, A.S., "Computer networks," Prentice-Hall, Englewood Cliffs, N.J., 1981.

[TaTo85] Tantawi, A.N., and Towsley, D., "Optimal Static load balancing in distributed computer systems," *ACM*, Vol.32, No.2, 445-465, April 1985.

[WaJu83] Wah, B.W., and Juang, J., "An efficient protocol for load balancing on CSMA/CD networks," *The 8th Conf. on Local Computer Networks*, Minneapolis, Minnesota, October 1983.

[WaRo66] Wallace, V.L., and Rosenburg, R.S., "Markovian models and numerical analysis of computer system behavior," *SJCC*, Vol.28, 141-148, 1966.

[WaMo85] Wang, Y., and Morris, R.J., "Load sharing in distributed systems," *IEEE Trans. on Computers*, Vol. C-34, No. 3, March 1985.

[Zimm80] Zimmermann, H., "OSI reference model - The ISO model of architecture for open systems interconnection," *IEEE Trans. on Communications*, 425-432, April 1980.