A UNIFIED ENVIRONMENT FOR DISTRIBUTED COMPUTING

By

Thomas Bernard Gendreau

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Department of Computer Science

1987

ABSTRACT

A UNIFIED ENVIRONMENT FOR DISTRIBUTED COMPUTING

By

Thomas Bernard Gendreau

A distributed system is a combination of a distributed architecture and distributed control algorithms. The potential advantages of distributed architectures include increased reliability, resource sharing, and computation speed-up. The distributed control algorithms are software which manages the distributed architecture in order to achieve the advantages listed above. The overall goal of the distributed control algorithms is to provide a unified service environment. The unified service environment provides a number of views or interfaces to the distributed system. These views may range from a simple menu driven interface in which the distributed nature of the system is completely hidden from the user to complex distributed programming environment.

Three problems that occur in the context of the development of a unified service environment include the development of an appropriate interprocess communication system, scheduling of user requests, and development of distributed applications. A universal set of interprocess communication primitives is proposed. This set of primitives includes the concept of a dynamic group which allows processes of a distributed algorithm to bind together for communication purposes. Two scheduling algorithms, bidding and drafting, are compared in an Ethernet based client/server environment of five workstations. The comparison is based on an emulation of these algorithms and our results indicate that the simpler bidding algorithm outperforms the drafting algorithm in this type of environment. An environment for investigating distributed algorithms called distributed game playing is proposed. Its relationship to problems in distributed computing is shown through a number of illustrative problems.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Lionel M. Ni for teaching me a great deal about computer science and how to do research in computer science. Without his knowledge and the time and guidance he gave me, this dissertation would not have been possible. He has provided me with a good role model for my future work as a computer scientist.

I would like to thank Dr. Lewis Greenberg, Dr. Abdol Esfahanian, and Dr. Raoul LePage for the time and effort they devoted to reading this dissertation.

A project such as a Ph.D. dissertation cannot be completed without the help of many people. In particular, I would like to thank Eric Wu and C.-T. King for many useful discussions and there help in keeping the Distributed Computing Research Laboratory running smoothly. To the many other faculty and students in the Computer Science Department who helped me learn so much about computer science I apologize for not having the space to list their names.

I would like to thank GTE for providing me with fellowship support during the last two years of my graduate studies.

I would like to thank my parents for all their years of concern and support.

Finally, I would like to thank my wife. Without her support and patience this work could not have been completed.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION TO DISTRIBUTED COMPUTING

Distributed systems have been rapidly gaining importance in the world of computing, especially with recent significant reductions in the cost of processors and memory, and with the wider availability of improved communication networks [Stan84, LuSw85]. Resource sharing, computation speedup, and reliability enhancement are major motivations in designing a distributed system. Major advantages of distributed systems include high system reliability, availability, extendibility, flexibility, and productivity.

The term *distributed system* has been applied to a wide range of system configurations. A distributed system is a combination of a distributed architecture and distributed control algorithms. A *distributed architecture* is a set of interconnected peer processors, where each processor has a fair degree of autonomy in managing its own system resources. Inter-processor communication is provided by a high speed and reliable communication medium. However, the inter-processor transmission time is non-negligible. Both local area networks and loosely coupled multiprocessors can be considered as distributed architectures [MeBo76].

Distributed control algorithms are software which manage the distributed architecture. The purpose of the distributed control algorithms is to realize the potential advantages of the distributed architecture. The distributed architecture presents the possibility of resource sharing, computation speedup and increased reliability, but without appropriate control software these advantages can not be achieved.

In the following sections we will discuss some basic issues associated with distributed systems. First, we present a general overview of distributed systems

design. Following that we will look at three issues associated with distributed systems design: communications, system management, and distributed applications. The last three sections of this chapter discuss the problems that are addressed in this dissertation, summarize the research contributions of this dissertation and present an outline of the dissertation.

1.1 Distributed Systems Design Model

The design of a distributed system is often structured as a series of layers. Each layer in the design provides a set of services to the higher layers. The most popular layered design is the International Standards Organization's Open System Interconnection (OSI) model (also known as the Seven Layer Reference model). Figure 1.1 shows the different layers of the OSI model. The physical layer defines the physical characteristics of the transmission of bits. The data link layer defines a set of services that allow two adjacent processors to communicate. A standard protocol to use in the data link layer is a sliding window protocol. The network layer provides services that allow any two processors in the network to communicate. In standard LANs, in which all processors are adjacent, there is usually not an explicit network layer. In point-to-point network the network layer will have to consider how to find a path or route from the source processor to the destination processor. The transport layer provides a first level of interprocess communication. It provides services that allow any two processes in the network to communicate. The session layer provides a higher level set of interprocess communication services. These services should be more flexible and reliable than those provided by the transport layer. The session layer should also provide a set of services that allow connections to be maintained over a long period of time (a session, hence the name). These services should be able to transparently handle problems like a broken connection. The presentation layer provides translation services to handle problems

associated with communication between heterogeneous machines and security features such as encryption. The services provided by the application layer are open-ended. All layers below the application layer are basically providing a set of communication services. The application layer is intended to provide a set of services that take advantage of the distributed nature of the system. This is the layer in which the primary algorithms that make the distributed architecture a distributed system will be implemented. Some general classes of services that could be provided at this layer include a unified service environment, a distributed operating system, and a distributed database.

The OSI model was designed from a communications point of view and the appropriateness of the OSI layers is still a hotly debated issue. For the purposes of our discussion we will use a slightly simpler model called the *distributed system design model.* The layers of the model are shown in Figure 1.2. We use this model because it provides a set of layers that are more appropriate for the discussion of a distributed system built using a LAN as its distributed architecture. The interprocessor communication layer corresponds to the lower three layers of the OSI model. The interprocess communication layer corresponds to the transport and session layers and the distributed application layer corresponds to the presentation and applications layers. This model provides a coarser division of services than the OSI model. This division of services is appropriate for a distributed system that will be used for both system development and distributed algorithm development.

Figure 1.3 shows a simplified view of a distributed architecture. The architecture consists of a number of processors attached to a communications medium. If the processors are all the same, then the system is referred to as *homogeneous*; otherwise, it is called a *heterogeneous* system. Attached to each processor may be a variety of other resources. These resources can include hard

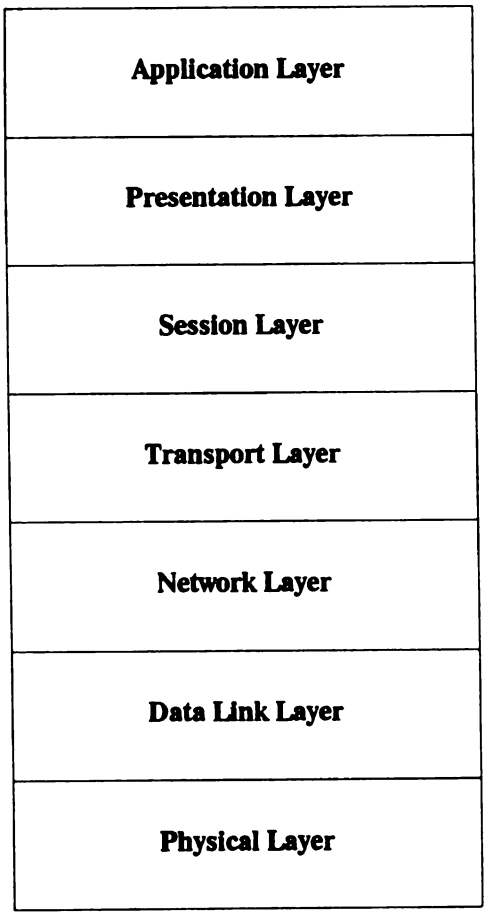

Figure 1.1. OSI Model

disks, tape drives, printers, and plotters. Two important design issues are the scheduling of processes in the system and how the file system will be constructed in the system. The scheduling of processes will greatly affect the performance of the system. A scheduling algorithm in a distributed system consists of a number of processes each of which manages a processor. The processes cooperate in the scheduling of processes in order to satisfy some criteria. Another important design consideration is how the disks drives will be used to provide a file system. If increased reliability and graceful degradation are to be achieved, then the file system will have to be constructed so that multiple copies of the files are stored on multiple disks. This is called replicating the files. If this is done, then files stored on a disk attached to a failed processor may still be accessible. Of course, appropriate distributed control algorithms must be developed to manage the replicated files. The management of other resources is a little easier since they can only be used by one user at a time and they are not critical to the performance of the system.

1.2 Communications Subsystem

The foundation of any distributed system is the underlying communication system. The general features of the communication system can be classified in two areas: interprocessor communication and interprocess communication. The interprocessor communication system determines how information is transferred between processors. The interprocess communication system determines how information is transferred between processes.

In a local area network environment the interprocessor communication will usually be based on one of three topologies: token ring, token bus, or contention bus. Each of these topologies allows for both one-to-one and broadcast interprocessor communication. One-to-one interprocessor communication means

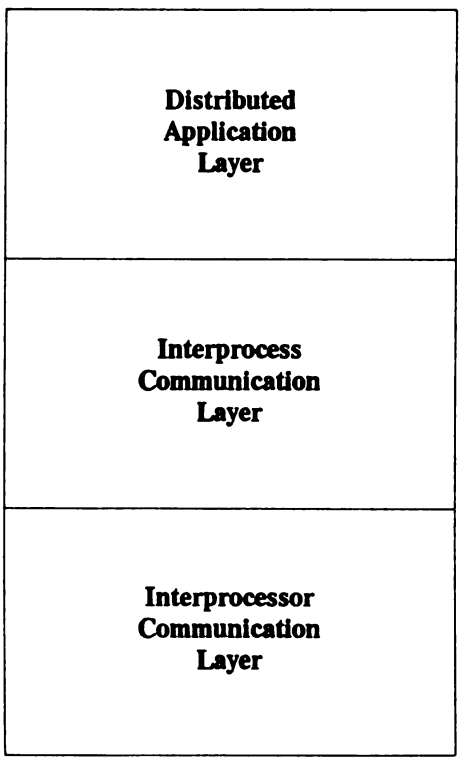

Figure 1.2. Distributed System Design Model

that one processor sends a message to another processor in the network. The destination of the message is indicated by a processor address in the message. A broadcast communication means that a message is sent from one processor to all other processors in the system. In each of the three LAN topologies, a broadcast message can be sent with one physical transmission of the message. The address in a broadcast message is a special address that indicates that it is a broadcast message. In a token ring protocol the message will travel around the ring and each processor will recognize the broadcast address and save the message. In the token bus and contention bus all stations can hear the message that is sent on the shared bus. When the stations hear the broadcast address, they will receive the message. In each of these topologies each processor is logically adjacent to all other processors so no routing algorithms are required.

Listed below is a simplified set of interprocessor communication primitives for local area networks. The ***send_packet*** primitive allows the processor to send a packet to another processor. The ***receive_packet*** primitive allows the processor to receive the next available packet. The ***broadcast_packet*** allows the processor to broadcast a packet to all other processors.

```
send_packet(source_id,destination_id,packet)
receive_packet(source_id,destination_id,packet)
broadcast_packet(source_id,packet)
```

In a loosely coupled multiprocessor the topology will usually be point-to-point. Some standard point-to-point topologies for loosely coupled multiprocessors are 2D mesh, 3D mesh, and hypercube. In a 2D mesh each processor is directly connected to its four neighbors. A 3D mesh is formed by stacking many 2D meshes and each processor is directly connected to its 6 neighbors. A hypercube is always given a size $N = 2^n$. The address of each processor is a bit string of n bits long. The neighbors of a processor are all processors with an address that differs by exactly one bit. Therefore each processor is connected to

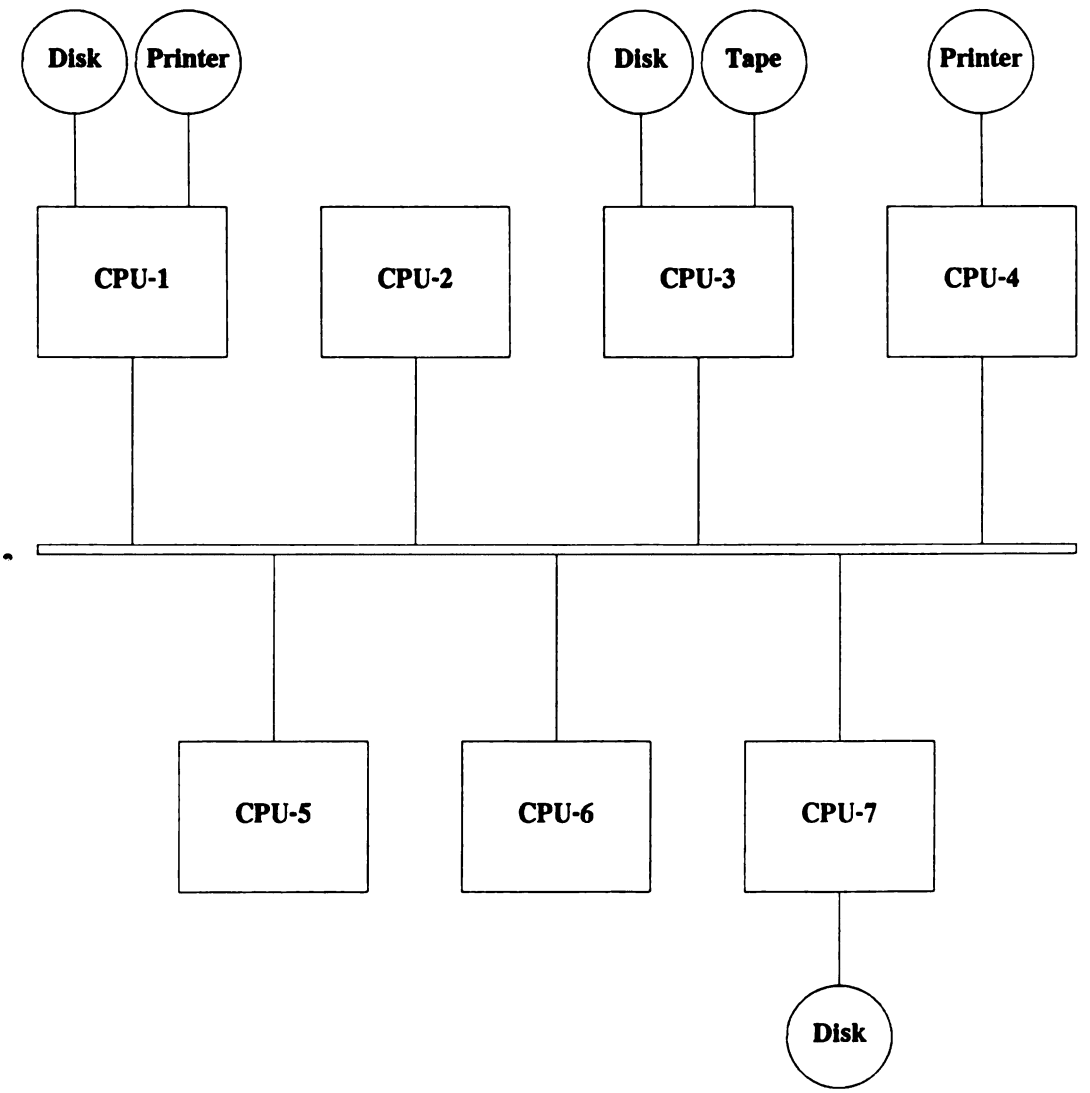

Figure 1.3. Distributed Architecture

n other processors. The point-to-point topologies require routing algorithms if a processor wants to communicate with a processor which is not adjacent to it. Also broadcasting of messages will require multiple transmissions of the message.

The quality of service provided by the inter-processor communication layer will greatly vary. The types of service that the inter-processor communication layer might provide can be put into three categories. *Guaranteed delivery service* guarantees that a packet will be delivered if the destination processor is accessible. If the destination processor is not available, the sender will be notified. *Acknowledged unreliable service* implies that packets may be lost in transmission but that the sender will always be correctly notified about the failure or success of the communication. The final class of service is *unreliable service.* With this type of service, packets may be lost in transmission and the sender will not always be notified (correctly) about the results of the transmission.

In a system in which multiple processes will be sharing each processor, interprocessor communication will not be sufficient. The sharing of processors will always be the case in LANs and may be the case in loosely coupled multiprocessors. The source and destination of each message will be a process and an interprocess communication system must be developed in order to facilitate this communication. This system will form the foundation of a distributed programming environment. The primitives defined in such a system determine the ease of communication in the system and provide the basic communication operations that will be used by application processes.

The general function of an interprocess communication system (IPC) is to create a set of mechanisms that allow location-independent exchange of information between any processes in the distributed system. In the 7-layer reference

model it is the responsibility of the transport and session layers to provide IPC. The transport layer builds a rudimentary IPC system on top of the (often unreliable) service provided by the network layer. The session layer implements a more easily used and flexible set of inter-process commands using the rudimentary IPC provided by the transport layer. The IPC commands provided by the session layer will be used by processes running at the presentation and application layers. In our distributed system design model, we will directly build a set of flexible and reliable IPC primitives on top of the inter-processor communication layer. The higher quality inter-processor communication service that can be provided in a LAN environment makes this feasible and the potential for developing a more efficient system makes this desirable.

1.3 Unified Service Environment

An ideal distributed system should provide an integrated view of the system and the services provided by the system. We call such a system a unified service environment. A *unified service environment* [NiGe85] provides a number of views or interfaces to the distributed system. These views may range from a simple menu driven interface in which the distribution of the system is hidden from the user to a complex set of distributed program development tools in which the user is provided with the means to take advantage of the distributed nature of the system. The goal of a unified service environment is not to hide the distributed nature of the system from all users. Rather, the goal of a unified service environment is to provide a set of system interfaces which are appropriate to a variety of users needs and to provide the distributed control algorithms to manage the system resources so that the various services can be provided in a fair, reliable, and efficient manner. In the following two sections we outline some areas of investigation in the development of a unified service environment.

1.3.1 System Management

Given a suitable IPC system we must now consider some of the higher level system management functions required for a unified service environment. These management functions are primarily concerned with the matching of resources and requests for resources. This matching must be done in a reliable, secure, efficient, and fair manner. The basic problems can be broken into three categories: scheduling, protection and reliability.

Scheduling algorithms try to match requests for service with the resources that can provide that service. The central scheduling problem is the scheduling of processes on processors. All request for service will eventually require some processor resources. A general scheduling algorithm will try to schedule a process on the appropriate processor taking into consideration issues such as processor load, fairness, other resource utilization (such as disks), and any other special information known about the process and processor characteristics.

Protection issues are concerned with verifying that a user is allowed to have access to the resources that have been requested and protecting the information that is transferred across the network. The protection of resources is usually done with capabilities. A capability to an object in the system represents a set of rights that a process has to an object. Capabilities have been used in uniprocessor systems for sometime but using them efficiently in a distributed system is still a research issue. Protection of information transferred across the network is achieved primarily through encryption. The encryption facilities may be provided by either the IPC layer of the distributed application layer. The protection of information that uses public networks is an important research issue.

A primary motivation for distributed system is the potential for increased reliability. The potential exist primarily because of the duplication of physical

resources. Appropriate distributed control algorithms which take advantage of the duplication of physical resource to replicate logical resources must be developed. These algorithms must enable the system to remain functionally complete in spite of failures by some resources. While the system might respond more slowly it should continue to provide the same set of services to users of the system. Earlier we described the concept of replicating files. This is an example of using the duplication of physical resource to provide a more reliable system.

1.3.2 Distributed Applications

Given a unified service environment we may want to consider the development distributed applications. A distributed application is an application in which the application designer or programmer takes advantage of the distributed nature of the system. There are three basic reasons why an application designer may want to implement an algorithm on a distributed system. The first reason is that the data required by the algorithm may already be distributed and thus it may be cheaper to do most of the processing where the data is generated. The second reason is to take advantage of the potential parallelism in an algorithm. Since a distributed system has many processors a great potential exist for computation speed-up through the use of parallel algorithms. The advantage of using a distributed system for parallel algorithm development will depend on characteristics of the distributed architecture and the characteristics of the parallel algorithm. For example, some application will be practical on a loosely coupled multiprocessor but not on a LAN. The third reason to use a distributed system for a distributed application is to provide a more reliable application. If the application has high reliability constraints a distributed system may be one way of satisfying these constraints. The amount of effort required by the application designer will depend on the support that is provided

by the distributed system.

1.4 Problem Statements

There are many problems that can be addressed in the area of distributed systems. In this dissertation we will address three problems: 1) the development of an interprocess communication system that can take advantage of the broadcast nature of many LANs and provides a good foundation for the development of distributed applications, 2) a comparison of two dynamic load balancing algorithms, bidding and drafting, in an Ethernet based LAN environment, and 3) the development of an environment for investigating distributed applications.

The quality of the interprocess communication system will be a primary element in determining the ease of developing both a unified service environment and distributed applications in a distributed system. The interprocess communication system must provide a flexible system that gives the users a choice in the cost and quality of service that is desired. The IPC system should also provide tools that allow processes of a distributed algorithm to indicate that they are related.

In general, application programs expect the inter-process communication to be highly reliable [ChMa84]. If an application process sends a message to another application process, it expects that the message will be delivered if the destination process exists. It has also been recognized that not all processes need the same level of reliability [SaRC84]. The degree of reliability influences the efficiency of communication. Therefore, it is useful to give an application program the ability to specify the reliability requirements of a particular communication transaction. Another issue related to reliability is "once-only-reception" [LiSc83, Shat84]. Once-only-reception implies that an inter-process communication system guarantees that a message will be delivered to the desti-

nation at most once. Providing this type of service generally decreases the efficiency of communication, but it does free the application program from some message management. A flexible inter-process communication system should provide the application program with the ability to specify once-only-reception.

A great deal of work has been done in the area of one-to-one inter-process communication. Representative work in this area includes [Hoar78, Rash80, BiNe84]. Hoare's CSP [Hoar78] provides basic synchronous send and receive. Messages are not buffered and the sender and receiver are blocked until the transaction is completed. Rashid's work [Rash80] allows processes to communicate through an object called a *port*. In essence a port is a buffer in which messages can be inserted (sent) and removed (received). Processes never send messages directly to a process. Multiple processes may have access to the same port. For example, for standard one-to-one communication between a process A and a process B, process A would have send access to port Y and receive access to port Z while process B would have receive access to port Y and send access to port Z. Associated with each port is a unique port id. *Remote procedure call* (RPC) [BiNe84] is another method by which processes may communicate information. Syntactically, a RPC looks like a standard procedure call. However, a RPC invokes a procedure running on a remote processor. The called procedure must be willing to accept a RPC. Values can be passed to the called procedure and results can be passed back to the calling procedure. RPC behaves like a synchronous send in which a reply can be returned. Other representative work includes [LeCo84, PoPr83, Spec82].

The area of scheduling in distributed systems has attracted a great deal of attention [Farb73, Stan84, NiXG85]. In order to build a unified service environment appropriate scheduling algorithms must be developed. These scheduling algorithms must address a number of issues. These issues include load

balancing, algorithm architecture matching, general resource utilization, and parallel algorithm scheduling. In this work we develop the important issues associated with distributed scheduling, and compare the performance of two load balancing algorithms through an emulation of these algorithms in an Ethernet based LAN.

Given a distributed system built on top of an LAN or a loosely coupled multiprocessor we would like to use that system to develop distributed applications. In order to investigate problems associated with distributed applications we would like to create an environment that can be used to create artificial problems to test the distributed system and to test and illustrate concepts in distributed computing. Very little work has been done in the investigation of distributed applications and the creation of an appropriate environment for investigating them should accelerate this investigation.

1.5 Summary of Research Contributions

In order to provide a flexible and reliable foundation for the development of a unified service environment we have designed an interprocess communication system for broadcast type networks. This system includes a wide set of primitives and arguments that should fit the needs of any distributed algorithm. A central feature of this IPC system is the concept of a dynamic group of processes. The dynamic group mechanisms allows processes participating in a distributed algorithm to bind together as a unit. It also allows processes to take advantages of the broadcast communication that is available in standard LANs. A description of the IPC operations is provided along with some implementation considerations.

Much of research in dynamic distributed scheduling is based on simulation results. In order to get some experimental results based on a real system we

have done an emulation of two load balancing algorithms on an Ethernet based distributed system of workstations. The first algorithm, called drafting [NiXG85], is a relatively new algorithm. The second algorithm, called bidding [Farb73], is an older algorithm that has been the basis for a great deal of work in distributed scheduling. The purpose of this work is to get experimental results for the drafting algorithm and to compare the performance of the drafting and bidding algorithms under a real system environment. The formal description of these algorithms will be given along with a description of the emulation design. Graphs describing the performance of these algorithms will also be presented.

In order to investigate distributed applications an environment for investigating distributed algorithm, call *distributed game playing* (DGP) is developed. The DGP environment allows us to investigate the basic features of distributed algorithms. These features include negotiation, remote state maintenance, and reliability. A major obstacle in investigating distributed algorithms has been the lack of an environment which is suitably rich to encompass the important aspects of distributed algorithms and sufficiently refined so that it can be used to highlight the distributed features of algorithms. Distributed Game Playing is such an environment. The DGP environment allows researchers the flexibility to highlight different features of distributed algorithms.

1.6 Dissertation Outline

The remainder of this dissertation is organized in the following manner. Chapter 2 describes the design of an interprocess communication system for broadcast networks. This chapter includes a classification of IPC operations, a description of a set of flexible IPC primitives that can be used as a foundation for distributed programming, and some implementation issues that must be

considered. Chapter 3 discusses the important issues in distributed scheduling. These issues include process migration, load balancing, and scheduling in heterogeneous systems. Chapter 4 discusses the emulation of two load balancing algorithms, bidding and drafting, in a LAN network environment. The major issues of implementing these algorithms in a LAN are discussed and comparisons of their performance under different characteristics are shown. Chapter 5 describes some characteristics associated with distributed algorithms. These characteristics include negotiation, remote state maintenance, and reliability. Chapter 6 describes an environment for investigating distributed algorithms called distributed game playing. The relationship between problems in distributed algorithms and the distributed game playing environment is shown and some illustrative problems are used to show how to use the distributed game playing environment. Finally, in chapter 7 we present concluding remarks and discuss issues for future work.

CHAPTER 2

AN INTERPROCESS COMMUNICATION SYSTEM

The foundation of a distributed computing environment is a set of flexible and reliable inter-process communication (IPC) primitives. The general function of IPC is to create a set of mechanisms that allow location-independent exchange of information between any two processes in the distributed system. In the 7-layer reference model, it is the responsibility of the transport and session layers to provide IPC. The *session layer* protocols provide a more easily used and flexible set of inter-process communication commands built on top of the rudimentary IPC provided by the transport layer. The transport layer builds a rudimentary IPC system on top of the (often unreliable) service provided by the network layer. The IPC commands provided by the session layer will be used by processes running at the presentation and application layers. In other words, the IPC primitives allow the application algorithm to express a requirement to send and receive information. In the distributed systems design model we have only one IPC layer that is responsible for interprocess communication. In the following discussion we are designing a system on top of an interprocessor communication layer.

With the increasing interest in distributed systems, distributed programming has attracted much attention in the past [Brin78, Lisk79, GoCK79, Feld79, Cook80, MaYe80, MaLe80, LiSc83, AnSc83, Geha84]. An important issue in designing a distributed programming language is to provide a flexible set of communication mechanisms. Many different distributed programming concepts or languages have been proposed or constructed. As usual, it is difficult to claim which language is better than the others. Details of the com-

parison of various distributed programming languages and their communication mechanisms are out of the scope of this chapter. Interested readers may find excellent surveys on this subject in [Silb80, Shat84].

Rather than addressing language requirements for distributed computing, this chapter concentrates on the construction of a flexible and powerful IPC system, the relationship of IPC to network protocols, and the implementation considerations of IPC. A flexible set of IPC primitives forms a basis for the construction of various communication mechanisms in distributed programming languages. The IPC primitives can be also implemented as system calls so that programmers can directly invoke these system calls in a conventional programming environment.

This chapter is organized as follows. A classification of IPC mechanisms is introduced in Section 1 which discusses the functions necessary to exploit inherent concurrency and to provide process synchronization. A universal IPC system which is able to support a unified service environment in a network-based distributed system is proposed in Section 2. This IPC system allows a flexible and convenient environment for the development of distributed application algorithms and distributed programming languages. In Section 3, we discuss concept of dynamic groups and primitives for group manipulation. In Section 4, we describe some general IPC layer services. More details about an implementation on top of the communication primitives provided in a Unix 4.2 environment can be found in [LGKN86].

2.1. Classification of IPC Primitives

An IPC mechanism consists of various types of *send* and *receive* operations. Processes communicate by sending and receiving messages. Various send and receive operations should be provided so that the programmer may exploit

potential concurrency in an algorithm and synchronize the communicating processes when it is necessary. In the following discussions, we try to classify various communication characteristics of IPC primitives.

2.1.1 One-to-One Versus One-to-Many Send

Most of the existing distributed programming languages or known IPC mechanisms support one-to-one inter-process communication. A comprehensive survey on one-to-one IPC in higher level languages can be found in [Shat84]. In a one-to-one send operation a process, called the *source*, sends a message to another process, called the *destination*. Only the source and destination processes know about the communication. However, in many distributed applications a group of processes are coordinated to solve a single task. These coordinated processes form a *dynamic group* and may reside in various physical processors. A process in a dynamic group may frequently wish to broadcast a message to those processes in the same group. Thus, a one-to-many send primitive is necessary to allow the message to be sent to a group of processes rather than a specific process.

The concept of cooperating processes forming a logical group can be very useful in the construction of a distributed system. A distributed system has various resources shared by all users. The users should be presented with a logically integrated view of the system, without needing to be aware of the physical distribution of the system. An automatic resource management scheme is needed to determine the best matching between requestors and resources. In such a unified service environment, the user only has to specify the desired service rather than a specific destination process. The processes (servers) that are able to provide a certain service can be formed into a dynamic group called a *server group*. Server groups can be used to provide a uniform encapsulation

mechanism for client/server applications. The encapsulation of a set of servers into a server group provides a means to hide issues such as scheduling of client requests and reliability concerns to be hidden form the client process.

Certainly, a one-to-many send operation can be achieved by performing many one-to-one send operations. However, this method does not take advantage of broadcast nature of many LANs. In a standard LAN environment, the network topologies are the bus and the ring [Stal84]. With either one of these topologies, each processor has a logically direct connection with every other processor. The *broadcast* nature of the network allows the protocols certain liberties with the way they handle messages, which is inherently different from store-and-forward networks. A one-to-many send operation can be designed to take advantage of this difference to make its implementation easier and its operation efficient.

In a send operation, we have to identify the source (sender) and the destination (recipient(s)). The sender is a process and thus is identified by a network-wide globally unique process id (PID). Since a process may be engaged in more than one communication, we have to identify the destination process(es). In a one-to-one communication, the recipient is another PID. In a one-to-many communication, the set of recipients is identified by a network-wide unique group id (GID). Note that because processes in a distributed system may move from one processor to another [NiXG85], the PID or GID may not be useful in locating the physical location of a process. It is the responsibility of the IPC layer to map a PID or a GID to the corresponding physical address(es) of the destination process(es).

2.1.2 Synchronous Versus Asynchronous Send

A *synchronous send* operation will block the sender until a response is

received from the recipient(s). In the case of a one-to-one synchronous send, the sending process waits for a response for the destination processes. In the case of a one-to-many send, the sending process waits for a response from one or more of the destination processes. Synchronous send is mainly used to allow processes to synchronize themselves at various points in an algorithm or when an immediate response from the destination process is required.

In an ***asynchronous send*** operation, there is no requirement for the sending process to wait for a response from the recipient. The sending process will continue processing in parallel with the actual transmission of the message over the communication channel. Thus, an asynchronous send allows an algorithm to exploit the concurrent nature of distributed systems.

In one sense, asynchronous send and synchronous send are equivalent in that one can be implemented in terms of the other in conjunction with the receive primitives. It is easier to implement synchronous send using a combination of asynchronous send and a blocking receive (to be described later). However, an asynchronous send may require a sizable intermediate message buffer in the recipient's side. For a synchronous send and a blocking receive pair, at most one message is outstanding at the recipient's side. Thus, exclusive use of the synchronous send can reduce the space requirements of an IPC system, at the cost of reducing the concurrency of the distributed algorithm.

2.1.3 Blocking Versus Non-blocking Receive

A recipient process indicates when it is ready to receive a message and accepts messages that are transmitted to it. If a message is available, it is stored into a location indicated in the receive command. In the case of a blocking receive, if a message is not available, the recipient process is blocked until a message arrives. If a message is not available when a non-blocking receive com-

mand is issued, the recipient process will continue. If it needs the message at a later time, it will have to issue another receive command.

A common structure used to allow processes to synchronize themselves is to have the sending process issue a synchronous send and the receiving process(es) issues a blocking receive. A non-blocking receive is commonly used when the recipient process expects messages to arrive from a variety of processes and those messages can arrive in any order. Other combinations of asynchronous or synchronous send and blocking or non-blocking receive are also possible. The appropriateness of a particular pairing depends on the behavior of each algorithm.

2.1.4 One-to-one Versus Many-to-one Reply

For a synchronous send operation, the recipient(s) has to explicitly reply to a message so that the sender process may continue. A reply message to a synchronous send process may carry information such as an acknowledgement, a result, or a go ahead indication. The reply can be considered as a special case of an asynchronous send operation. However, the implementation of the reply mechanism must be reliable so that the sender process won't be blocked forever.

A one-to-one reply is associated with a synchronous one-to-one send and a many-to-one reply is associated with a synchronous one-to-many send. A reply mechanism is a one-to-one communication which explicitly indicates the destination process (the process who issued the synchronous send). In the case of a one-to-one synchronous send there is only one reply which must be delivered and the implementation issues are similar to those of one-to-one send. However, in case of a synchronous one-to-many send, the sending process may need replies from a subset of the recipient processes. An important design issue is

how to make the delivery of multiple replies efficient and reliable.

2.1.5 Selective Versus Non-selective Receive

For a receive operation, the recipient may or may not know the identity of the sender in advance. If the sender is known, it is a selective receive; otherwise, it is a non-selective receive [Shat84]. A server process is likely to execute a blocking non-selective receive operation in which it is waiting for an unknown process to make a service request. The sender is identified after a message is received.

2.2. A Universal Set of IPC Primitives

Based on the various communication requirements, we classified the send and receive operations into different operating characteristics. Various IPC primitives may be constructed by taking some combination of these characteristics. In this section, we shall introduce a set of universal IPC primitives for design of distributed programming languages and distributed applications. We summarize the information presented in this section in three tables. Figure 2.1 shows the IPC commands that we define. Table 2.2 shows the arguments that these commands can use and Table 2.3a and 2.3b show how each command may use each of the arguments.

2.2.1 Asynchronous One-to-one Send

The *send* command is an asynchronous one-to-one send primitive. This command does not cause the process to be blocked. Since the process is not blocked, the status (the value of status indicates the result of the operation) argument does not contain any detailed information. The values that the status argument can get are *local_accept* and *local_reject.* A status value of

IPC Primitive Type	IPC Command
asynchronous one-to-one send	Send
synchronous one-to-one send	Send_and_Wait
asynchronous one-to-many send	Multicast
synchronous one-to-many send	Multicast_and_Wait
blocking selective receive	Receive
blocking non-selective receive	Receive_Any
non-blocking selective receive	Cond_Receive
non-blocking non_selective receive	Cond_Receive_Any
non-blocking reply	Reply
blocking reply	Reply_and_Wait

Figure 2.1 Interprocess Communication Primitives

Arguments	Explanation
spid	Source process id
dpid	Destination process id
gid	Process group id
message tag	A message type identifier used by the application layer to distinguish between different kinds of messages.
message	A string of bytes that holds the contents of the message.
length	The size of message. The number of bytes that are being sent or should be received. In a receive command, if the message has a record form, the length must be long enough to receive the next message.
form	The structure of the message. If the form is *record* then the message is treated as a separate entity. If the form is *stream* then the boundary between messages (sent by the same process with the same tag and to the same destination) is not important.
transaction_time	Used only by send type commands which indicates the time (absolute clock time) by which a message should be sent (put on the transmission medium).
priority	Used only by send type commands in which a static priorty assignment to the message by the application layer.
buffertime	If the argument is used by the send command , it indicates the maximum time that the message may be buffered at the destination. If it is used by a receive command, it indicates that the message that is received should not have been buffered longer that buffertime time units.
flush	Used with a receive command in conjunction with the buffertime argument. All messages that have been buffered more than buffertime time units should be discarded.
timeout	This argument is used by commands that cause the issuing process to block. It indicates that the process should become unblocked after timeout units if the message transaction has not been completed.
receive_discipline	Used only by receive type commands to indicate how the message to be received should be selected. Some possible receive disciplines are first in first out, last in first out, priority, and random.
minimum_deliveries	Used only by one-to-many type commands. It indicates that the message should be delivered to at least minimum_deliveries number of processes.
minimum_replies	Used only by the synchronous one-to-many type commands. The issuing process should be blocked until minimum_replies number of replies are received.
match	Used only by reply type commands. It indicates that this reply is a reply to the last message that was received.
security	Indicates the security level of a message that is being sent.
status	After a message transaction is completed, the status argument contains information about the results of the transaction

Figure 2.2. A list of arguments associated with IPC commands

IPC Argument Options						
	send	send_and_wait	multicast	multicast_and_wait	reply	reply_wait
spid	x	x	x	x	x	x
dpid	x	x	-	-	x	x
gid	-	-	x	x	-	-
message tag	x	x	x	x	x	x
message	o	o	o	o	o	o
length	1	1	1	1	1	1
transaction_time	o	o	o	o	o	o
buffertime	o	o	o	o	-	-
flush	-	-	-	-	-	-
timeout	-	o	-	o	-	o
receive_discipline	-	-	-	-	-	-
minimum_deliveries	-	-	o	o	-	-
minimum_replies	-	-	o	o	-	-
priority	o	o	o	o	o	o
security	o	o	o	o	o	o
status	o	x	o	x	o	x
match	-	-	-	-	o	o

x:- required arguments
o: optional arguments
-: argument may not be used
1: optional arguments that may be used if and only if the message argument is used

Figure 2.3a. Argument requirements and options for IPC commands

IPC Argument Options				
	receive	receive_any	cond_receive	cond_receive_any
spid	x	x	x	x
dpid	x	x	x	x
gid	-	o	-	o
message tag	x	x	x	x
message	o	o	o	o
length	1	1	1	1
transaction_time	-	-	-	-
buffertime	o	o	o	o
flush	2	2	2	2
timeouto	o	o	-	-
receive_discipline	o	o	o	o
minimum_deliveries	-	-	-	-
minimum_replies	-	-	-	-
priority	-	-	-	-
security	-	-	-	-
status	x	x	x	x
match	-	-	-	-

x: required arguments
o: optional arguments
-: argument may not be used
1: optional arguments that may be used if and only if the message argument is used
2: optional argument that can only be used if the buffertime argument was used

Figure 2.3b. Argument requirements and options for IPC commands

local_accept indicates that the IPC layer accepts the request while a status value of *local_reject* indicates that the IPC layer rejects the request. The IPC layer may reject the request because it is overloaded or because there are problems in lower layers of the communication system. The sender does not know if the *send* was successful unless an explicit acknowledgement is returned from the destination process. The status argument is optional.

The *send* command may optionally use the the arguments *message, form, transaction_time, priority, buffertime,* and *security.* Each message is a combination of a *message tag* and a *message.* The message tag forms the static part of the message and can be used to indicate the type of the message. In some cases sending only the static part of a message is sufficient. In other cases the message will have a static part and a variable part. The variable part is found in the message argument. In a send type command the message argument is a pointer to a string a bytes that form the variable part of the message. If the message argument is used, then the *length* argument must be used so the IPC system knows how many bytes of information are in the variable part of the message. The way in which the combination of message tag and message will be used will vary greatly from programmer to programmer. For example in the drafting algorithm [NiXG85] (see Chapter 4), the scheduling processes exchange messages about the current load of their processors. One way of expressing the passing of this information is to define a message having a static part such as "new load information" and a variable part that indicates whether the load is light, normal, or heavy. On the other hand the programmer could define three different static messages, "light load", "normal load", and "heavy load". Each of these types of message would not need a variable part because the static part communicates sufficient information. Allowing a process to send messages with different tags to the same process is similar to allowing a process to communi-

cate with another process over many logical channels and can save the application layer processes from some message management. Message tags are most useful in cases where a process may receive a variety of messages but at at a particular time is only willing to receive certain messages.

If the *form* argument is used, it will have the value of either *record* or *stream*. If the form argument is not used, the message is assumed to be a record message. Each record message is treated as a single entity and will be delivered as a single entity. A stream message is treated as part of a larger message. The boundaries between consecutive stream messages (with the same tag and same destination) are not important. The sender may send 100 bytes at a time while the receiver may receive 50 bytes at a time. In both stream and record messages the order of the messages is maintained.

The *transaction_time* argument is used by send type commands in real-time applications. The argument indicates that the message should be put on the transmission medium no later than the absolute clock time indicated in the transaction_time argument. The transaction_time argument is associated with the message until the message is sent. The receiving end does not know that a transaction_time was associated with the message. Reliable implementation of the transaction_time argument requires a data link layer that provides priority handling. The priority that would be associated with a message that used a transaction_time argument would be a dynamic priority. As the deadline for sending a message approaches, the priority associated with the message should increase.

The *priority* argument is used to indicate the relative importance of the message that is being sent. The priority assigned to a message is static. Unlike the transaction_time argument, the priority argument is associated with the message until it is received. The priority of a message may affect when it is

received (see the discussion of receive discipline below).

When the *buffertime* argument is used with a send type command, it indicates how long the message may be buffered at the destination. This gives the sender some control over the length of the messages existence. Note that since the sender cannot control how long it takes the message to arrive at the destination, the buffertime argument only provides a very coarse control over the age of messages.

The security argument indicates that the message contains sensitive data and should therefore be encrypted. The message is encypted at the source IPC layer and decrypted at the destination IPC layer. For very sensitive data special encryption routines should be provided by the application layer for explicit use by users.

2.2.2 Synchronous One-to-one Send

The *send_and_wait* command is a synchronous one-to-one send primitive. When the process issues a send_and_wait command, the process is blocked until a reply is received or a timeout expires. If a reply is returned within the timeout period, it is stored in the *reply_message* argument. Since the *send_and_wait* command blocks the sending process, more information can be found in the status argument. A possible set of values for the status argument in this case are *success, can_not_accept, can_not_send, can_not_deliver*, and *no_reply*. A status value of *success* indicates that a message was successfully delivered and that a reply was successfully received. A status values of *can_not_accept* indicates that the IPC layer will not accept the request. A status value of *can_not_send* indicates that the request was accepted by the IPC layer but because of problems in the lower layers the message could not be sent. A status value of *can_not_deliver* indicates that the message was sent but

could not be delivered to the destination. Finally, a status value of *no_reply* indicates that the message was received by the destination process but that a reply was not returned. For the synchronous one-to-one send the sending process is blocked until the transaction is successful or until the timeout expires (which ever occurs first). If the timeout expires, the status argument will contain one of the above values (excluding *success*) to indicate at what point the request failed. If a timeout argument is not used the system may provide a default timeout. The send_and_wait command may use the optional arguments, message, transaction_time, buffertime and security in the same way as the send command does. Messages sent with a send_and_wait command always have a *record* form.

2.2.3 Asynchronous One-to-many Send

The *multicast* command is our asynchronous one-to-many send primitive. Qne-to-many sending at the process level is intended to mimic broadcasting at the processor level. In order to do this we need a method to identify the processes which should be the destinations of a one-to-many send. In order to do this we will use the dynamic group concept. In Section 3 we will discuss some primitives for group manipulation. For now we only need to remember that the group_id argument indicates that a set of processes is the destination of a one-to-many send. If the destination group is an *open group* [ChZw85] then the source process does not have to be a member of the destination group. On the other hand, if the destination group is a *closed group*, then the source process must also be a member of the destination group. The status argument takes on the same values as in the asynchronous one-to-one send case. Since the sending process is not blocked by this command, the sending process will not know how many processes actually received the message unless explicit acknowledgements are sent from the destination processes.

The multicast command may use the same optional arguments as the asynchronous one-to-one send command. The status argument will have the same values as in the case of the asynchronous one-to-one send. In addition it may also use the optional argument *minimum_deliveries*. This argument is an integer which indicates the minimum number of processes that should receive the message. Providing reliable broadcasting is a difficult problem. When the broadcast is done at the data link layer, some of the processors may not receive the message (for example because of receive buffer overflow). Consequently, it may be expensive to guarantee that all processes in the destination process group receive the message. The *minimum_deliveries* argument allows the sender to indicate the degree of reliability that is required. If the minimum_deliveries argument is not used the message will be broadcast only once. Further comments on providing reliable broadcasting can be found in [ChMa84].

2.2.4 Synchronous One-to-many Send

The *multicast_and_wait* command is a synchronous one-to-many send. This command blocks the issuing process until *minimum_replies* number of replies have been returned or until a timeout expires. If the expected number of replies is received, the status argument will have the value *success*. If only some of the replies have been received when the timeout expires, then the status argument will have the value *not_enough_replies*. In this case the *minimum_replies* argument will contain the value of the actual number of replies that were received. In all other cases where the message transaction was not a success the status argument will take on the same values as in the synchronous one-to-one send case. In all cases, after the message transaction is completed the value of the *minimum_deliveries* argument will have the value of the number of processes that are known to have received the message. Note that the number of processes who actually received the message may be higher

than the number of processes that the local IPC layer knows received the message. Frequently, the *minimum_deliveries* and *minimum_replies* argument will have the same values. However, this is not required. The only requirement is that the minimum_replies argument is no bigger than the minimum deliveries argument. If the minimum_replies argument is smaller than minimum_deliveries, the IPC layer will continue to try and satisfy the minimum_deliveries requirement until it is satisfied or until minimum_replies replies have arrived (this is dependent on the method of reliable broadcasting that is used).

2.2.5 Selective Blocking Receive

The *receive* command is a selective blocking receive primitive. This command causes the issuing process to be blocked until a message is received or a timeout expires. The *receive* command indicates the process id of the process from which it expects to receive a message. The *spid* argument must have the value of the process id of the source process when the *receive* command is issued.

The receive command can optionally use the *buffertime, flush, receive_discipline, message tag*, and *message* arguments. When the buffertime argument is used by a receive command, it indicates that the message that is received should not have been buffered for longer than buffertime time units. This gives the receiving process some coarse control over the age of the messages that it receives. If the buffertime argument is used, the *flush* argument can also be used. The flush argument does not have a value. If it is used, it indicates that all messages that have been buffered for longer than buffertime time units should be discarded.

When a process issues a receive command it can restrict the set of mes-

sages that it is willing to receive by using the buffertime argument. Once this restriction is established the IPC layer must decide which of the eligible messages will be delivered to the receiving process. Traditionally the oldest eligible message would be delivered. In order to provide more flexibility to the application processes we provide a *receive_discpline* argument. This argument allows an application process to indicates how the next message should be chosen. Some possible receive disciplines are last in first out (LIFO), first in first out (FIFO), priority, and random. A LIFO discipline indicates that the youngest waiting message should be delivered. A FIFO discipline indicates that the oldest waiting (eligible) message should be delivered. A priority discipline indicates that the message with the highest priority should be delivered. A random discipline indicates that a message should be chosen at random from the eligible messages. If a receive_discipline argument is not used, then a FIFO discipline is assumed.

- The receive command can optionally use the *message tag* and *message* arguments. However, it must use at least one of these arguments. If the message argument is used, the *length* argument must also be used. If the message tag and message arguments are both used, the next available message (subject to the restrictions of the buffertime and receive_discipline arguments) with the correct tag will be stored in the message argument. If only the message argument is used, then the next available message (again subject to other arguments) will be stored in message. In this case the information stored in message will be both the message tag and the variable part of the message. If only the message tag argument is used, then a successful receive indicates that the appropriate message had arrived. If the message contains a variable part, that part is not received and is discarded by the IPC layer.

When the message argument is used with the receive command, the length argument must also be used. If the next message is a record message, then the length argument must be at least as big as the message; otherwise, an error results. If the message is a stream message, then the next length number of bytes will be stored in message (unless there are less than length bytes available). If the application process does not know the length of the next message, it can use the *message_length* command to find out the length. The message_length command can use the arguments message_tag, message, buffertime, and receive_discipline in same way as the receive command uses the arguments. The message_length command does not change any of the stored messages. All it does is to return the length of the next message that would be returned if the same arguments were used by a receive command.

The status argument must be used with the receive command. The values that the status argument can be given are *success, message_too_long,* and *failure.* A status value of *success* indicates that a message was received correctly. A status value of *message_too_long* indicates that the *length* argument given in the command was too short for the next message. This status value will occur only if the next message had a *record* form. Remember that a *form* value of *record* indicates that the message should be delivered as a single entity. A status value of *failure* indicates that no message was available within the timeout period.

2.2.6 Non-selective Blocking Receive

The *receive_any* command is a non-selective blocking receive primitive. When a process issues a *receive_any* command it is indicating that it is willing to receive a message from any process. The current value of the spid argument is not used. When a message is received, the process id of the source of that

message will be stored in spid. The receive_any command has the option of using a *gid* (group_id) argument. If this argument is used, it indicates that the process wants to receive a message that was multicast to the indicated group. In order to do this the issuing process must be a member of the indicated group. Except for the use of the *spid* and *gid* arguments, the *receive_any* command works like the receive command.

2.2.7 Selective Non-blocking Receive

The *cond_receive* command is a selective non-blocking receive primitive. If a message is waiting when the command is issued, then this command behaves exactly like the selective blocking receive. If a message is not waiting when the command is issued, then the status argument is set to *failure* immediately. This command never causes the issuing process to block.

2.2.8 Non-selective Non-blocking Receive

The *cond_receive_any* command is a non-selective non-blocking receive primitive. If a message is available, when the command is issued, then the cond_receive_any command works exactly like the receive_any command. If a message is not available when the command is issued, then the status argument is immediately set to *failure* and the process continues. The issuing process is never blocked by the cond_receive_any command. Non-blocking receive commands like asynchronous send commands allow the programmer to more fully exploit the potential parallelism in an algorithm.

2.2.9 Replies

The *reply* command is used to reply to both synchronous one-to-one and synchronous one-to-many sends. The reply commands behave like one-to-one send commands. Unlike the asynchronous send the reply command always

requires reliable service. The *reply* command does not block the replying process. The IPC layer should still use reliable service but replying process will not know if the reply was successfully received. The status argument for the reply command can be given the values of *local_accept* or *local_reject.* The meaning of these status values is the same as in the asynchronous one-to-one send case.

2.3. Group Manipulation Primitives

The concept of encapsulating a set of related processes within a group is important in designing an IPC system that can support distributed programming in a multiuser distributed system. In Section 2.2 we saw how a group of processes associated with a unique group id can be the destination of a one-to-many send. In this section we discuss some basic primitives (listed below) that can be used to manipulate groups.

```
Create_Group(group_id, process_id, group_type, group_structure,
        join_method, status)
Destroy_Group(group_id, process_id, kill_processes, status)
Join(group_id, process_id, timeout, status)
Invite(group_id, host_id, guest_id, timeout, status)
Leave(group_id, process_id, notify, status)
Remove(group_id, process_id, victim_id, status)
```

2.3.1 Creating a Group of Processes

The *create_group* command allows a process to create a new group. If a new group is successfully created the argument group_id will be given the value of the new group's unique group id. The process_id argument contains the process id of the process that issues the create_group command. Also the process which creates the new group becomes the first member of the group. The group_type argument indicates the type of the group. Some possible group types include, *open groups* and *closed groups.* These group types have been defined in Section 2.2. The *group_structure* argument can take the values *coordinated* or *peer.* In a coordinated group the creator of the group has special

privileges in accepting new members and in destroying the group. The coordinator of the group is said to own the group. In an peer group all processes are equals. The join_method argument indicates how a new process may become a member of the group. Some possible join_methods include ***know_gid***, ***sponser***, and ***election***. The status argument can be given the values of ***success*** or ***failure***. If the status value is success, then a new group was successfully created. If the status value is failure, then a new group is not created. A ***create_group*** command may fail because the creating process did not have premission to create new groups or because the system puts a limit on the number of groups that can be created and that limit has been reached.

2.3.2 Destroying a Group of Processes

The ***destroy_group*** command allows a process to destroy an existing group. In a coordinated group only the creator of the group may destroy the group. In a peer group, when a process issues a ***destroy_group*** command, all members of the group are asked to vote on the request to destroy the group. If a majority of the members vote to destroy the group, then this group will be destroyed. The ***kill_process*** argument is a boolean value. If ***kill_process*** is true, then all the processes in the group are destroyed when the group is destroyed. If ***kill_process*** is false, then only the binding of the processes to a group_id is destroyed when the group is destroyed. That is the processes continue to exist after the group is destroyed. The status argument will get the value ***success*** if the group is destroyed; otherwise, it will get the value ***failure***

2.3.3 Joining a Group of Processes

If a process wants to join a group it will issue a ***join*** command. When a process issues a ***join*** command it indicates the group id of the group it wants to join and its own process id. If the join_method is ***know_gid***, then the process

will automatically become a member of the indicated group (if it exists). In this way the group id acts as a capability to the group. If this method is used, group ids should be unforgible. If the join_method is *sponser*, then the joining process will be blocked until the coordinator invites (see below) the process to join the group. If the join_method is *election*, then when a process tries to join the group the other processes will vote to accept or refuse the request.

The status value of the join operation could be *success, reject, bad_group_id*, or *timeout*. A value of success indicates that the process is now a member of the indicated group. A value of reject occurs only if the join_method was election and means that the current members of the group voted to reject the join request. A value of bad_group_id means that the value of group_id is not the id of any existing group. A value of timeout means that an invitation from the coordinator or the election results did not arrive within the indicated timeout period.

2.3.4 Inviting a Process into a Group

In a coordinated group the creator of the group may invite a new process to join the group. In order to do this this creator executes an *invite* command. When a process issues an invite command it indicates the group_id of the group into which it is inviting the process, the process id of the process it is inviting, and its own process id. In order for the invite command to be successful the *guest* process must issue a *join* command. The status argument will take on the values *success* or *failure*. If the guest process had issued the appropriate join command, then the invitation will be successful and status will get the value success. If the guest process does not become a member of the group, then status will get the value of *failure*. Only the coordinator may invite a process into the group.

2.3.5 Leaving a Group

A process may leave a group at any time by issuing the *leave* command. The argument *notify* is a boolean variable. If the value of notify is true, then all members of the group are notified that the process is leaving the group. If the value of notify is false, then the process leaves the group without notifying the members of the group. The status argument will get the value *success* except in the case where the group_id is invalid in which case status will be given the value *failure.*

2.3.6 Removing a Process from a Group

In a coordinated group the coordinator may remove a process from the group (for example the process may be behaving in an unreliable or eratic manner) by using the *remove* command. The process_id argument is the process id of the coordinator and the victim_id is the the process id of the process that will be removed from the group. If the the victim process exists and is currently a member of the group, it will be removed from the group and status will get the value *success.* If the victim process does not exist or it is not a member of the group then status will get the value *failure.*

2.4. General IPC Layer Support for IPC Primitives

There are a number of ways in which the services described in the previous sections can be provided. One possible method is to create a complex IPC layer which provides the services. In this section we explore the requirements of such an IPC layer. We identify six classes of service that the IPC layer should provide: *one-to-one expedite, one-to-one batch, one-to-one transaction, one-to-many expedite, one-to-many transaction* and *one-to-many ordered.*

2.4.1 One-to-one Expedite Service

One-to-one expedite service is the simplest service that the IPC layer provides. This type of service allows a process to send a message to another process without connection establishment or acknowledgements. When a process indicates that it wants to send a message using this type of service, the IPC layer finds the location (processor address) of the destination process and sends the message. The IPC layer may also attach a checksum to the message so that garbled messages can be discarded by the destination. The destination does not send any acknowledgement and the message is sent only once (no retransmissions). Some implementations will provide buffering at the destination processor while others will discard the message if the destination process is not ready to receive it. From the IPC layer perspective this is the fastest and most efficient type of interprocess communication. The usefulness of this type of service will depend on the quality of the communication medium and the requirements of the application. If most messages are delivered correctly, the application can absorb the loss of some messages, and the application requires fast service, then this may be the best service for the application to use. One type of application that would find this type of service acceptable would be a real-time application that periodically transmits measurements taken by a sensor. Such an application wants the information transferred quickly and because the measurements are periodically updated, the loss of a few measurements will not be a problem.

2.4.2 One-to-one Batch Service

One-to-one batch service provides reliable transmission of a large quantity of data. This is the type of service that is found most frequently in commercial IPC layers such as those that support TCP or XNS. This type of service is provided in three phases: a connection establishment phase, a data transmission phase, and a connection termination phase. The connection establishment phase identifies that both ends are ready and establishes some parameters that are

used by the protocol. The data transmission phase involves the actual transfer of data. The data is broken into packets and a checksum is usually attached to each packet. The packets are acknowledged by the destination but there is not necessarily one acknowledgement per packet. Packets are retransmitted until they are received correctly. If the underlying communication medium remains functional, the data will eventually be transmitted correctly. During the data transfer phase, data and acknowledgements can be transmitted in both directions. The connection termination phase identifies that both ends have completed their data transfers and that all packets that have been sent have been correctly received.

A standard type of protocol that is used for one-to-one batch service is a *sliding window protocol.* In this protocol a window size is established during the connection establishment phase. The window size indicates the maximum number of unacknowledged packets that the sender may have outstanding. The sender can continue to send packets as long as this limit is not reached. If the limit is reached the sender will have to stop sending and wait for an acknowledgement. A single acknowledgement message can be used to acknowledge multiple packets (ideally the acknowledgements arrive frequently enough that the window size limit is not reached). In this way the number of acknowledgements can be reduced. Associated with each packet is a sequence number. This number is used to identify the packet during acknowledgements. There are two general forms of the sliding window protocol: *go-back-n* and *selective retransmission.* In the go-back-n method if a message is lost or garbled, then that packet along with all other packets that were sent after the lost packet will have to be retransmitted. In the selective retransmission method only those packets that are lost or garbled will have to be retransmitted. For more details on sliding window protocols see [Tann81].

2.4.3 One-to-one Transaction Service

One pattern that communication in distributed applications often follows is a request-response communication. In this type of communication the sender sends a messages and waits for a response from the destination process. The *send_and_wait* primitive follows this communication pattern. This communication usually requires a reliable sending mechanism. One-to-one transaction service is intended to provide efficient service for request-response type communication. This service can take advantage of the fact that after a message is sent a reply from the destination process is expected. If the reply is sent quickly enough, it can also serve as the acknowledgement. If a reply cannot be sent quickly enough, then an explicit acknowledgement will be sent. At the sending end, if a reply or acknowledgement is not received within a timeout period, the packet is retransmitted. Two important (and related) design parameters are the time the destination IPC layer should wait before it sends an explicit acknowledgement and the timeout period before the sender should retransmit the packet. With a careful choice of these parameters a reliable and efficient request-response type communication can be provided. The parameters may be chosen statically as part of the system design or dynamically based on information provided by the user (e.g., estimation of the time required before a response will be received).

2.4.4 One-to-many Expedite Service

In order to provide access to the broadcast nature of many LANs the IPC layer should provide one-to-many communication. The simplest type of one-to-many communication is *one-to-many expedite* service. With this type of service a process can broadcast a message to a particular group. The message will be broadcast only once and in general the message may be received by some

members of the group and not others (e.g., because of receive buffer overflow at some stations). Providing this type of service requires the IPC layer to know about the different process groups that exist in the system and the process ids of the local processes that belong to each group.

2.4.5 One-to-many Transaction Service

As in the one-to-one case one-to-many communication can form a request-response pattern. In this case a process broadcasts a message to a group and requires a response from one or more members of the group. This service needs to guarantee that at least N members of the group receive the message if N responses are required. Following the pattern of one-to-one transaction service, the IPC layer will use responses to a message as acknowledgements that the message was received. If a response is not received within a timeout period, the message is broadcast to the group again. In the case of one-to-one transaction service, if the destination could not send a response quickly enough it would send an independent acknowledgement. If this is performed in the one-to-many case, the communication system could be overloaded with acknowledgements. In order to alleviate this problem, we never have the destination processes send acknowledgements. Instead the IPC layer at the senders end will continue to retransmit the message until a response arrives. In order to control the number of retransmissions the time between successive retransmission will get larger for every retransmission. After a response is received, the length of the timeout period is reset and a retransmission is sent if another response is not received within the timeout period. This procedure is continued until N responses are received or a maximum number of retransmissions have occurred. This method may result in some unneeded retransmissions but the cost is much less that having destination processes send explicit acknowledgements. The values associated with the timeout are critical to the performance of this method. Setting

the timeout dynamically based on some user provided information should reduce the number of unnecessary retransmissions.

2.4.6 One-to-many Ordered Service

In one-to-many transaction service members of the same group may receive multicast messages in different orders. In some applications it is desirable to have all members of a group receive the messages in the same order. This provides a kind of shared state for the processes in the group that can reduce the complexity of the algorithm. An interesting method that allows all processes in a group to receive the group messages in the same order was proposed in [ChMa84]. In this method one IPC layer, called the token holder, acts as a funnel through which all group messages must pass. When a process multicasts a message to the group it expects to receive an acknowledgement from the token holder. If an acknowledgement is not received the sending process retransmits (rebroadcasts) the message. Note, each time the message is sent it is broadcast. Attached to each message is a process id and a local sequence number. This allows the receiving IPC layers to reject duplicates of the message. When the token holder receives the message, it sends an acknowledgement to the sender. The acknowledgement is also broadcast so that all IPC layers can receive it. The acknowledgement contains the process id of the sender, the local sequence number that sender attached to the message and a global (per group) sequence number that the token holder associated with the message. This global sequence number determines the order in which the messages should be given to the members of the group. Since all IPC layers can hear the acknowledgement, they can know the local sequence number. If an IPC layer (not the token holder), hears an acknowledgement which has a sequence number higher than the next sequence number expected, it will ask the token holder to retransmit the the messages that it did not receive.

The advantages of the above method are that once the token holder acknowledges the message, the sender may discard the message (the token holder keeps a copy and is responsible for retransmission). The number of acknowledgements will usually be much less than one per process in the group. In addition the sender can try to receive its own messages and can know in what order other processes received the message. This can be helpful information in some algorithms such as elections [KiGN86].

One issue that must be addressed is when can the token holder discard a message. That is, how can the token holder know when all the necessary IPC layers have received the message. [ChMa84] suggests that the (virtual) token be passed from one IPC layer to another. When the token is passed the new token holder must have received all the old messages that the old token holder has received. If the token is passed to every IPC layer before it repeats its path, then when an IPC layer becomes a token holder it can discard all messages it had receive before the last time it was a token holder. All of this messages must have been received by the other IPC layers since the other IPC layers could not have accepted the token if they had not received the messages.

CHAPTER 3

DISTRIBUTED SCHEDULING ISSUES

A central problem in distributed systems is the scheduling of processes onto processors. This problem is motivated by issues such as load balancing, parallel algorithm requirements, algorithm-architecture matching, and utilization of resources. Without a satisfactory solution to the distributed scheduling problem, the creation of a unified service environment will not be possible.

The basic decision that must be made by a distributed scheduling algorithm is whether to migrate a process from its current location (processor) or to leave it where it is. The process migration procedure will require that two processors, one the sender of the process and the other the receiver of the process, agree to move the process. The information that will be used to make the decision on a particular process migration will depend on the goals of the distributed scheduling algorithm. When a process is to be migrated, enough information about that process must be sent to the receiving processor so the process can be executed at the receiving processor. Frequently the receiving processor will have to return the results produced by the process to the sending processor.

A primary consideration of many distributed scheduling algorithms is load balancing. The overall performance of a distributed system can be improved if extreme differences in the loads of the processors can be smoothed. The goal of a load balancing algorithm is not to exactly even the load at each processor rather it should try to smooth (through process migration) those extreme differences in load that may result in underutilization of some processors (or resources in general). Other issues in distributed scheduling will be discussed

later.

Figure 3.1 shows an idealized view of the scheduling structure. The *user* block represents the source of work or processes. The *command interpreter* will determine if the process is migratable. In general we do not want to migrate processes that have a very short execution time. For standard user commands the command interpreter may have some information about the expected length of the job. For example a request to list the contents of a file is not a good candidate for migration while a compilation of a large program may be a good candidate for migration. For user programs the command interpreter will need some information from the user in order to make a good decision. The *scheduling agent* makes the global scheduling decision. It will determine along with its peer scheduling agents the place where a migratable job will be scheduled. The *file system agent* contains information about the distributed file system. The file system agent will provide information such as the individual disks loads and the locations of copies of file to the scheduling agent. The scheduling agent will make use of this information along with any information it has about the process that is being scheduled, the load of the other processors, and the static features of the system in making the scheduling decision. The *local scheduler* makes the local scheduling decision. If the local system is a single processor system, the scheduling decision may be something simple such as first come first serve or round robin. If the local system is a multiprocessor the local scheduler may have to make more complicated decisions.

3.1 Classification of Distributed Scheduling Techniques

The problem of scheduling in distributed systems has been addressed from a variety of perspectives. These perspectives can be classified as deterministic, probabilistic, and dynamic. In a deterministic [CHLE80, Ston78, Efek82]

approach a known amount of work must be distributed throughout the system. For example we may be given the fact that we have N processes that must be run and the running time of each process. The problem is to distribute the processes among the available processors so that all processes finish in the shortest possible time. More complicated approaches may assume more information. For example they may know a precedence relation between the processes and the communication behavior between the processes. In general this problem is NP-hard and heuristic methods must be developed. Results of this approach are primarily of theoretical interest and will not be further discussed. In general they are not applicable to a general LAN environment. In some cases they may be useful in a loosely coupled multiprocessor or a dedicated LAN.

In the probabilistic approach the nature of the work that arrives to the system is assumed to be defined by some probability distributions. For example we may be given probability distributions for arrival times and execution times. If the distributions are well known, this may be an acceptable approach. However, in general these distribution will not be known. Also, it has been shown in [NiXG85] that dynamic algorithms can still outperform a probabilistically balanced system under some conditions. The basic problem with the probabilistic approach is that it does not cope with system changes due to workload fluctuations.

In a dynamic scheduling approach [Farb73, Stan84, NiXG85] the scheduling agents cooperate in order to scheduling processes based on both dynamic and static information. The dynamic information is based on the current status of system resources such as processors, communication channels, and disks. The static information may include special processor features known by the system and process characteristics made known to the scheduling agent when a process is given to the scheduling agent.

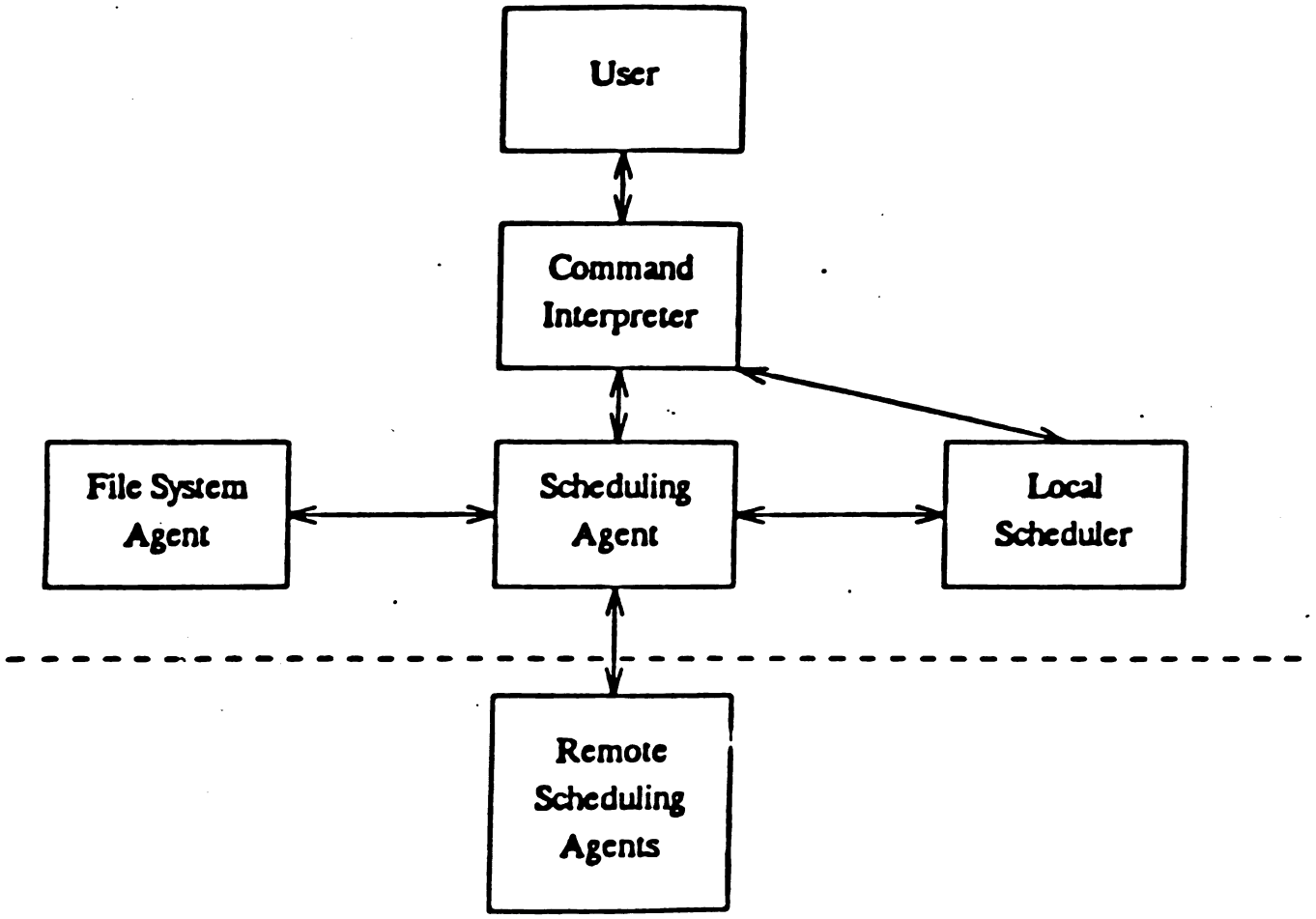

Figure 3.1. Ideal structure of scheduling agents

3.2 Heterogeneous Systems versus Homogeneous Systems

A distributed scheduling algorithm will have to consider the features of the machines in the system. If all machines are identical, then the system is called *homogeneous*. In the most extreme case the term identical implies that each processor is the same and is connected to the exact same set of resources. If a system is not homogeneous, it is called *heterogeneous*. In a homogeneous system, the scheduler does not need to consider any static features of the processors when it makes the scheduling decision.

Different levels of distinction can be made in determining if a system is homogeneous or heterogeneous. Some possible classifications that can be used include machine architecture, machine speed, and machine resources. A system that is heterogeneous at the machine architecture level has a distributed architecture consisting of a variety of machine types. For example we could have a system consisting of personal workstations, multiuser mini-computers and mainframes, vector processor machines, multiprocessors, and other special purpose architectures. In this case the distributed scheduling algorithm will need to consider matching processes with an appropriate architecture. A system that is heterogeneous only at the machine speed level has a distributed architecture in which the processors all have the same basic characteristics (e.g. they all use the same machine language) but the processors may run at different speeds. In a system that is heterogeneous at the machine resources level, all processors may be the same but those processors will be connected to different resources. For example some stations may have disks attached to them while other stations may be diskless.

A general distributed system will be heterogeneous at least at the machine resources level. In general a distributed scheduling algorithm for a unified service environment will have to take into account both the static and dynamic

features of the system. In order to make the best use of this information the scheduling agent should also know certain characteristics of the processes that are being scheduled. As a simple example, consider a process that indicates that it will access a number of files. One consideration that the scheduling agent should use in determining where to schedule the process is the physical location of the files that the process will access. In another case a process may indicate that it needs a special type of architecture. In this case the scheduling will need to contact other scheduling agents that control a processor that satisfies the processes needs.

3.3 Multiple Job Entry Points Versus Single Job Entry Point

In a distributed architecture based on an LAN there will generally be multiple entry points for processes. For example, in an LAN of workstations anyone of the workstations may be an entry point for work into the system. This makes it practically impossible for any processor to maintain a complete view of the total system load. However in a loosely coupled multiprocessor such as a hypercube there is only one place (the cube manager) where work can enter. Thus we naturally have a centralized point of control. In this case it may be wise to allow the cube manager to make some coarse scheduling decisions (especially with multiprocess jobs) and make only some finer modifications in a distributed fashion.

3.4 Multiple Process Jobs Versus Single Process Jobs

Most work in distributed scheduling (including our discussion in Chapter 4) treats processes in the system as if they were independent entities. In many systems this is a reasonable assumption. However, if the unified service environment is to provide an environment in which distributed applications can be written, then this is not a reasonable assumption. In a distributed application,

processes will have a certain relationship with other processes in the algorithm. This relationship can be described in terms of the concurrency relation between the processes and the communication relation between the processes. The concurrency relation indicates how much of the processes work can be done concurrently. For example, we could have two processes in the algorithm that do not communicate with each other and whose only purpose is to compute some result and send it to a third process. In this case the work of both processes can be done concurrently. At the other extreme we could have two processes that work in lock step with process A computing a result and sending it to process B and waiting for process B to do its work and returning a result back process A. In this case there is no concurrency between the processes.

The communication relationship between processes indicates the amount of information that is exchanged between processes. The more information that is exchanged, the closer together we would like to schedule the processes. In a sense the concurrency relation is related to communication as well. The concurrency relation represents the frequency of a process waiting for information from another process. Since in a distributed system we assume that information is transferred only through explicit communication, the concurrency relation is related to the frequency of communication. While the communication relation is related to the amount (e.g. bytes) of the communication.

If we are going to accurately schedule processes that are part of a distributed application, the scheduling agent will want to have information about the concurrency and communication relations. If the information is statically provided, then this problem is similar to deterministic or probabilistic methods discussed above. However, in a highly dynamic distributed algorithm it may be possible for these relationships to change during the life of the algorithm. In this case protocols will have to be developed that allow the application

processes to communicate with the scheduling agent. One possibility is to have the distributed application schedule itself in cooperation with the schedule agents. The scheduling agents will work from global knowledge of the system state and system goals while the distributed application will work from local knowledge of the best distribution (or scheduling) of the processes in the distributed application.

3.5 Preemptive Versus Non-Preemptive Process Migration

In choosing a job to be migrated an important consideration is whether jobs that have been started should be migrated. In general the answer is no. A job should be migrated only if it is expected that it could receive better response time than if it were not migrated. This is unlikely if the job is already being given service at the local processor. Another problem with migrating jobs that have already received service is that their whole memory image will have to be migrated. This could cause a great deal of communication overhead.

CHAPTER 4

EMULATION OF LOAD BALANCING ALGORITHMS

In Chapter 3 we introduced the concept of load balancing as an important issue in distributed scheduling. In this chapter we discuss two load balancing algorithms, ***bidding*** and ***drafting***, and look at the performance of these algorithms in an Ethernet based LAN of workstations. The performance measurements of the load balancing algorithms are generated by an emulation of the algorithms on our LAN system of Sun workstations. The standard method of evaluating dynamic load balancing algorithms is through simulation. The simulation will make some assumptions about the network configuration, arrival rates, and service rates. Sometimes the simulation will also consider the time required to send messages between processors. The positive features of a simulation are that it can be run on a single computer, the techniques for simulating such systems are well known and the parameters of the system are completely under the control of the person doing the simulation. The primary drawback of a simulation is that it can not consider all of the parameters that will be part of the actual system. The simulation may indicate that a particular algorithm has promise but this will not guarantee its performance in the real system.

Another way to test the performance of a dynamic load balancing algorithm is to build it into the kernel of the operating system. The primary advantage of this method is that it provides the most realistic results. The primary disadvantage is that it is a very time consuming task.

A third way to test the performance of a dynamic load balancing algorithm is to do an emulation of the algorithm. An emulation of an algorithm is an implementation of the algorithm on top of the operating system. The pri-

mary advantages of this method is that we get results in a realistic environment, including the overhead of the algorithm itself, and it is less time consuming to implement.

The remainder of this chapter is outlined as follows. Section 1 presents a description of the bidding algorithm. In Section 2 we present the drafting algorithm along with some simulation results of the drafting algorithm. In Section 3 we present results of our emulations of the bidding and drafting algorithms.

4.1 Bidding Algorithm

The bidding algorithm [Farb73] is a well known algorithm for dynamic load balancing. The basic concept of the bidding algorithm can be defined as follows. When a new job arrives, the scheduling agent will send a request for bids message to all other processors in the system. Each processor that receives a request for bid message will respond with a bid. The bid represents the quality of service that the processor thinks it could provide to the new job. In a homogeneous system the bid may represent the current load of the processor. In a heterogeneous system the bid may also be influenced by the special characteristics of the processor and the special requirements of the process. When the processor that initiated the bid request received all the bids, it will compare the bids with its own bid and award the job to the best bid.

Some variations on the general bidding algorithm are possible. For example a processor may not request bids unless it finds that it is overloaded. This introduces the problem of determining when a processor is overloaded. A good measure of load will reduce unnecessary bid requests while a bad measure may result in too few or too many migrations. Figure 4.1 shows an pictorial description of the bidding algorithm.

4.1.1 A Formal Description of the Bidding Algorithm

In this section we present a programming language description of the bidding algorithm. The algorithm is presented in a Pascal-like language supplemented by the IPC primitives that we introduced in Chapter 2. Listed below is a description of four processes: the request_bids process, the send_bids process, the receive_bids process and the accept_new_jobs process.

```
process request_bids;
var
   pcb: process_control_block;
   bid_waiting_list: list_of_pcb_type;
begin
   loop
      get_new_job(pcb);
      if idle then
         load_local_execution_queue(pcb);
      else
         multicast(schd_agent_group,my_id,"bid request",pcb.id);
         { the bid_waiting list is shared with the
                  receive_bids process}
         add_pcb_to_list(pcb,bid_waiting_list);
      end if
   forever;
end_process;


process send_bids;
var
   bid: bid_type;
   source, id: process_id_type;
begin
   loop
      receive_any(schd_agent_group,source,"bid request",id);
      calc_bid(bid);
      send(my_id,source,"bid reply",bid,id);
   forever;
end process;

process receive_bids;
var
   pcb: process_control_block;
   bid : bid_type;
   source, id: process_id_type;
   bid_waiting_list: list_of_pcb_type;
begin
   loop
      receive_any(schd_agent_group,source,"bid reply",bid,id);
      update_pcb_list(bid,id, bid_waiting_list);
      { when all the bids for a particular process have
```

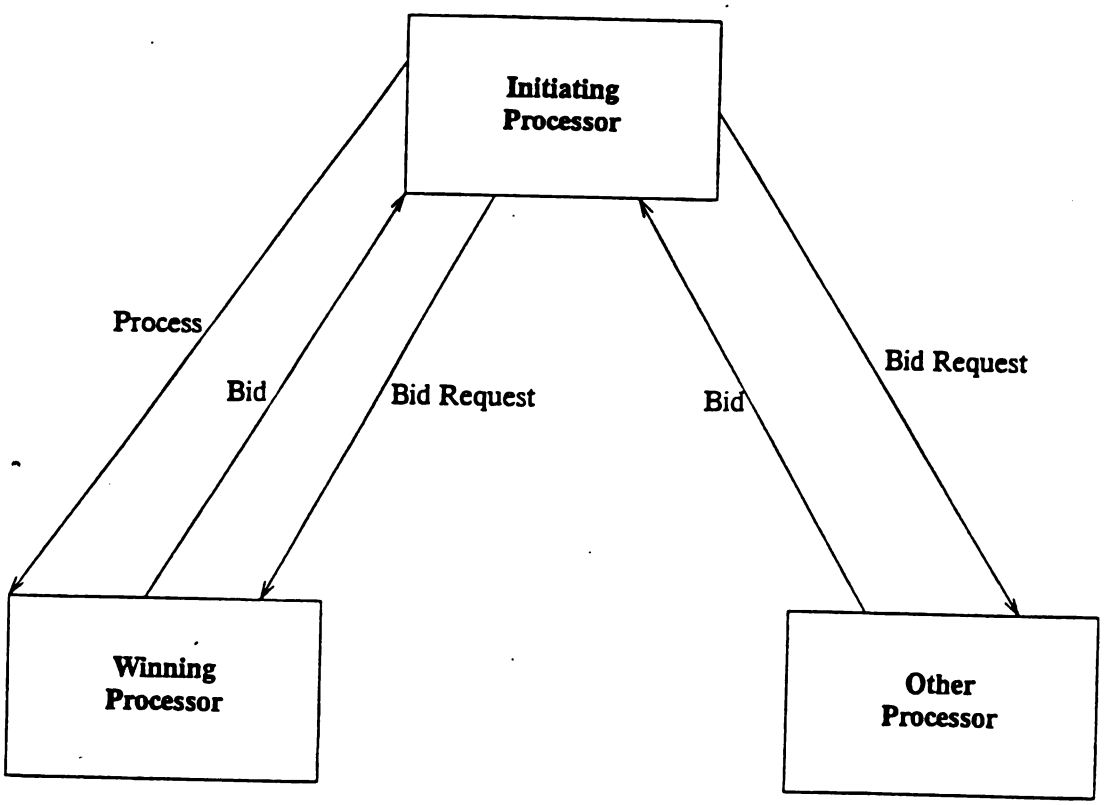

Figure 4.1. Bidding algorithm

```
            arrived the decision about where to schedule the
            process is made }
        if all_bids_received(id,bid_waiting_list) then
            remove_pcb(pcb,id,bid_waiting_list);
            if pcb.best_bid > local_bid then
                send(my_id, pcb.best_bid_agent,"new job", pcb);
            else
                load_local_execution_queue(pcb);
            end_if
        end_if
    forever;
end_process;

process accept_new_job;
var
    pcb: process_control_block;
    source: process_id_type;
begin
  loop
      receive_any(schd_agent_group,source,"new job",pcb);
      pcb.remote := true;
      pcb.source := source;
  forever;
end_process;
```

4.2 Drafting Algorithm

The drafting algorithm is a relatively new algorithm introduced in [NiXG85]. The bidding algorithm is an example of a sender initiated algorithm. It is so called because the processor that initiates the algorithm wants to send a process to a remote processor. The drafting algorithm is a type of receiver initiated algorithm. In this case the processor that would like to receive some processes from other processors initiates the algorithm. There has recently been interest in comparing the performance of sender initiated algorithms with receiver initiated algorithms.

In the drafting algorithm each processor maintains its current processor load. Instead of maintaining a numerical value for the load of each processor, three load states are defined to represent the processor load. A processor in the light load state indicates that it can accept some migrant processes. A processor in the heavy load state indicates that some of its processes can be migrated. A

normal load state indicates that no migration action should be taken by that processor. Several load evaluation methods may be used. For example we could count the number of waiting processes, the average length of time the waiting processes have been waiting, the amount of main memory available, or a combinations of these or other characteristics of the processor.

Each processor maintains a load table. There will be one entry in the load table for each candidate processor. A candidate processor is a processor from which a process may be migrated. Due to communication delay the load table will not always accurately reflect the load of the candidate processors. An important design issue is when announce load changes. Announcing the changes too frequently may result in too much communication traffic while not announcing them frequently enough will reduce the effectiveness of the algorithm.

The basic work of the drafting algorithm is handled by some concurrent processes that make up the scheduling agent. These processes implement the drafting algorithm in the following manner. The *send_draft_request* process will be activated whenever the processor becomes lightly loaded. When this process is activated, it will examine the load table. If all candidate processors are lightly or normally loaded, then no process migration is necessary and send_draft_request process will go to sleep. As long as the processor remains in the lightly loaded state, the send_draft_request process will be awakened if the processor is notified that a candidate processor has entered the heavy load state. When the send_draft_request process finds a candidate processor that it thinks is heavily loaded, it will send a *draft_request* message to that processor. This message indicates that the drafting processor is willing to accept migrant processes. Each processor has a *respond_draft_request* process. This process will listen for draft_request messages. When the respond_draft_request process

receives a draft_request message it will respond with a draft_age message. The draft_age message will contain the draft_age of the processor. This is a measure of how urgently the processor wants to migrate a process. It can be seen as a more detailed figure on the current load of the processor. For example the draft_age could be the number of waiting processes or the time that the oldest waiting process has been waiting. The draft_age will be used by the lightly loaded processor to determine from which candidate processor a process should be migrated. If a processor receives a draft_request message when it is no longer heavily loaded, it will return a draft_age that reflects this fact.

After the drafting processor has received draft_age messages from all the candidate processors that it thought where heavily loaded, it will determine a *draft_standard.* The value of the draft_standard will be based on the values of the draft ages that have been received. Note that if all the draft ages indicate that all candidate processors are no longer heavily loaded, then the send_draft_request process will go to sleep. After the draft standard is calculated the drafting processor will send a draft_select message to the processor that returned the highest draft age. The draft_select message will contain the draft_standard. The simplest draft_standard could just indicate that if the winning candidate processor is still heavily loaded then it should migrate a process. More complex draft standards could be based on the age of the waiting processes or on the number of waiting processes (of course if the draft standard is based on values such as these the draft age should also be based on these values). For example, if the draft age is the waiting time of the oldest waiting process, the drafting processor could choose a draft_standard to be the second highest waiting time.

Each processor has a respond_draft_standard process. This process waits to receive a draft_select message. When a draft_select message arrives it checks

the draft_standard to see if it can still satisfy the draft_standard. If it can satisfy the draft_standard, it will send a process to the drafting processor. If it cannot satisfy the draft standard, it will send a *too_late_message* to the drafting processes. A pictorial description is shown in Figure 4.2. Some simulation results presented in [NiXG85] indicate that the drafting algorithm can provide significant improvement in system response time.

4.2.1 A Formal Description of the Drafting Algorithm

In this section we present a programming language description of the drafting algorithm. The algorithm is presented in a Pascal-like language supplemented by the IPC primitives that we introduced in Chapter 2 and by event variable. We define three basic processes that make of the scheduling agent for the drafting algorithm. These processes are located at the same processor and can communicate through event variables. An event variable is a variable that can be used to notify a process of a state change or be interrogated by a process to check if a state is in effect. Each event variable, *EV*, can take one of two values, *current* or *notcurrent*. An event variable can be manipulated by the following procedures.

```
wait(ev)
signal(ev)
clear(ev)
current(ev)
```

The *wait* procedure causes the issuing process to be blocked until the event becomes current. The *signal* procedure cause the value of the event variable to become current. The *clear* procedure causes the value of the event variable to become notcurrent. The current procedure returns a value of true if the value of the event variable is current and it returns a value of false if the value of the event variable is notcurrent. Changing the value of an event variable is an atomic operation. Listed below is a description of three processes:

respond_draft_request, respond_draft_standard, send_draft_request.

```
process respond_draft_request;
var
      source: process_id_type;
      age: draft_standard_type;
begin
   loop
      receive_any(schd_agent_group,source,"draft_request");
      calc_draft_age(age);
      send(my_id,source,"draft_age",age);
      {the source of the draft request is the destination
      of the draft_age message }
   forever
end_process;


process respond_draft_standard;
var
      source: process_id_type;
      pcb: process_control_block;
      migreply: migration_reply_record;
      standard: draft_standard_type;
      found: boolean;
begin
   loop
      receive_any(sch_agent_group, source, standard);
      select_process(pcb,standard,found);
      if found then { this means a process that
                  satisfied the standard was found}
         migreply.kind := process;
         migreply.pcb := pcb;
      else
         migreply := too_late;
         migreply := get_current_load();
      end_if
      send(my_id,source,"migration reply", migreply);
   forever
end_process;


process send_draft_request;
var
      numheavy: integer;
      processid: process_id_type;
      standard: draft_standard_type;
      proc_list: list_of_process_ids;
      agelist: list_of_draft_ages;
      migreply: migration_reply_record;
begin
   loop
```

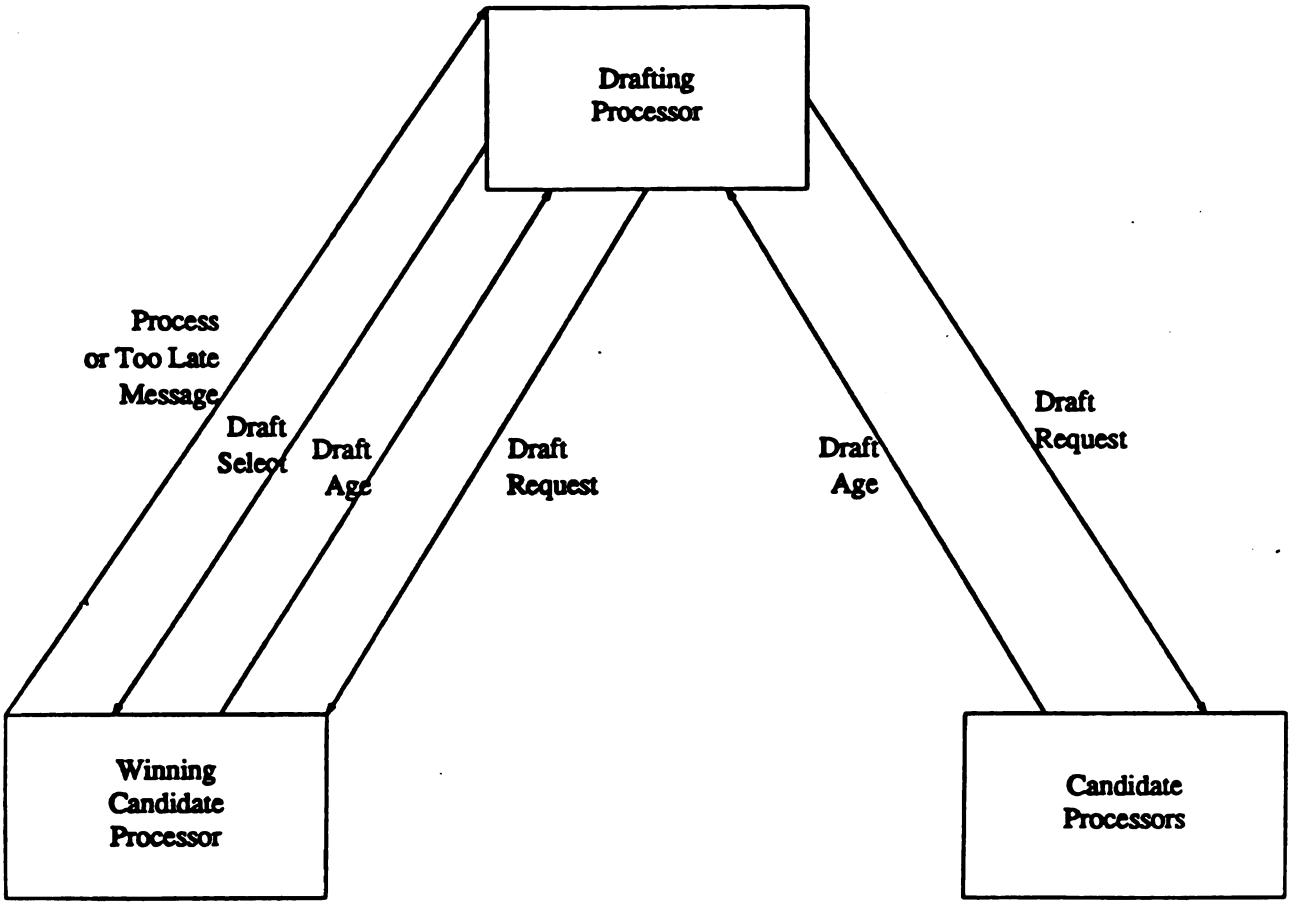

Figure 4.2. Drafting algorithm

```
        wait(light_load);
        {check if there are any heavy loaded candidates}
        any_heavy(numheavy, proc_list);
        if numheavy = 0 then
           wait(normal_load or heavy_message_arrive);
           clear(heavy_message_arrive);
           if current(normal_load) then goto L1
           else goto L2;
        end_if
        if numheavy > 1 then
           multicast_and_wait(proc_list,"draft_request",
              agelist, status);
           calc_draft_standard(proclist,agelist,
              standard, processid, foundheavy);
           if not foundheavy then
              wait(normal_load or heavy_message_arrive);
              clear(heavy_message_arrive);
              if current(normal_load) then goto L1
              else goto L2;
           end_if
        else
           standard := 0;
           processid := first(proc_list);
        end_if
        if not current(light_load) then goto L1;
        send_and_wait(my_id, processid, "draft_select",
           standard, migreply);
        if migreply.kind = process then
           load_mig_queue(migreply.pcb)
        else
           update_load_table(processid,migreply.load);
        end_if
     L2: until (not current(light_load))
     L1: forever;
  end_process;
```

4.3 Emulation of Bidding and Drafting

A version of both the bidding algorithm and the drafting algorithm have been implemented on the Sun workstations in the Distributed Computing Research Lab (DRCLab). The physical configuration of our network consists of five Sun-2 workstations connected by a 10 MBPS Ethernet. The Sun workstations are running the Unix 4.2 operating system. The Unix 4.2 operating system does not provide the type of communication primitives that were presented in Chapter 2. In Section 4.3.1 we will discuss the communication primitives that are provided by Unix. 4.2. In Section 4.3.2 we will discuss the emulation

programs and the results that we have achieved.

4.3.1 Unix 4.2 IPC Primitives

The Unix 4.2 operating system provided two families of transport level protocols that can be used for interprocess communication. The protocols are Transmission Control Protocol (TCP) and Unnumbered Datagram Protocol (UDP). TCP provide connection oriented (virtual circuit) one-to-one batch communication. The communication provided through TCP is highly reliable. UDP provides unreliable one-to-one expedite communication. This can be used for both one-to-one and broadcast based communication.

The interface to the communication system is through an object called a *socket*. Unlike the IPC system defined in Chapter 2, in Unix 4.2 all messages are sent to or received from sockets. A process can send or receive messages by creating sockets. In order for two processes to communicate using TCP one process must publish a socket id (or name) on which it will be willing to accept connections. The other process will then try to create a connection by issuing a *connect* command. Thus the communication is not symmetric during the connection phase. Once a connection is established the two processes can communicate using a blocking receive command, *recv*, and an asynchronous one-to-one send command called *send*. Listed below are two generic examples of the send and recv command. Each command uses the socket, *sk*, that is the endpoint for its communication, a point to a buffer (array), the buffer_length that represents the number of bytes that is being sent or received.

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

If the UDP protocol is used then no connection phase is required. If process knows the name of a remote socket, it can use the *sendto* command to send information to the socket. If a process is willing to receive information

from a UDP socket it issues a *recvfrom* command. UDP can also be used to do a limited form of broadcasting. Using a special address format it can broadcast a message to a socket on each machine in the network. The destination of the message will indicate that it is a broadcast message and will provide a port name to deliver the message to. Any socket that has the correct local port name will receive the message. Since at most one socket can use the same port name per machine, this protocol only allows a process to broadcast to one other process on each processor. Unfortunately we found that this broadcast capability was not very reliable and thus we did not use it in our emulation.

4.3.2 The Emulation Design

Figure 4.3 shows a high-level view of the emulation design. Each scheduling agent consists of three processes: the job creation process, the job execution process and the scheduling controller process. Since Unix 4.2 does not provide the ability of processors to share memory, these processes must exchange information in another ways. In our current structure the job creation process and the job execution process are children of the scheduling controller process.

The job creation process creates processes with a Poisson arrival pattern. Each newly created process is assigned a process id and has its starting time attached to it. Each time a new process is created the job creation processes sends its job id and its arrival time to the scheduling controller process through a pipe. A *pipe* is a simple one way communication channel.

The scheduling controller process is responsible for making the global scheduling decision in cooperation with the other scheduling controller processes. The first thing that the scheduling controller process does is to create TCP connections with all the other scheduling controllers. This connections are maintained throughout the life time of the emulation. After the connections

are established the scheduling controller creates the job execution process and the job creation process. After these processes have been created, the scheduling controller starts executing the load balancing algorithm.

The job execution process does the actual execution of the jobs. It receives new jobs to execute from the scheduling controller. It executes jobs in a first come first serve manner. When it completes a job it returns the job to the scheduling controller and attaches its completion time. If the job was a remote arrival, the scheduling controller will ignore the completion time and send the job back to its original processor. If the job was a local process, the scheduling agent will record its completion time. The actual jobs that are executed are functions that are called by the job execution process. In the emulations that we have run thus far these jobs only use CPU resources.

The application processes that are executed are artificially generated but they do use CPU time. During the emulation, the CPU on each station is shared by the following processes: artificially generated application processes, the job creation process, the job execution process, the scheduling controller process, and some Unix operating system processes that could not be suspended. In all of our tests we only used the four client stations. The file server was not used for application processes because it serves many different functions and cannot be accurately used to compare performance with the client stations. As a simple example, when a client writes some results of the emulation to a file, the file server must do some work. For each client writing the results creates the same amount of overhead. However, if the file server were running application jobs, these jobs would entail the overhead of all the clients file activity. Thus the application processes on the file server would not be running under the same conditions as those on the client stations.

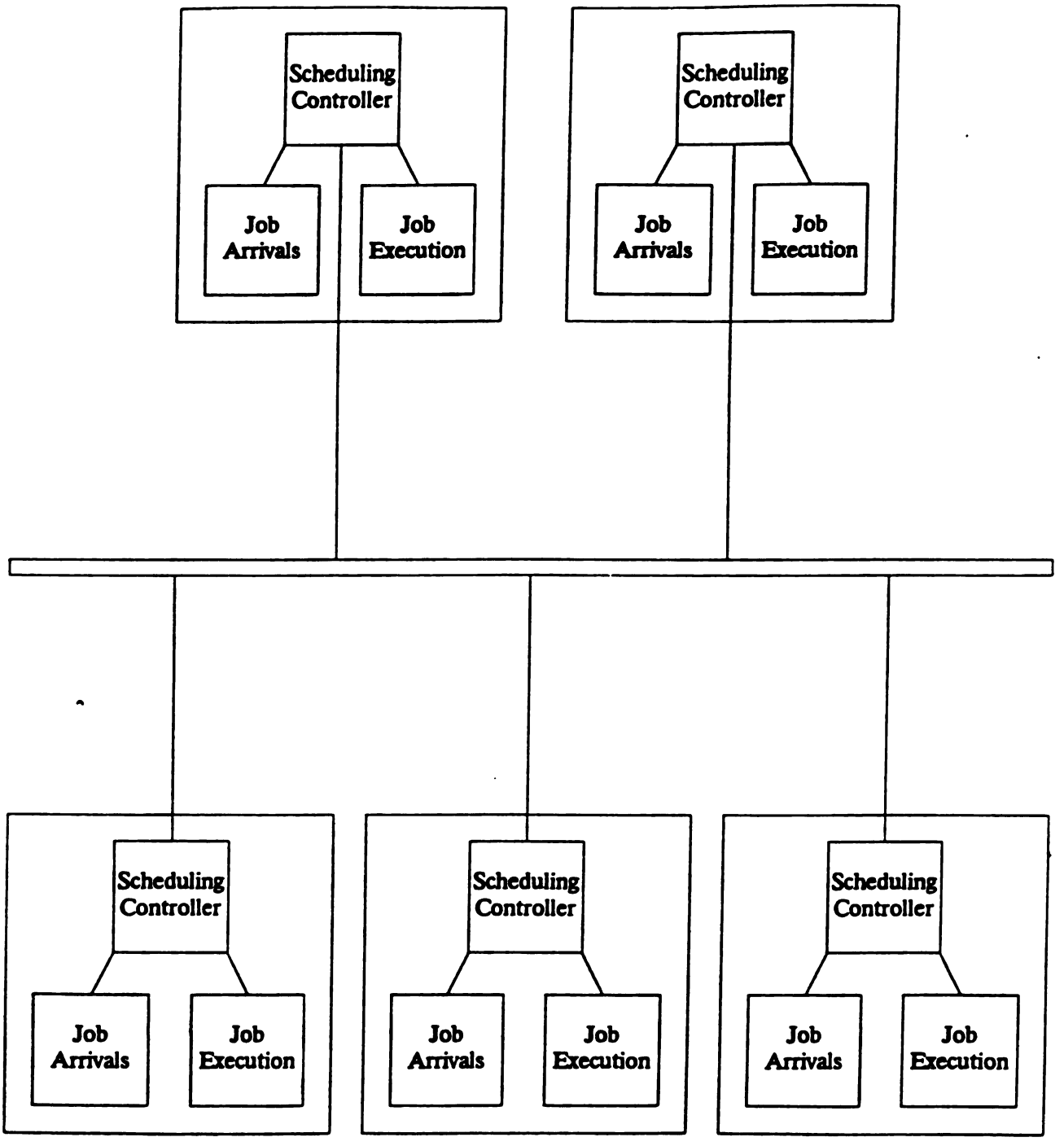

Figure 4.3. Emulation structure

As we indicated in our discussion of the bidding and drafting algorithms, there are a number of parameters that must be defined when the algorithms are implemented. These parameters include the following: the value used for bids, when to request bids, the value used for draft ages, the choice of a draft standard, and a measurement of heavy load and light load for the drafting algorithm. In the experiments discussed below we used the following parameters. The value used for bids and draft ages was the number of waiting processes. A station requested bids for a new job if there was at least one job waiting when the new job arrived. A simplified draft standard was defined. As long as the winning station was still heavily loaded when the draft select message arrives, it could migrate a job. Note this is not the fairest draft standard to use since it does not attempt to create a first come first serve ordering across stations. However, this will be the best choice in terms of minimizing response time and since the goal of our experiments was to compare bidding and drafting in terms of response time we chose this standard. In the drafting algorithm any station with four or more waiting processes was considered heavily loaded and any stations with zero or one waiting processes was considered lightly loaded.

Figure 4.4 presents a graph of the results of our first experiment. The arrival pattern was such that 2/5ths of the arrivals arrived at client 1, another 2/5ths arrived at client 2 , 1/5th of the arrivals arrived at client 3, and client 4 had no external arrivals. The jobs that were executed were all relatively long (the mean value of the execution time was 12 seconds and the execution time of the shortest job was 4 seconds) thus making them all migratable.

Figure 4.5 presents the results of second experiment. In this experiment each station had the same arrival rate making the stations probabilistically balanced. The job mix was the same as that used in experiment one.

In both of our experiments we executed approximately 4000 application jobs in the system. Because of the long execution time each point on the graphs took about 6 hours to generate.

As can be seen from both graphs both load balancing algorithms provide improvement over the no load balancing case. Under our current conditions the bidding algorithm outperforms the drafting algorithm. The primary reasons for this are 1) the drafting algorithm reacts too slowly to load changes, 2) maintaining global information creates too much communication overhead and does not provide significant benefits in a small network, and 3) the drafting algorithm is computationally more expensive and thus takes away cpu time from the application jobs. Note that these results are valid for networks of a few workstations (less than 10). In the bidding algorithm that we used, the station with the winning bid is immediately sent a job. In a small network this is acceptable because a station cannot win too many bids at the same time. In a large network it is possible that a station could win many bids within a short time and thus become overloaded. In this case we might migrate a job from one overloaded processor to another overloaded processor. In order to prevent this a more complicated bidding algorithm must be used. When a processor wins a bid, it is notified of this fact and it is asked if it is still willing to accept the job. This allows the processor to control the number of migrant jobs that it accepts but it also requires two extra communications. This version of the bidding algorithm will create more overhead and will react more slowly to load changes.

As a final comment, the drafting algorithm as it is now defined migrates only one job at a time. An interesting modification to this algorithm would be to allow it to migrate 2 (or more) jobs at a time. When a process receives a draft select message, it could be allowed to migrate 2 (or more) jobs that satisfy

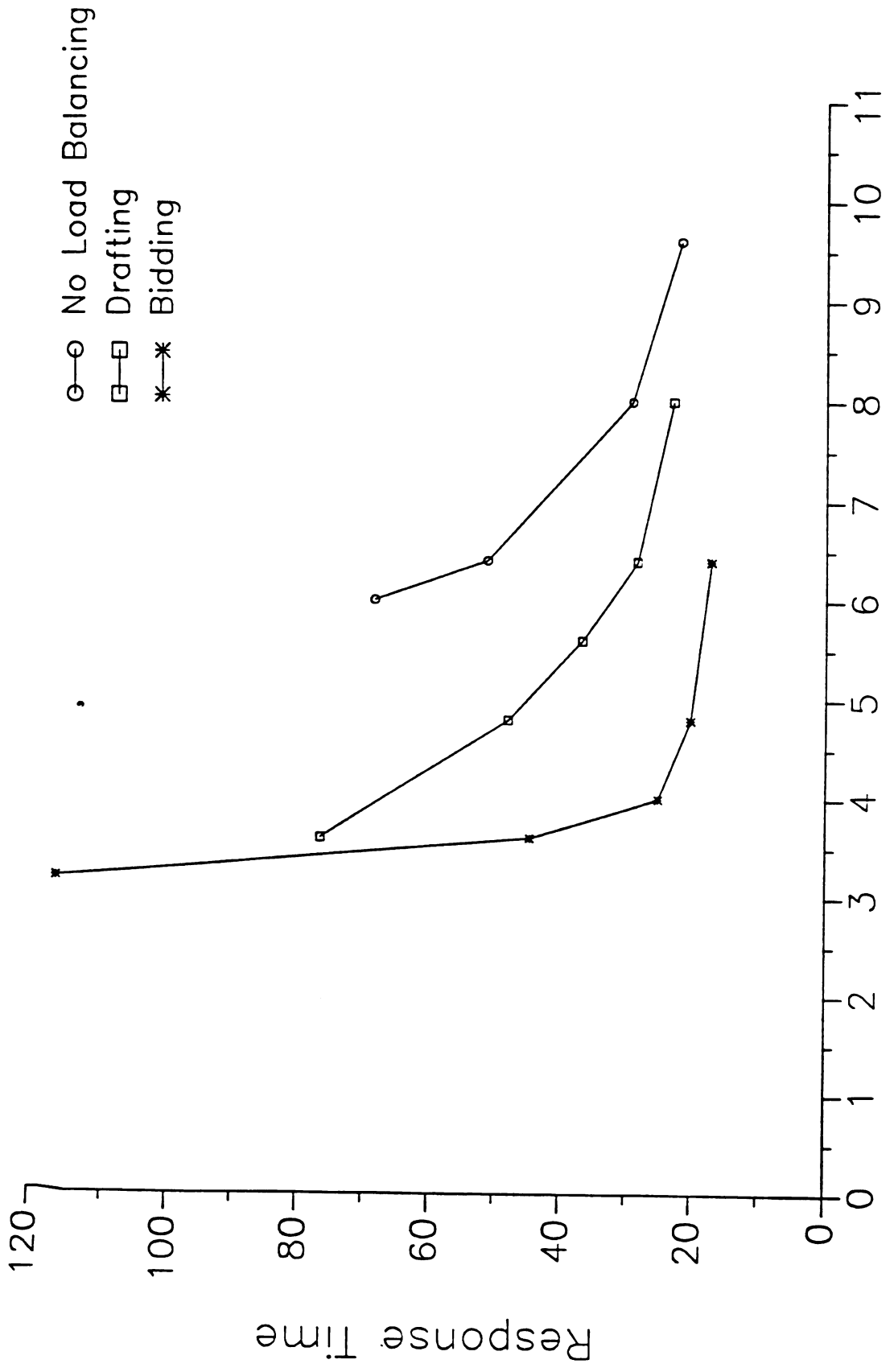

Figure 4.4. Comparison of bidding and drafting

the draft standard. This would allow the drafting algorithm to keep very lightly loaded processors from becoming idle too frequently.

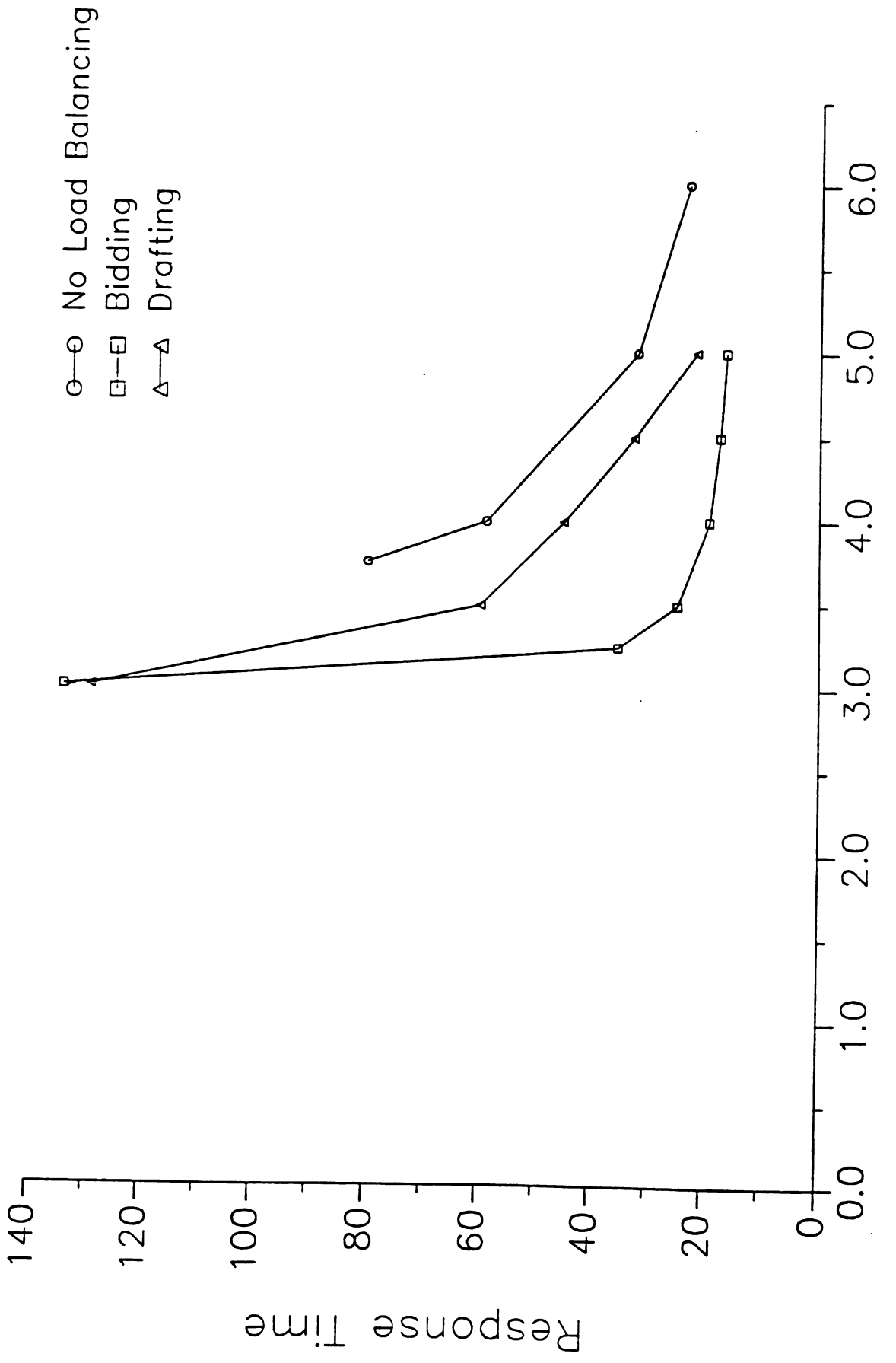

Figure 4.5. Bidding and drafting with static load balancing

CHAPTER 5
DISTRIBUTED ALGORITHMS

One use of a distributed system is to develop distributed algorithms. Most of the current work emphasizes the creation of distributed programs which control the distributed architecture through the creation of distributed operating systems [WiVa80, PoMi83] and distributed database systems [Walk83]. More recent work has addressed the issue of taking advantage of the potential parallelism provided by the distributed architecture [FiMa85]. The focus of this chapter is the development of general concepts of distributed algorithms. The ultimate goal of this research is to sufficiently develop some design issues specific to distributed algorithms so that greater use of distributed systems can be made.

5.1 Distributed Algorithm Definitions

We define a *distributed algorithm* to be a set of processes, running on various processors, that communicate through message passing and cooperate in order to solve a problem. Distributed algorithms can be classified into two categories, *naturally distributed algorithms* and *artificially distributed algorithms* [GeNi85]. A distributed algorithm is considered naturally distributed if the cooperating processes are located on different processors because the data needed by the processes are located or generated at different locations. In other words, the processes are distributed because the data are distributed. Most distributed control algorithms can be classified as naturally distributed. For example, a load balancing algorithm [NiXG85, StSi84] which monitors the load at each processor and can migrate processes if improvement in system performance is expected. Such an algorithm has processes, located at each processor,

which monitor the load at each processor, report changes in the load to other members of the algorithm and agree to migrate processes when it appears beneficial. In this example, the distributed data is the current load at each processor. Note that processes of a naturally distributed algorithm are not migratable.

Any algorithm which exhibits some parallelism could be expressed as an artificially distributed algorithm [FiMa85]. In an artificially distributed algorithm, it is not necessary to access physically distributed data. Instead, the algorithm would be distributed in order to take advantage of parallelism or to make a more efficient use of resources. The advantages of distributing such an algorithm will depend on the particular algorithm's communication patterns and on the type of distributed system available.

There are three important features of distributed algorithms: *negotiation*, *remote state maintenance*, and *reliability*. Each of these will be discussed in the following three sections.

5.2 Negotiation in Distributed Algorithms

A negotiation is a set of actions through which two or more processes attempt to reach an agreement. The agreement may be an agreement on a shared value or it may be an agreement to perform some action. The influence of negotiation can be seen in one of the earliest distributed systems, Farber's *Distributed Computing System* [Farb73]. In this system, an algorithm, called the *bidding algorithm*, requires a negotiation to find where a newly arrived job would be scheduled. In [Smit80], Smith recognized the importance of negotiation in his contract net protocol. Some more recent work in this area includes [DLPS85, DwLS84].

A negotiation can take two general forms: *coordinated* and *peer*. In the

case of a coordinated negotiation, one of the negotiators is called a *coordinator* and the other negotiators are called *respondents*. The coordinator controls the negotiation. Usually, information is passed to or from the coordinator. The respondents do not communicate between themselves, unless they are instructed to do so by the coordinator. During the negotiation phase, the coordinator provides a central point of control. In a peer negotiation, all negotiators (processes) are equals. There is not a centralized point of control and no negotiator has a complete view of the negotiation. This is a more distributed form of negotiation and a form which is important in distributed game playing. An example of a peer negotiation is an *election algorithm* [Garc82, LyFr85]. In an election algorithm, a set of peer processes elect a process to perform some special function.

It should be remembered that a single distributed algorithm may have a number of negotiation phases. It is not necessary for each phase to exhibit the same characteristics. For example, for reliability purposes a coordinated negotiation might require the coordinator to periodically communicate with each respondent. If the coordinator fails, the respondents might be required to elect a new coordinator. In this case we see two phases of negotiation, the former is coordinated and the latter is peer.

Negotiators can be classified as either *selective* or *non-selective*. Each negotiation follows a pattern in which one of the negotiators makes an offer and the other negotiators may accept the offer, reject the offer, or make a counter offer. A selective negotiator will agree only if a standard or range of standards is satisfied. A selective negotiator is free to leave a negotiation without agreement. An example of this is the migrant phase of the drafting algorithm [NiXG85]. In this case, a processor is willing to accept a migrant process from a remote processor only if the process can meet the draft standard (a decision

making threshold). A non-selective negotiator must reach agreement in a negotiation. While the negotiator will try to reach the best agreement possible, it must always accept some offer. A example of this is a simple bidding algorithm in which bids are requested for each new job. Since the job must be scheduled somewhere, one of the bids must be accepted (the local scheduler may also bid).

Some important examples of negotiation in distributed computing include *byzantine agreement*, *mutual exclusion*, and *process synchronization*. The byzantine agreement [PeSL80, LaSP82] problem addresses the problem of finding agreement among a set of faulty processes. The faulty behavior of the processes can include malicious attempts to mislead other processes. The malicious processes may want to prevent an agreement or they may want the correct processes to think an agreement has been reached when it has not.

The mutual exclusion problem [RiAg81] in distributed systems has the same goal as the mutual exclusion problem in centralized systems: to guarantee that at most N (where N is usually 1) processes have access to a critical section, such as updating a shared value, at any time. In distributed systems the problem is made more difficult because the critical section may be replicated in many different locations. In a distributed mutual exclusion algorithm it is also more difficult to provide fair access to the shared value or critical section. In order to allow a process to access a shared value, an agreement among all processes that maintain a copy of the shared value must be reached.

The problem of process synchronization [Silb79, HLLR85] can also make use of negotiation. Synchronization issues arise in the areas of initiation of distributed programs, termination of distributed programs [Kuma85, SzSP85], and ordering of process actions. In initiating a distributed algorithm, all processes in the algorithm must have identified themselves as ready to begin. Problems arise in deciding how the processes should inform the others that they are ready

and how to decide that all processes in the algorithm are ready. In distributed termination, the problems include how to determine when the algorithm has completed and how to assure that all processes in the algorithm know that it is completed. Ordering of processes [Lamp78, LaSt81, LiSc83] is concerned with identifying that the actions which must be completed before a process can start actually have been completed.

5.3 Remote State Maintenance in Distributed Algorithms

Another important feature of many distributed algorithms is remote state maintenance. The processes which are cooperating must have some knowledge of each other's state [ChLa85, NiXG85]. In order to do this, the processes will periodically report their current state to some of the other processes in the algorithm. It is important to note that no matter how frequently information is exchanged between the processes, each process will have a slightly different view of the system. The primary problems associated with remote state maintenance are the frequency of remote state reporting and the amount of information that should be reported. A tradeoff exists between the cost of transmitting information and the value of more accurate information. The algorithm designer must identify the essential state information and changes to that information that are significant to the remote processes.

5.4 Reliability of Distributed Algorithms

The final primary feature of distributed algorithms that we will discuss is reliability. The reliability of an algorithm is a measure of how it performs in the presence of faults. For distributed systems there are three primary classifications of faulty behavior: ***fail-stop, omission, and byzantine*** [Stan85]. A processor (each of these classifications will be described in terms of processors but the concepts can also be applied to processes) is considered a fail-stop

processor if it stops when a fault occurs. In other words the processor either behaves correctly or it stops. In general other processors in a distributed system cannot determine if a processor has stopped or is very slow. A processor that can have omission faults may fail to send some required information but when it does send information it is correct. A processor that may experience byzantine faults may fail to send information and may send incorrect information. During a byzantine failure a processor may act in malicious ways in order to confuse the correct processors or to get them to act incorrectly.

Once a fault has occurred there are a number of steps that have to be taken. They include *fault detection/diagnosis, system reconfiguration* , and *system recovery* [Stan85]. Fault detection/diagnosis is concerned with identifying the faulty processors. In a distributed system of fail-stop processors this is difficult because it is not always possible to determine if a processor is slow or it is stopped. In the byzantine case this is even a more difficult problem. One possible approach is to use the concept of authentication as described in [PeSL80]. System reconfiguration is concerned with reorganizing the system after the faulty processor has been identified. This may require the processors to redistribute the responsibilities of the failed processor. System recovery is concerned with those actions that must be taken to repair any damage that was done by the failed processor.

5.5 Conclusions

There are other important issues in distributed algorithms including security, algorithm correctness, algorithm complexity. Security is important whenever sensitive data is being handled. Guaranteeing security becomes even a greater problem when data is transferred over publically accessible communication media. Correctness is an obvious goal of any algorithm design. However,

showing that a distributed program is correct is very difficult. Because of the high degree of parallelism in distributed programs, testing the programs provides less assurance that it is correct than in the case of sequential programs. In order to gain greater assurance that an algorithm is correct, a formal methodology that can be used to prove correctness must be developed. The complexity of a distributed algorithm is also an important design criterion. Some common measurements of complexity include communication costs (number of messages sent), computation costs, and the communication/computation cost ratio. The communication/computation cost ratio can be used as a measure of the appropriateness of implementing a distributed algorithm on a particular architecture. While each of the above features, security, correctness, and complexity, are important in distributed algorithms, we do not emphasis them in this chapter because they do not have a special relationship to the distributed nature of an algorithm as do negotiation, remote state maintenance, and reliability.

We conclude this discussion with some comments on the architectural environment for distributed programming. A distributed architecture consists of a number of processors, where each processor has its own local memory and there is no shared memory. The processors are connected to a communication subsystem that allows any two processors to communicate. The most common example of this type of architecture is a local area network. Two features of a distributed architecture are important. They are the time it takes to send a message from one processor to another and the degree of autonomy that the processors have. Two architectures are appropriate for distributed programming. They are *local area networks* (LAN) and *loosely coupled multiprocessors* [MeBo76]. The boundary between these two architectures is not well defined. In terms of distributed programming, LANs can be characterized as having a

longer (relatively) transmission time and greater processor autonomy; whereas the loosely coupled multiprocessors has a shorter transmission time and less processor autonomy. Naturally distributed algorithms will be developed for both types of systems (for example, the control algorithms of a distributed operating system). The primary difference will be how conducive the system is to the development of artificially distributed algorithms. In order to efficiently execute artificially distributed algorithms, the message passing time should be very short and the system should have strong control over where processes are scheduled.

CHAPTER 6

THE DISTRIBUTED GAME PLAYING ENVIRONMENT

The availability of distributed systems has created interest in the development of distributed algorithms. In order to investigate distributed algorithms an environment for investigating distributed algorithms, called *distributed game playing* (DGP), has been developed. The DGP environment can be used to investigate the basic features of distributed algorithms that were defined in Chapter 5. The DGP environment allows researchers the flexibility to highlight various features of distributed algorithms while creating managable size problems. In this chapter the basic features of the DGP environment are defined and the relationship between the DGP environment and problems in distributed computing is shown.

6.1 Generic Player Structure

A *distributed game playing* algorithm is a three tuple (P,L,A), where P is a set of players, L is a language represented by a set of message forms that the players understand, and A is a set of actions that players may take based on the state of the game. Each DGP algorithm has five phases: *tournament formation, round formation, game playing, round termination*, and *tournament termination*. The three middle phases may be repeated many times. Figure 6.1 presents the overall structure of a game player .

```
process player;
type definitions;
message form definitions;
variable definitions;

begin
      tournament formation phase;
      repeat
```

```
            round formation phase;
            game playing phase;
            round termination phase;
        until tournament is over;
        tournament termination phase;
    end player;
```

Figure 6.1. Generic Player Structure

The tournament formation phase involves a set of players trying to form a tournament. The number of players in the set is not known until the tournament is formed. In a *close* tournament the new players cannot join the tournament when the tournament formation phase is completed. In an *open* tournament players may join even after the tournament formation phase is completed. Usually the new players will be able to join the open tournament only between rounds. In some cases, when the tournament has been formed, all players in the tournament will know the name of all the other players in the tournament.

Each *tournament* consists of a number of *rounds*. In each round, players are bound to a *game*. The binding is accomplished through the cooperation and agreement of the players in each game. There will be many games per round. For example, a tournament with $2n$ players that requires exactly two players per game would have n games per round. All games of one round are played concurrently. Each player will be bound to at most one game per round. The round formation phase involves players agreeing to join with other players to form a game. When all players have joined a game or the maximum number of games has been formed, the round formation phase is completed.

In the game playing phase, the players of each game must decide on an order of play. In some games, players will want to elect a coordinator. During the playing of the game, the players (or the coordinator) will be responsible for maintaining the shared game state. The game state provides information that all players can use to decide how to make their next play. If the players do not

trust each other (i.e. unreliable players), they can monitor each play in order to assure themselves that the play is legal. The players (or the coordinator) will be responsible for determining when the game is over. When the game is completed, the players (or the coordinator) will have to announce that the game is over and report any scoring information that is pertinent to the tournament. After this, the players enter the round termination phase.

Players enter the round termination phase at different times depending on when their current game ends. In some DGP programs, the round termination phase requires that a player waits until all games of the previous round are completed before the player returns to another round formation phase. In this case a player in the round termination phase listens for announcements of completed games. When the player finds that all games are completed, the player completes any remaining round termination actions and enters another round formation phase (if the tournament is not over). In other DGP programs, a player may enter a new round whenever there is enough players available to form a new game. In this case a player in the round termination phase will listen for games to be completed. When the player decides that enough games have completed, the player may enter the round formation phase and try to form a new game. In some cases, the player may be required to keep track of the number of games in which the player has participated.

The tournament termination phase involves the confirmation by all players that the tournament is completed. Announcement of pertinent information concerning the results of the tournament may also take place. For example, the name of the winning player and the scores of other top players could be announced.

The active entities in a tournament are the players (and possibly game managers). The players are not necessarily associated with human players.

From the environment's point of view, this is not an issue. Each player will be represented by a process. It is the interaction of the processes which represent players that is of interest to us. The internal structure of players in the same tournament does not have to be the same. However, they must speak the same language. In other words, when they communicate with other players, they must follow the same protocols. There is not (necessarily) a coordinator at the beginning of the tournament, although one can be elected after the tournament is formed.

6.2 Characteristics of the DGP Environment

Some common features of distributed programs were presented in Chapter 5. These features can be easily investigated in the DGP environment. Negotiation plays an important role in DGP programs. The only active entities in the program are the players. Players must negotiate in order to form tournaments and rounds. They must also negotiate to determine the order of play in a game. The flexibility of the negotiation will depend on how the players are structured. The agreement might require only the simple exchange of information or it might require an extended negotiation. For example, a player might arrive when a number of *tournaments* are being formed. This player could negotiate with players in each tournament before deciding to join one. The complexity of the negotiation will depend on what attributes are associated with the player.

Synchronization issues are important in the initiation and termination of rounds and tournaments. In the case of the tournament formation phase, players in the tournament must agree when the tournament formation phase has been completed so that *new players* are not accepted after the *tournament* is formed. When agreement on this point is reached, the tournament can begin with the initial round formation. In the case of round termination, new round

formation can not begin until all games of the previous round are completed (in some cases) and all players agree that the games are completed. Synchronization issues are also important in ordering of the players turns within one game and in updating of the game state. Different strategies can be developed to reflect different levels of concurrency.

Remote state maintenance is needed whenever players need information about the actions of other players, in which they were not directly involved. For example, a copy of the shared game state will usually be kept by each player. When one player takes its turn, it will change the shared game state. The changes must be reported to all players in the game. The amount of information included in the shared game state will affect how well players can make decisions during their turn. A variety of player designs can be used to see how changes in the amount of information stored in the shared game state affect player performance. Remote state information will also be used to keep track of information between rounds. This information could include scores associated with each player and the identity of the winners of previous games.

Reliability problems are an issue when the players may be unreliable. In this case players may monitor other players to make sure they update the shared game state in a legal manner. Individual players could be represented by a number of processes to take advantage of redundancy. In this case a players actions could be decided by a vote among the processes that represent the players.

The DGP environment can be used to investigate and illustrate problems in distributed computing. The multiphase structure of a DGP algorithm corresponds to the multiphase structure of many distributed algorithms [Kuma85, ChLa85]. The tournament formation phase can be used to investigate problems in load balancing and distributed initiation. The round formation

phase can be used to investigate problems in system reconfiguration and cooperative scheduling. The game playing phase can illustrate problems in mutual exclusion (shared game state), distributed experts (teams of players), and reliability (monitoring other players actions). The round termination phase can be used to investigate distributed termination problems. In Section 6.1 we will survey some problems in distributed computing and show how they can be expressed in terms of DGP.

6.3 Classification of DGP Features

In Figure 6.2, we present some classifications that can be associated with the DGP environment. In the tournament column, we list some classifications for tournaments. A *fixed length* tournament will end after a fixed number of rounds. The actual number of rounds might depend on the round formation structure and the number of players in the tournament. In a *fixed time* tournament, a new round will not start after a certain time. The time value will usually be agreed upon by the members of the tournament during tournament formation. In an *unbounded* time tournament, the length of the tournament is not known after the tournament formation phase is completed. At some point during the tournament, all (or some other agreed upon number) players will agree to end the tournament. A *fixed size* tournament requires a fixed number of players in order to start. The size is usually expressed as a multiple of the number of players required for one game. For example, a tournament in which the games required 3 players could require that the tournament has $3n$ players. In this case the tournament formation phase might be required to remove some players in order to get a multiple of three players. The *fixed size* attributed could be combined with other attributes. For example, we could have a *fixed size, fixed length* tournament.

Some DGP Classifications			
Tournament	Round	Game	Player
Fixed Time	Round Robin	Fixed Size	Stationary
Fixed Length	Tournament Ladder	Bounded Size	Mobile
Unbounded Time	N-and-Out		Stop-Fail
Fixed Size	Open Negotiation		Byzantine

Figure 6.2. Some DGP Classifications

In the round column, we list some possible round structures. In a *round robin* structure, all possible combinations of games must be played. For example, if each game requires two players, then every player will try to play with every other player. In a *tournament* ladder structure, only the winners of the games in a round advance to the next round. The tournament ends when there is only one player remaining. In an *N-and-out* structure, a player remains in the tournament until the player losses N games. After each loss, a player moves into a different bracket in which all players have lost the same number of games. At some point in the tournament, there will be N players remaining. Where one player has lost zero games, one player has lost one game, etc. These players will play each other until only one player remains (i.e., all the other players have lost N games). In an open negotiation structure, there are less guidelines about how a new round can be formed. New rounds will be formed through a more flexible negotiation between the players. One possibility is that when a round ends, all players become "free agents" who can negotiate with any other players to form a new game. Another possibility is that when a game ends, the players in that game choose a player to trade (a kind of election). When the player has been chosen, the group of players negotiate with other groups of players to complete a trade.

In the game column, we list some attributes associated with games. A *fixed* size game is one that requires an exact number of players. For example, a game

could require exactly 4 players. In a bounded size game, there is a maximum and minimum number of players required for the game. For example, a game could require between 3 and 5 players.

In the player column, we list some attributes associated with players. A *stationary* player is a player who is bound to one processor. This would be the usual case when a player process is associated with a human player. A *mobile* player can be scheduled at any processor. Such a player will be moved from one processor to another in order to reduce communication costs. For example, all the players in one game can be located at the same processor if the are all mobile. Fail-stop and byzantine [Lamp83] are attributes that can be associated with players if an algorithm designer wants to investigate reliability issues within the DGP environment. A fail-stop player is a player that acts correctly or does not act at all. A byzantine player may perform illegal and malicious actions.

The classifications listed in Figure 6.2 represent only some possible classifications. They are presented as a starting point for problems that can be used to illustrate concepts in distributed computing and to practice distributed computing. Problems that highlight different issues can be created by associating different attributes with tournaments, rounds, games, and players.

6.4. Using DGP to Illustrate Distributed Computing Concepts

The DGP environment can be used to illustrate, teach and investigate distributed computing concepts. In this section we review some important issues in distributed computing and show how these issues can be illustrated using the DGP environment. The concepts that we will discuss include *bidding, mutual exclusion, voting, distributed termination,* and *byzantine agreement.*

6.4.1 Bidding in the DGP Environment

As we mentioned earlier, the concept of bidding was one of the earliest concepts in distributed computing. This concept has been used as a means of load balancing in distributed systems [Farb73, StSi84] and as a control mechanism for distributed problem solving [Smit80].

Bidding is a rather simple concept that is very powerful because of its generality. For example, in the case of load sharing, the process with excess work could be a scheduling process that finds its processor overloaded. As new work arrives this scheduling process may try to send that new work (processes) to other processors by asking the remote scheduling processes if they will accept new work. In a distributed problem solving example we could have a set of cooperating distributed experts. These experts may have different areas of expertise. One expert may find it needs a solution to a problem for which it is not an expert. It may announce its need to solve a particular type of problem and wait for bids from experts that know how to solve the problem.

For each specific application of the bidding concept, a number of design choices must be made. These design choices include identifying processes that should be sent request-for-bid messages and determining how long a process will wait for bids. The destination of request-for-bid messages can be determined statically or dynamically. In a static method all of the cooperating processes could be sent each bid request. In a dynamic method the nature of the available work that is being offered and the most recent state of the remote processes could determine which processes will be sent request-for-bid messages. For example, in a load sharing algorithm we could send request-for-bid messages in three different ways: 1) send request-for-bid messages to all remote scheduling processes, 2) send request-for-bid messages to the remote scheduling processes that control lightly loaded processors, or 3) examine the characteristics of the work (process) and send request-for-bid messages to the remote scheduling

processes that control processors that can best satisfy these characteristics. In determining the length of time that a process should wait for responses from processes that were sent request-for-bid messages, we can look at three options. One option is to have the process wait for a bid from each process that was sent an announcement (this requires the process to know the number of processes that have received the request-for-bid messages). Another option is to have the process wait for a specific time period for bids. A third possibility is to have the process wait until it has found a bid that is acceptable. In determining who should be sent request-for-bid messages and how long to wait for bids the algorithm designer must take into consideration the nature of the algorithm and the nature of the underlying communication system.

The concept of bidding can be illustrated using DGP in a variety of ways. For example, we could have a tournament that requires the players in the same game to be located on the same processor. Each processor would have a *game manager* which controls the players during round formation. The game managers would cooperate to create a new round formation. Two goals of the game managers could be to keep the (approximately) same number of players in each game and to make sure that the players play with different players in each round. The game managers could attempt to accomplish this by announcing that certain players were available at the beginning of the round formation phase. Following that other game mangers would send bids indicating their desire to have a particular player. The announcements of available players could include each available player's name and a history of the previous players with which it had played.

In another example we could have a tournament that has teams of players. Each team could have a team manager which decides on the next "move" that should be made. The players on the team would have a variety of skills that

can be used to help the team manager make its decision. The team manager would ask for help from the other players by asking for bids to evaluate the current game state. Other players could also ask players for help when they are evaluating their part of the game state. The team manager will decide on the move to make based on its own evaluation and on the partial evaluations of the other members of the team.

6.4.2 Mutual Exclusion in the DGP Environment

The problem of mutual exclusion in distributed systems has the same general goal as in the case of centralized systems: to guarantee that at most N (where N is usually 1) processes have access to a critical section at any time. The problem is made more difficult in distributed systems because the shared values associated with the critical section may be replicated. For example, a set of processes could share an integer value where each process maintains its own copy of the shared value. If a process wants to update the shared value, it must make sure that no other process is trying to update the value. In some cases processes will be prevented from reading a value if it is currently being updated. Another problem faced by a distributed mutual exclusion algorithm is fairness. In a centralized system requests to enter a critical section can be queued at a central point and processed in a FIFO (if necessary subject to priorities) order. In a distributed system there is not a central point for queueing such requests so it will take extra work to get the cooperating processes to agree to handle requests in a fair order.

An early solution to this problem was presented by [RiAg81]. In this solution a process that wanted to enter a critical region would request permission from all processes that might want to enter the critical section. A process could only enter the critical section when all processes returned their permission. If

there where N processes that wanted to enter the critical section (and controlled a copy of the critical section) then this algorithm would require 2(N-1) messages. Other solutions to this problem can be found in [SuKa85, Maek85].

Mutual exclusion problems can be illustrated most easily in DGP by using the game playing phase. In this phase a ***shared game state*** will be maintain by each player in the game. When a player makes a "move" it will update the shared game state. There are a number of ways that the updating of the shared game state can be controlled. If the game does not require that players play in a specific order, then we can let players operate asynchronously and the mutual exclusion algorithm that controls the shared game state will not address problems of fairness. In another example we allow players to operate asynchronous but if two or more players try to access the shared game state at the (approximately) same time, the player that has accessed the shared value the least number of times should be given priority. In a third example, we could force the players to make moves in a fixed order, thus no player should access the shared game state if the other players have not taken their turns. In this case the algorithm should address the problem of a player failing and thus stopping the other players from making further moves.

6.4.3 Distributed Termination

Another interesting problem in distributed computing is the problem of distributed termination. Some distributed algorithms can be characterized as operating in a series of phases: phase 1, phase 2, ... , phase N. Within each phase processes operate asynchronously. The end of a phase represents a point of synchronization where all cooperating processes must determine that they have all completed phase i and are ready to start phase i+1. The distributed termination problem can be characterized as the problem of determining if all

processes have completed a particular phase of an algorithm. Distributed termination can also be used to identify deadlocks. For example we could have a set of processes waiting to receive messages and no processes sending messages. A distributed termination protocol could be used to detect this situation.

There are a number of ways in which the distributed termination problem can be addressed. In the simplest case a process may have two states: *working* or *terminated.* When a process enters the terminated state it cannot re-enter the working state (if it has terminated a phase it will enter the working state when it finds that all other processes are terminated and enters the next phase). In this case when a process enters the terminated state it simply announces this fact to the other processes and waits for all other processes to report that they are terminated. If processes and the communication system are reliable, this problem is relatively easy. A more complex distributed termination problem allows a process to be in one of three states: *working, idle,* or *terminated.* The working and terminated states have the same meaning as before. The idle state indicates that the process is no longer doing any useful computation but it may receive messages that will allow the process to re-enter the working state. A process moves from the idle state to the terminated state when it can guarantee that no message will arrive making it re-enter the working state. In general a process will be able to enter the terminated state only when all other processes are idle and there are no outstanding messages that could make any process re-enter the working state. Solutions to this problem can be found in [DiSc80, Fran80, MiCh82].

The problem of distributed termination can be investigated in many places in a DGP algorithm. The structure of the DGP environment fits nicely into the multi-phase structure algorithms described above. For example we could define the tournament formation phase to be completed when the there are no more

processes that want to join the tournament. In order to discover this fact we could make sure that all players that could possibly join the tournament are idle and that there are no outstanding messages indicating that a player wants to join the tournament.

We can use the round formation phase as another example. Consider a round formation phase in which players must be matched in pairs. Each player could be in one of three states *unmatched*, *temp_matched*, or *permanent_matched*. The round formation phase is completed when all players are in the permanent_matched state. An unmatched player will try to become temp_matched with another player. A temp_matched player may break its current matching and become temp_matched with another player if the new match is a better match (based on some criterion such as distance between players). A player can only enter the permanent_matched state if it knows that a better match will not be offered. In general this will happen when all players are in the temp_match state and there are no outstanding messages offering a new match.

6.4.4 Voting and Election Algorithms

Voting and election algorithms represent another important class of distributed algorithms. These type of algorithms are used to increase the reliability of a computation. For example, in some algorithms one process, called the coordinator, will perform some special functions. Since the coordinator could fail, there must be a mechanism that allows a new process to become a coordinator. One way in which a new process can become the coordinator is to have an election of a new coordinator after the failure of the old coordinator is recognized. Another use of voting algorithms is to check that a computation was done correctly. Assume that a particular computation is performed by a number of

processes. If some of the processes perform the computation incorrectly, there will be more than one result of the computation from which to choose. The result that will be used can be chosen by having the processes vote on each result. In simple election algorithms each process may have a single vote while in more complex voting schemes processes may have multiple votes (and each process will not necessary have the same number of votes).

Some solutions to elections algorithms have been proposed in [Giff79, Garc82, GaBa85]. The *Bully algorithm* [Garc82] is an election algorithm in which the process will the highest process id or priority that recognizes the failure of the coordinator will become the new coordinator. In this algorithm when a process recognizes that the coordinator has failed it broadcasts this fact to all other processes. If no response is returned in T time units, this process becomes the new coordinator. If a process with a higher process id responds within T time units then the process defers to the process with the higher id. This algorithm requires many assumptions, most notablely the communication system must be reliable. In [GaBa85] a method of distributing votes among a number processes in order to achieve some reliability constraints is discussed.

Elections can be useful in many places in the DGP environment. For example, after the tournament formation round is completed the players could elect a tournament manager who would control the round formation phases and who would keep information about the current status of the tournament. For another example we could have a player split itself into a number of identical players. When it was the players turn to move each copy of the player would chose the next move it thought should be taken. The players would then vote on which move (there may be more that one if some copies of the player are unreliable) should actually be taken. Another use of voting can take place when a player wants to join a tournament. One method that can be used to

determine if the player should be allowed to join the tournament is to have the current members of the tournament vote on each new player's acceptance.

6.4.5 Byzantine Agreement

A problem that has recently attracted a lot of attention is that of byzantine agreement. Byzantine agreement algorithms try to solve the problem of a set of processes reaching agreement when some of the processes may be faulty. The types of faults that may occur are very extreme. The faulty processes may fail in arbitrary and malicious ways. The faulty processes may try to mislead the correct processes by sending different information to different processes. The goal of a byzantine agreement is to have all correct processes reach the same conclusion.

Two important papers on byzantine agreement are [PeSL80, LaPS82]. In these papers the basic results of this problem in synchronous networks are established. The basic result is that given a system of N processes in which at most M processes can fail, an agreement is possible only if $N \geq 3M+1$. This solution assumes that a faulty process may send different messages to different processes and that it may pass incorrect information about the known state of the system. Another algorithm which makes use of authentication (unforgible identifiers) allows an agreement to be reached for $N \geq M$.

To some extent each of the problems discussed above are concerned with distributed agreement. If it is possible for the processes to exhibit byzantine behavior, then each of those problems will have to address the problem of byzantine agreement. The agreement required in these algorithms is more complex than those assumed in the past work on byzantine agreement and requires further research if byzantine behavior is to be adequately address. Also the players could be required to follow a set of rules when they play a game.

Byzantine behavior can be investigated by identifying that a player is not following the rules and developing methods to exclude players that do not follow the rules from further participation in the tournament.

6.5. Solutions to DGP Problems

In this section we present some possible solutions to DGP problems. Our purpose is to illustrate how problems can be constructed in the DGP environment and to discuss some of the assumptions that can be made. We discuss three problems: a tournament formation problem, a player matching problem in a broadcast network, and a virtual ring structure for a round robin tournament. In our examples we use simplified interprocess communication that includes asynchronous send, blocking receive, and broadcasting. For further discussion of interprocess communication in distributed systems see [Geha84, Shat84, LGKN86].

6.5.1 Tournament Formation in a Broadcast Network

The goal of the tournament formation phase is to identify a set of players which want to form a tournament. This procedure starts when one or more players announce that they would like to begin a tournament. When other players hear the announcement of a new tournament, they can send a message to the creator of the tournament indicating that they would like to participate in the tournament. The tournament formation can end when all players that might want to partake in the tournament have responded to the tournament creator, when the tournament creator stops accepting new players, or when the members of the tournament decide (as a group) to stop accepting new players.

An important issue in the tournament formation problem is to determine the potential players that might want to join a tournament. The are two general categories that we will consider: ***known players*** and ***anonymous players.*** If

each player knows the identity of the other players that might want to play, we have the case of known players. In this case we assume we have a set of players, $P = \{p_1, p_2, \cdots, p_n\}$, where each player will be in one of four states, $S = \{idle, playing, joining, creating\}$. In this situation a player who is creating a tournament could wait for responses from all other potential players.

If a player does not know the identity of the other players that might want to join the tournament, we have the case of anonymous players. In this case, a player creating a tournament cannot wait for each player to respond in order to determine the end of the tournament formation phase. With anonymous players a player may arrive (or be created) at any time. When it is created it could try to create a new tournament or join a current tournament. If it cannot create a new tournament or join a tournament it will wait for other players to arrive or it will leave (die). A possible solution to this problem is to create *game managers*. The game managers will act as servers that will try to place a player in a tournament. The players will be the clients of the game managers. Any player will be able to asked a game manager to place it in a tournament. The identity of a potential player is not known until it contacts a game manager. The game managers will have well-known ids so that they can be easily contacted by the players and other game managers. This type of structure can be used to investigate client/server type applications.

In Figure 6.3 we show one possible solution to the tournament formation problem. In this solution we assume that the players and the communication medium are reliable and that we have known players. A player may decide to either create a tournament or to try and join a tournament. If a player wants to create a tournament, it will broadcast a "*create tournament*" message to all other players. If a player wants to join a tournament it will wait to receive a "*create tournament*" message. When such a message arrives, the player will

decide if it wants to join that particular tournament. If it does want to join, it will send an "*I want to join*" message; otherwise, it will send an "*I do not want to join*" message. When the tournament creator receives responses from all possible players, it will broadcast a list of the members of the tournament.

In the solution presented in Figure 6.3 a reliable broadcast protocol is assumed. However, broadcast messages may arrive in different order to different players. If multiple players try to create a tournament at the same time, more than one tournament will be created. In the solution presented in Figure 6.4 we assume that we have reliable ordered broadcasting [ChMa84]. With reliable ordered broadcasting all players will receive all broadcast messages in the same order. We use this fact in the solution in Figure 6.4 to have one tournament created even if many players try to create a tournament. In this example, players that want to join a tournament will join the first available tournament and players that try to create tournaments when another tournament is being created will join the tournament that began creation first. The tournament that began creation first is determined by the first "*create tournament*" message that is received by all players. The algorithm is similar to the algorithm in Figure 6.3 except that a player that tried to create a tournament will also listen for "*create tournament*" messages. If it finds that its "*create tournament*" message was not the first "*create tournament*" message to be sent it will join the tournament started by the player who sent the first "*create tournament*" message.

Our final solution to the tournament formation problem involves game managers. We assume the existence of a set of game managers each of whom may control one tournament at a time. When player wants to join a tournament, it will send a message to a well-known game manager. The game manager will try to place the player in a tournament. If the game manager is currently forming a tournament, it can accept the player into that tournament;

```
CREATING A TOURNAMENT:
      broadcast("create tournament");
      for i := 1 to n-1   {n is the number of potential players}
            receive(player_id, message);
            if message = "I want to join" then
                  add_player(player_id, tournament_members);
      end for;
      broadcast("begin tournament",tournament_members);
      {begin round formation}

JOINING A TOURNAMENT:
   joined := false;
   while not joined do
      receive(player_id, "create tournament");
      if want_to_join() then
            send(player_id, "I want to join");
            joined := true;
            receive(player_id, "begin tournament", tournament_members);
      else
            send(player_id, "I do not want to join");
      end if;
    end while
    {begin round formation}
```

Figure 6.3: Tournament formation with reliable unordered broadcasting.

otherwise, the game manager will send a "*request for bid*" to the other games managers. The player will be assigned to the game manager that returns the best bid. In this case the bid will be the number of players currently in the tournament that is being formed. The game manager with the least number of players (lowest bid) will be awarded the new player. The solution is presented in Figure 6.5. Note that in this example we have a number of independent cooperating sets of processes: the game managers and each tournament that is taking place. If a process wants to broadcast a message to other processes that are cooperating with it, it will have to have a way of indicating the destinations of the broadcast. One way that this can be done is to have the set of cooperating processes form a dynamic group [AhBe85, ChZw85, GeNi85]. Each dynamic group will have a unique id. This id will be used as destination of the broadcast message. For example, the statement broadcast(group1,message) will cause the message to be sent to all the processes in group1. In the above example we would form one group for all game managers and one group for each tournament managed by a game manager.

6.5.2 Player Matching in a Broadcast Network

The responsibility of the round formation phase is to group players into games. In this section we look at the problem of matching players in order to form two player games. We are given a set of players each of which wants to be matched with an opponent (another player in the tournament). Each player has a history associated with it that provides a record of the players with which it has previously played. The players must decide on a matching that satisfies the following: no player is matched with a previous opponent and the maximum number of matches is created in each round. In general the maximum number of matches is $\lfloor n/2 \rfloor$, where n is the number of players in the tournament. However, because of the history associated with each player this many

```
CREATING A TOURNAMENT:
      broadcast("create tournament");
      created := true;
      receive(player_id, "create tournament");
      if player_id ≠ my_id then
            send(player_id, "I want to join");
            receive(player_id, "begin tournament", tournament members);
            {go to the round formation phase}
      end if;
      for i := 1 to n-1
            receive(player_id, message);
            if message = "I want to join" then
                  add_player(player_id, tournament_members);
      end for;
      broadcast("begin tournament",tournament_members);
      {begin round formation}

JOINING A TOURNAMENT:
      receive(player_id, "create tournament");
      send(player_id,"I want to join");
      receive(player_id, "begin tournament", tournament_members);

      {begin round formation}
```

Figure 6.4: Tournament formation with reliable ordered broadcasting.

matches will not always be possible. Thus, some players may sit idle in a round. We assume each player knows ids of all other players and that the player ids are (1,2, ... n).

One solution to this problem is to have each player broadcast the complement of its history (all players with which it could play). After this phase each player has the same information as every other player and if each player uses the same algorithm they can all find the same matching solution. The information that indicates which players can be matched with other players can be formed into a graph structure and this graph can be searched. The graph will be a directed acyclic graph with n levels where a node on level i represents a possible match for player i. The graph can be searched using a modified depth first search (DFS) (the modification takes into account the fact that if player i is matched with player j this players cannot be used later in the graph). In the algorithm presented in Figure 6.6 the DFS procedure can be called many times. Each time it runs into a dead end it will return the path it has found and the length of the path. If the path length satisfies the minimum path length requirement then the path is accepted; otherwise, the search is continued. If the best path is required then the minimum path length should be n (a match for each player). Note, this does not imply that a path of length n will be found since one might not be possible (it depends on the history of the players). The algorithm for this solution is presented in Figure 6.6.

The solution presented in Figure 6.6 does not take advantage of the potential parallelism of the problem. With a simple modification to the solution in Figure 6.6 we can introduce some parallelism. Each player will choose a different match for player 1 (if there are not n choices for player 1, a few players will search from the same starting point or we could have each player choose a different match for player i, where player i is the player with the most

```
RECEIVE PLAYER REQUEST:
      receive(player_id,"I want to join a tournament");
      if tournament_forming then
            send(player_id, "you are accepted", group_id);
            add_player(player_id, tournament_members);
            if size(tournament_members) ≥ min_tournament_size then
                  go to round formation phase
            end if;
      else
            broadcast(game_managers,"request for bid")
            low_bid := -1;
            low_manager := null;
            for i := 1 to num_game_managers -1
                  receive(manager_id, bid);
                  if bid < low_bid then
                        if low_manager ≠ null then
                              send(low_manager, "reject", null);
                        end if;
                        low_bid = bid;
                        low_manager := manager_id;
                  else
                        send(manager_id, "reject", null);
                  end if;
            end for;
            if low_bid > -1 then
                  send(low_manager,"you win", player_id);
            else
                  send(player_id, "no tournament available");
            end if;
      end if;

RECEIVE REQUEST FOR BIDS:
   while size(tournament_members) < min_tournament_size do
      receive(manager_id,"request for bid");
      if tournament_forming or not tournament_in_progress then
            tournament_forming := true;
            bid := size(tournament_members);
            send(manager_id,bid);
            receive(manager_id,message, player_id);
            if message = "you win" then
                  send(player_id, "you are accepted");
                  add_player(player_id,tournament_members);
            end if;
      else
            send(manager_id,-1);
      end if;
   end while;
   tournament_in_progress := true;
   {round formation}
   broadcast(tournament_id,"begin tournament", tournament_members);
   {complete round formation}

JOINING A TOURNAMENT:
      send(game_manager, "I want to join a tournament");
      receive(game_manager, message, tournament_id);
      if message = "you are accepted" then
            receive(game_manager, "begin tournament", tournament_members);
            {go to round formation}
      else
            {try again or die}
      end if;
```

Figure 6.5: Tournament formation with game managers

possible opponents). In this way instead of having each player look at all possible paths we partition the paths into n different sets. Each player will search one set. There is of course a price to pay for this speed-up. Since each player searches a different set of paths, the players will not automatically arrive at the same solution. Each player will have to broadcast the best solution that it found. The players will then use the best solution that is found.

6.5.3 A Virtual Ring Structure for a Round Robin Round Structure

In this problem we assume the tournament formation phase has been completed and we present an algorithm that performs round formation and round termination. Assume that each game in the tournament requires exactly two players and the players are mobile. We also assume that there are $2n$ players and m processors, where $m \geq n$. During the tournament formation phase, players are paired at different processors so that there is at most one pair of players per processor. The round formation phase requires each player to identify itself to the player located at the same processor. When this is done, the game playing phase may begin. In the round termination phase, players must wait for all the games in the round to end. When the games are over, the players will rotate in a predefined pattern to get into position for the next round. Figure 6.8 shows the different patterns that each round will have for a six-player tournament.

The tournament can start at any round depending on the initial pairing of players. After the initial pairing of players is made, the rounds will be followed in order (round 1 follows round 5) until five rounds have been played. In general, a tournament with $2n$ players will require $2n-1$ rounds. Each rotation of a player requires the player to move to a new processor. The rotations that a player will make in this algorithm can be described as follows. After the tour-

```
PLAYER MATCHING:
    broadcast("potential players", player_list);
    for i := 1 to n-1 {n is the number of players in the tournament}
        receive(player_id,"potential players", player_list);
        add_player_list(player_id,player_list,player_list_array);
    end for;
    length := 0;
    DFS(player_list_array,path,length);
    while length < minimum_acceptable_length do
        if length > best_length then
            best_length := length;
            best_path := path;
        end if;
        DFS(player_list_array,path,length);
    end while;
    {at this point a path has been found and the game playing can begin}
```

Figure 6.6. Player matching in a broadcast network

nament formation phase is completed, each player knows the names of all the other players in the tournament. The players will order themselves 1,2, $\cdots$,$2n$ according to the relative order of their names. The rotation pattern will use this relative ordering. Player 1 will never move. Player 2 will follow player 3. Player $2n-1$ will follow player $2n$. For the remaining players, player m will follow player $m+2$ if m is odd or player $m-2$ if m is even. This rotation pattern allows the tournament to be completed in the fewest number of rounds. Figure 6.8 shows the rotation pattern for the six-player case. Remember that the players in each game are assumed to be located at the same processor. Also, one processor handles at most one game. These are not assumptions which are part of the DGP environment. They are only assumptions of our solution to this problem.

Round Robin Pairings					
Games	Round 1	Round 2	Round 3	Round 4	Round 5
Game 1	1,2	1,4	1,6	1,5	1,3
Game 2	3,4	2,6	4,5	6,3	5,2
Game 3	5,6	3,5	2,3	4,2	6,4

Figure 6.8. Round and game assignments in a six-player tournament

Some readers may question the fact that the algorithm allows players to request to be moved to another processor. This can be accomplished by issuing a *migrate* command. We assume that the migration is always successful. In a real system, the migration command would be a request to the operating system which would not always be granted. If the request was not granted, the process would take appropriate actions. In this case the algorithm would not

```
PARALLEL PLAYER MATCHING:
      broadcast("potential players", player_list);
      for i := 1 to 2n-1
            receive(player_id,"potential players", player_list);
            add_player_list(player_id,player_list,player_list_array);
      end for;
      {the next line chooses the starting point for each player}
      {player_list_array[1,0] gives the number of possible matches for player 1}
      path[1] := player_list_array[1,min(my_id,player_list_array[1,0])]
      length := 0;
      DFS(player_list_array,path,length);
      while length < minimum_acceptable_length do
            if length > best_length then
                  best_length := length;
                  best_path := path;
            end if;
            DFS(player_list_array,path,length);
      end while;
      {at this point a path has been found that can be broadcast to the other player}
      broadcast("path announcement", path)
      best_path = null;
      best_length = 0;
      for i := 1 to n -1
            receive(player_id,"path announcement", path)
            length := get_length(path);
            if length > best_length then
                  best_path := path;
                  best_length := length;
            end if;
      end for;

      {go to round formation}
```

Figure 6.7. Parallel player matching in a broadcast network

change significantly. The problem of scheduling processes in a distributed system has drawn considerable attention [PoMi83, WaJu83, NiXG85, StRC85, TaTo85]. In many cases, the scheduling problem is addressed as if processes are all independent. Some more recent work [StSi84] has addressed the scheduling of related processes. In an environment which supports distributed programming, a question arises concerning which processes are closely cooperating and thus would benefit from being scheduled at the same processor. If the relationship between the processes is static, this coupling can be indicated at compile time. However, if the relationship is dynamic, some run time indication of the relationship between processes is necessary. We are currently investigating the problems associated with providing this type of run time indication. The solution described above can be found in Figure 6.9.

6.6 Some Additional Problems

Listed below is a set of potential problems of interest. Each of the problems listed below can be given various degrees of difficulty depending on the assumptions about the players and the communication. Some possible sets of assumptions include: 1) reliable, fail-stop, and byzantine players 2) reliable one-to-one communication, unreliable one-to-one communication, 3) no broadcasting, unreliable broadcasting, reliable unordered broadcasting, and reliable ordered broadcasting. One choice can be made from each category.

Counting Players in a DGP Tournament: In this problem we are interested in knowing the number of currently active players in the tournament. We assume that a player may leave the tournament without warning or that players may die without warning. Solutions to this problem will investigate the cost (in messages or frequency of communication) required to reach a certain level of accuracy concerning the actual number of players. The

```
ROUND FORMATION:
    opponent_id := player_list[round];
    send(opponent,"I am ready");
    receive(opponent,"I am ready")'
    {begin game playing phase}


ROUND TERMINATION:
    if my_id < opponent_id then
        broadcast(tournament_id, "game over");
    end if;
    for i := 1 to n-1
        receive(player_id,"game over");
    end for;
    round := round + 1;
    if round < n -1 and my_id ≠ 1 then
        processor_id := get_processor_id();
        send(successor(my_id),"here I am",processor_id);
        receive(predecessor(my_id), "here I am", new_processor_id);
        migrate(new_processor_id);
    end if;
    {go to round formation if the tournament is not completed}
```

Figure 6.9: Round formation and termination in a round robin tournament

evaluation of a solution can be based on an assumed probability of a player dying or leaving the tournament. The current estimate of the number of existing players can be used to decide if the tournament can continue. This problem is related to problems where reliability conditions would require a certain degree of redundancy. If this degree of redundancy is not maintained, the operation would move into a shutdown or hold state.

Player Matching: The problem of player matching is a very interesting one. We saw one example of this problem in Section 2. These problems are useful for investigating concepts in distributed scheduling, distributed decision making, and process mapping and grouping in loosely coupled multiprocessors. The matching problem does not always have to be concerned with matching two players. The general problem is one of forming subsets of players that satisfy certain criteria. Some more specific problems are listed below.

Player Matching in a Point-to-Point Network: This is similar to the problem that we discussed in Section 2. We have a set of players that belong to a tournament where each player wants to get matched with one opponent. The goals of a solution to this problem are: no player is matched with a previous opponent, the maximum number of matches are achieved, and the matches represent a least cost matching. In this case, the cost is the sum of the links that separate each pair of opponents. To actually find an optimal matching may require an exhaustive search, but heuristics could be developed to find a near optimal solution in a shorter time.

General Round Robin Tournament: This is a generalization of the problem discussed in Section 2. Assume each game requires exactly N players and there are M players in the tournament. Create a round formation strategy that

allows each player to play with every combination of $N-1$ players and to have the tournament complete in the least number of rounds. Note, this problem can also be seen as a player matching problem. In this case the subset size is N instead of 2 and the cost criterion is a global cost associated with the tournament rather than the local cost of one round. The goal of minimizing a global cost may require a look ahead feature that estimates how good a matching is in terms of estimated future cost. Solutions may be able to make use of min/max and α/β searching and pruning techniques.

Tournament Formation in a Multi-Tournament Environment: This problem is concerned with developing strategies that allow players to join a particular tournament when there are many tournaments seeking players. Solutions can be based on either players competing to get into certain tournaments or tournaments (represented by their manager or acting as a group) competing to get certain players. This problem can be used to investigate various distributed decision making strategies. In particular it can be used to investigate different methods by which a process is allowed to join a particular dynamic group.

6.7. CONCLUSIONS

We have presented an environment of distributed programming called distributed game playing. This environment provides a good environment for the investigation of the basic features of distributed algorithms: negotiation, remote state maintenance, and reliability. An almost unlimited number of problems can be created from this environment. This environment will provide many manageable size problems. The environment is very flexible. The problems can be defined so the interests of the user can be highlighted and other factors can be minimized. In general, the interaction between players can be emphasized

while the sequential parts of the players can be kept to a minimum. Problems can be designed to provide practice in distributed programming. Other problems can be designed to evaluate an interprocess communication system. For example, a user can create a set of problems which have varying degrees of communication and computation requirements to produce a varied workload for testing a distributed system.

We are currently developing more problems and solutions in the DGP environment. The development of appropriate higher-level language structures is essential to the development of distributed programming concepts. We are investigating standard player interactions in order to discover general structures that can be formalized into higher-level language constructs. We are also investigating the possibility of developing distributed debugging and execution control tools for our DGP environment. By creating expert game playing strategies we can investigate problems related to artificial intelligence such as communicating or cooperating expert systems.

CHAPTER 7
CONCLUDING REMARKS AND FUTURE WORK

The importance of distributed computing is growing rapidly. With increased requirements for computation speed, highly reliable systems, and intergrated communication and computing resources the area of distributed computing will continue to be important for many years. There are many problems that must be addressed if distributed systems are to reach their full potential. In this dissertation we addressed three problems: 1) the design of a an interprocess communication system that can take advantage of the broadcast nature of many LANs and provides a foundation for the development of distributed applications, 2) a comparison of two load balancing algorithms, bidding and drafting, in an Ethernet based environment of workstations, and 3) the development of an environment for investigating distributed applications.

7.1 Summary

The following results were achieved through this research:

(1) An interprocess communication system was designed. This system provides a wide variety of communication primitives that should satisfy the needs of any distributed application designer. An important feature of this system is the concept of a dynamic group of processes. This allows application processes to make use of the broadcast feature of many LANs in a multiuser multiprogrammed environment. The dynamic group concept also provides a good encapsulation for distributed applications.

(2) A comparison of the bidding and drafting algorithm shows that the bidding algorithm is able to outperform the drafting algorithm in an Ethernet based LAN of workstations with a small (10 or less) stations. The extra

overhead incurred by the executions of the drafting algorithm results in the bidding algorithm outperforming the drafting algorithm even though the drafting algorithm maintains greater knowledge about the system. Through the use of emulation techniques we were able to capture the real overhead of these algorithms, both computation overhead and communication head, that is often lost in a simulation study.

(3) The basic features of distributed algorithms are examined. These feature include the difference between naturally and artificially distributed algorithms and the features of negotiation, remote state maintenance, and reliability.

(4) An environment for investigating distributed applications called distributed game playing was developed. The basic features of the environment were defined. The relationship between the distributed game playing environment and problems in distributed computing was shown through the use of some illustrative examples. The distributed game playing environment provides a flexible environment for investigating distributed algorithms and can be used to illustrate known problems and to develop new problems in distributed computing.

7.2 Future Work

In this research we have touched on only a few of the problems in distributed computing. Below we list some other problems that should be addressed.

(1) Scheduling in large networks and internetworks.

Consider the problem of having hundreds or thousands of computers connected by a communication system. For example, we could have a number of local networks of workstations interconnected together. Can a distributed scheduling algorithm take advantage of all the computers in the internetwork?

Requiring the scheduler on a processor to communicate with all other processors in the internetwork may cause too much communication overhead. A possible solution is to create a hierarchy of dynamic groups of schedulers that cooperate to schedule processes throughout the internetwork.

(2) Scheduling in Message Passing Multiprocessors

A message passing multiprocessor such as a hypercube is also a kind of distributed architecture. This architecture is based on a point-to-point communication medium. One could investigate the performance of the drafting and bidding algorithms in this type of architecture. Also, in many causes the message passing multiprocessor will be used to solve one multiprocess problems. The issues associated with scheduling multiprocess problems should also be developed in order to take advantages of this type of architecture.

(3) Incorporating the dynamic group concept into message based multiprocessors

In our research thus far we have looked at the dynamic grouping concept only for broadcast based LANs. If we have a large message based multiprocessor that is to be shared by multiple users, then the dynamic group concept may be useful. For example in order to test an algorithm in a message based multiprocessor, a user may only require a subset of the processors. This would allow the multiprocessor to be shared by many users during the development phase. After the algorithm had been tested it could be allocated to the whole multiprocessor. In this type of an environment the dynamic group concept may provide a useful encapsulations for controlling processors. A specific problem that should be solved in this context is an efficient method of multicasting in a point-to-point environment.

(4) Development of more problems in the DGP environment

The DGP environment provides the possibility of creating many problems related to distributed computing. We will continue to develop problems and solutions to problems in this environment. We will also try to find related problems and see if a classification of problems in the DGP environment can be developed.

Of course there are many other problems in distributed computing that can be investigated. Such as those related to fault tolerance or security. The above problems were mentioned because of their relationship to the work presented in this dissertation.

BIBLIOGRAPHY

BIBLIOGRAPHY

[AhBe85] Ahamad, M. and Bernstein, A., "Multicasting Communications in Unix 4.2BSD," *Proc. of the 5th Int'l Conference on Distributed Computing Systems,* pp. 80-87, May 1985.

[AnSc83] Andrews G.R., and Schneider, F.B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, pp. 3-43, Mar. 1983.

[BeCh84] Berglund, E. J. and Cheriton, D. R., "Amaze: A Distributed Multi-Player Game Program Using the Distributed V Kernal," *Proc. of the 4th Int'l Conference on Distributed Computing Systems* pp. 248-253, May 1984.

[Birr82] Birrell, A.D. et al, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM*, April 1982.

[Brin78] Brinch-Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, pp. 934-941, Nov. 1978.

[BiNe84] Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, pp. 39-59, Feb. 1984.

[ChLa85] Chandy, K.M. and Lamport, L., "Distributed Snapshots: Determining Global State of Distributed Systems," *ACM Transactions on Computer Systems*, pp. 63-75, Feb. 1985.

[CheM84] Cheriton, D.R. and Mann, T.P., " Uniform Access to Distributed Name Interpretation," *Proceedings of the 4th International Conference on Distributed Computing Systems*, pp. 290-297, May 1984.

[ChZw85] Cheriton, D. R. and Zwaenepoel, W., "Distributed Process Groups in the V Kernal," *ACM Trans. on Computer Systems*, pp. 77-107, May 1985.

[ChMa84] Chang, J. M. and Maxemchuk, N. F. "Reliable Broadcast Protocol," *ACM Trans. on Computer Systems*, pp. 251-273, Aug. 1984.

[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.

[Chiu84] Chiu, S.Y., "Debugging Distributed Computations in a Nested Atomic Action System," *Ph.D. Dissertation*, MIT/LCS/TR-327, Lab. for Computer Science, MIT, Dec. 1984.

[Cook80] Cook, R.P., "Mod- A Language for Distributed Programming," *IEEE Trans. on Software Eng.*, pp. 563-571, Nov. 1980

[DiSc80] Dijkstra, E.W. and Scholten, C.S., "Termination Detection for Diffusing Computations," *Inf. Process. Lett.*, pp. 1-4, Aug. 1980.

[DLPS85] Dolev, D., Lynch, N., Pinter, S., Stark, E. and Weihl, W., "Reaching Approximate Agreement in the Presence of Faults," *Tech. Report*, MIT/LCS/TM-276,Lab. for Computer Science, MIT, May 1985.

[DwLS84] Dwork, C., Lynch, N. and Stockmeyer, L., "Consensus in the Presence of Partial Synchrony," *Tech. Report*, MIT/LCS/TM-270, Lab. for Computer Science, MIT, Oct. 1984.

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

[Farb73] Farber, D. J., et. al., "The Distributed Computing System," *Proc. Compcon Spring 73*, pp. 31-34, 1973.

[Feld79] Feldman, J.A., "High-Level Programming for Distributed Computing," *Communication of the ACM*, pp. 353-368, June 1979.

[FiMa85] Finkel, R. and Manber, U., "DIB-A Distributed Implementation of Backtracking," *Proc. of the 5th Int'l Conference on Distributed Computing Systems*, pp. 446-452, May 1985.

[Fran80] Francez, N., "Distributed Termination," *ACM Trans. on Prog. Lang. and Systems*, pp. 42-55, Jan. 1980.

[Garc82] Garcia-Molina, H., "Elections in a Distributed Computing System," *IEEE Trans. on Computers*, June 1982.

[GaBa85] Garcia-Molina, H. and Barbara, D., "How to Assign Votes in a Distributed System," *Journal of the ACM*, pp. 841-860, Oct. 1985.

[Geha84] Gehani, N. H., "Broadcast Sequential Processes (BSP)," *IEEE Trans. on Software Eng.* , pp. 343-351, July 1984.

[GeNi85] Gendreau, T. B. and Ni, L. M., "A Distributed Game Playing Model for a Distributed Programming Environment," *Tech. Report*, Department of Computer Science, Michigan State University, 1985.

[Giff79] Gifford, D.K., "Weighted Voting for Replicated Data," *Proceedings of the 7th Symposium on Operating System Principles*, pp.150-162, Dec. 1979.

[GoCK79]Good, D.I., Cohen, R., and Keeton-Williams, M., "Principles of Proving Concurrent Programs in Gypsy," *Proc. 6th ACM Symp. Prog. Lang.*, pp. 42-52, Jan 1979.

[HLLR85]Hasegawa, S., Liu, J., Lu, C. and Railey, M., "Reliable Clock-Driven Process Synchronization Algorithms," *Tech. Report*, UIUICDS-R-85-1195, University of Illinois, 1985.

[Hoar78] Hoare, C. A. R., "Communicating Sequential Processes," *CACM*, pp. 666-677, Aug. 1978.

[HwBr84] Hwang, K. and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1984.

[HwSN81]Hwang, K., Su, S.P. and Ni, L.M., "Vector Computer Architecture and Processing Techniques," *Advances in Computers*, Vol. 20, M. Yovits, Ed., New York: Academic Press, pp. 115-197, 1981.

[Isoi83] ISO/TC97/SC16, *Information Processing Systems - Open Systems Interconnection - Basic Reference Model*, ISO International Standard IS 7498, April 1983.

[Jone78] Jones, A.K., "The Object Model: A Conceptual Tool For Structuring Software," *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham and G. Seegmuller, Eds., Springer-Verlag, pp. 3-19, 1978.

[Kuma85]Kumar, D., " A Class of Termination Detection Algorithms for Distributed Computations," *Tech. Report*, TR-85-07, University of Texas at Austin, May 1985.

[KiGN86] King, C.T., Gendreau, T.B., and Ni, L.M., "Reliable Elections in Broadcast Networks," *Proceedings of the 1986 International Computer Symposium*, Taiwan, R.O.C, Dec. 1986.

[LGKN86]Liu, Y.H., Gendreau, T.B., King, C.T., and Ni, L.M., "A Session Layer Design of a Reliable IPC System in The Unix 4.2 Environment," *Proceedings of the 1986 IEEE Computer Networking Symposium*, Washington D.C., Nov. 1986.

[Lamp78] Lamport, L., "Time, Clocks, and Ordering Events in a Distributed System," *CACM*, July 1978.

[Lamp83] Lamport, L., "The Weak Byzantine Generals Problem," *Journal of the ACM*, pp. 668-685, July 1983.

[Lamp84] Lamport, L., "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Transactions on Programming Languages and Systems*, pp. 254-280, April 1984.

[LeCo84] LeBlanc, T.J. and Cook, R.P., "Broadcast Communications in StarMod," *Proceedings of the 4th International Conference on Distributed Computing Systems*, pp. 319-325, May 1984.

[Leei84] Lee, I., "A Programming System for Distributed Real-Time Applications," *Tech. Report MS-CIS-84-51,* University of Penn., Sept. 1984.

[LeWi85] LeBlanc, R.J. and Wilkes, T., "System Programming with Objects and Actions," *Proc. of the 5th Int'l Conf. on Distributed Computing Systems*, 1985.

[LaSP82] Lamport, L., Shostak, R. and Pease, M., "The Byzantine Generals Problem," *ACM Trans. on Prog. Lang. and Systems*, pp. 382-401, July 1982.

[LaSt81] Lampson, B.W. and Sturgis, H.E., "Atomic Transactions," *Lecture Notes in Computer Science*, Springer-Verlag, pp. 246-265, 1981.

[LiLi83] Lian, R.C. and Liu, M.T., "Cells: An Approach to Design of a Fault Tolerant Network Operating System," *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, pp. 163-172, Oct. 1983.

[Lisk79] Liskov, B., "Primitives for Distributed Computing," *7th ACM Symp. on Operating Systems*, pp. 33-42, 1979.

[LiSc83] Liskov, B. and Scheilfler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Trans. on Programming Languages and Systems*, pp.381-404, July 1983.

[LuSw85] Lundstrom, S.F. and Swartzlander, E.E. Jr., "Forward: Advances in Distributed Computing Systems," *IEEE Transactions on Software Engineering*, pp. 1092-1096, Oct. 1985.

[LyFr85] Lynch, N.A. and Frederickson, G.N., "A General Lower Bound for Electing a Leader in a Ring," *Tech. Report*, MIT/LCS/TM-277, Lab. for Computer Science, MIT, March 1985.

[MaLe80] Maccabe, A.B. and LeBlanc, R.J., "A Language Model for Fully Distributed Systems," *Proc. Dist. Comp. Compcon 80*, pp. 723-728, 1980.

[MaYe80] Mao, T.W., and Yeh, R.T., "Communication Port: A Language Concept for Concurrent Programming," *IEEE Trans. on Software Eng.*, pp. 194-204, Mar. 1980.

[Maek85] Maekawa, M., "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems," *ACM Transactions on Computer Systems*, pp.145-159, May 1985.

[MeBo76] Metcalf, R. M. and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM*, pp. 395-404, July 1976.

[MiCh82] Misra, J. and Chandy, K.M., "Termination Detection of Diffusing Computations in Communicating Sequential Processes," *ACM Trans. on Prog. Lang. and Systems*, pp. 37-43, Jan. 1982.

[Moss81] Moss, E., "Nested Transactions: An Approach to Reliable Distributed Computing," *MIT Lab for Computer Science*, TR 260, April 1981.

[NiGe86] Ni, L.M. and Gendreau, T.B., "REUSE: A Reliable Unified Service Environment for Distributed Systems," *Army Workshop on Future Directions in Computer Architecture and Software*, May 1986.

[NiLi82] Ni, L.M. and Li, X., "Modelling and Analysis of Computer Load Estimation," *Proceedings of the 1982 International AMSE Conference on Modelling and Simulation*, pp.72-76, July 1982.

[NiXG85] Ni, L. M., Xu, C. and Gendreau, T. B., "A Distributed Drafting Algorithm for Load Balancing," *IEEE Trans. on Software Eng.*, pp. 1153-1161, Oct. 1985.

[PeLS80] Pease, M., Shostak, R., and Lamport, L., "Reaching Agreement in the Presence of Fault," *Journal of the ACM*, pp. 228-234, April 1980.

[PoMi83] Powell, M. L. and Miller, B. P., "Process Migration in Demos/MP," *Proc. of the Ninth Symp. on Operating System Principles (ACM)*, pp. 110-119, 1983.

[PoPr83] Powell, M.L. and Presotto, D.L., "Publishing: A Reliable Broadcast Communication Mechanism," *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pp. 100-109, Oct 1983.

[RaSt84] Ramamritham, K. and Stankovic, J.A., "Dynamic Task Scheduling Distributed Hard Real-Time Systems," *Proceedings of the 4th International Conference on Distributed Computing Systems*, pp. 96-107, May 1984.

[Rash80] Rashid, R., " An Interprocess Communication Facility for Unix," *Tech. Report CMU-LS-80-124*, Carnegie-Mellon University, June 1980.

[RiAg81] Ricart, G. and Agrawala, A.K., " An Optimal Algorithm for Mutual Exclusion in Computer Networks," *CACM*, pp. 9-17, Jan. 1981.

[SaRC84] Saltzer, J.H., Reed, D.P. and Clark, D.D., " End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, pp. 277-288, Nov. 1984.

[ScWu84] Scheuermann, P. and Wu, G., "Broadcasting in Point-to-Point Computer Networks," *Proceedings of the 1984 Int'l Conference on Parallel Processing*, pp. 346-351, Aug. 1984.

[Schi81] Schiffenbauer, R.D., "Interactive Debugging in a Distributed Computational Environment," *M.S. Thesis*, MIT/LCS/TR-264, Lab. of Computer Science, MIT, 1981.

[Shat84] Shatz, S. M., "Communication Mechanisms for Programming Distributed Systems," *IEEE Computer* , pp. 21-28, June 1984.

[Shoc79] Shoch, J.F., "Inter-Network Naming, Addressing, and Routing," *Computer Networks*, pp. 72-79, Feb. 1979.

[Silb79] Silberschatz, A., "Communication and Synchronization in Distributed Systems," *IEEE Transactions on Software Engineering*, pp. 542-546, Nov. 1979.

[Silb80] Silberschatz, A., "A Survey Note on Programming Languages for Distributed Computing," *Proc. Dist. Comp., Compcon 80*, pp. 719-722, 1980.

[Smit80] Smith, R. G. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *IEEE Trans. on Computers*, pp. 1104-1113, Dec. 1980.

[Soll85] Sollins, K.R., "Distributed Name Management," *Ph.D. Dissertation*, MIT/LCS/TR-331, Lab. for Computer Science, MIT, Dec 1985.

[Spec82] Spector, A.Z., "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM*, pp.246-249, April 1982.

[Stal84] Stalling, W., *Local Networks: An Introduction*, McMillan Book Co.,1984.

[StSi84] Stankovic, J. A. and Sidhu, I. S., "An Adaptive Bidding Algorithm for Processes, Clusters, and Distributed Groups," *Proc. of the 4th Int'l Conference on Distributed Computing Systems*, pp. 49-59, May 1984.

[Stan84] Stankovic, J. A., "A Perspective on Distributed Computer Systems," *IEEE Trans. on Computers*, pp. 1102-1115, Dec. 1984.

[Stan85] Stankovic, J.A., *Reliable Distributed System Software*, IEEE Computer Society Press, 1985.

[StRC85] Stankovic, J.A., Ramamritham, K. and Cheng, S., "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Trans. on Computers*, pp. 1130-1143, Dec. 1985.

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

[SuKa85] Suzuki, I., and Kasami, T., "A Distributed Mutual Exclusion Algorithm," *ACm Trans. on Computer Systems*, pp. 344-349, Nov. 1985.

[Svob81] Svobodova, L., "A Reliable Object-Oriented Data Repository fir Distributed Computer Systems," *Proceedings of the 8th ACM Symposium on Operating System Principles*, pp.47-58, Dec 1981.

[SzSP85] Szymanski, B., Shi, Y. and Prywes, N.S., "Synchronized Distributed Termination," *IEEE Transactions on Software Engineering*, pp. 1136-1140, Oct. 1985.

[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," *Journal of the ACM*, pp. 445-465, April 1984.

[WaJu83] Wah, B.W. and Juang, J.Y., "An Efficient Protocol for Load Balancing on CSMA/CD Networks," *Proceedings of the 8th Conference on Local Computer Networks*, Oct. 1983.

[Walk83] Walker, B., Popek, G., English, R., Kline, C. and Theil, G. "The LOCUS Distributed Operating System," *Proc. of the Ninth Symp. on Operating System Principles (ACM)*, 1983.

[WiVa80] Witte, L. D. and Van Tilborg, A. M., "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE Trans. on Computers*, pp. 1133-1144, Dec. 1980.