INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
Pardo, Roberto

Interprocess Communication and Synchronization for Distributed Systems

The Ohio State University

University Microfilms International

300 N. Zeeb Road, Ann Arbor, MI 48106

18 Bedford Row, London WC1R 4EJ, England
INTERPROCESS COMMUNICATION AND SYNCHRONIZATION  
FOR  
DISTRIBUTED SYSTEMS  

DISSERTATION  

Presented in Partial Fulfillment of the Requirements for  
the Degree Doctor of Philosophy in the Graduate  
School of The Ohio State University  

By  
Roberto Pardo, B.Sc., M.Sc.  

*****  

The Ohio State University  
1979  

Reading Committee:  
Dr. Ming T. Liu, Chairman  
Dr. Sandra A. Mamrak  
Dr. Jerome Rothstein  

Approved by  
Dr. Ming T. Liu  
Advisor  
Department of Computer  
and Information Science
Esta tesis está dedicada a las tres personas más importantes en mi vida: mi Papa, mi Mama y Luly.
ACKNOWLEDGMENTS

I praise my advisor, Professor Ming T. Liu, for his constant support and encouragement during the development of this research. His faith and interest in all of my ideas are very much appreciated. His outstanding quality as a person deserves special mention.

I want to thank Dr. Sandra A. Mamrak and Professor Jerome Rothstein for serving on my reading committee. I also want to thank Dr. Bruce Weide for improving the clarity of some chapters.

Many of the graduate students have also contributed in one way or another to this research. Gojko A. Babic and K. S. Natarajan contributed to create during the early days of this research an interesting technical (and non-technical) atmosphere in which to work. Jerry Chou, through his constant interest, also contributed to clarify some of the ideas of this thesis.

I would also like to thank the Fulbright Commission and the National Science Foundation (Grant MCS-7723496) for their financial support.
VITA

October 15, 1952 .... Born - Bogota, Colombia

1974 ................. B. Sc., Systems Engineering, Universidad de los Andes, Bogota, Colombia

1977 .................. M. Sc., Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1977 .................. Teaching Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

1977-1979 .......... Research Associate, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Computer Organization and Architecture.
Dr. Ming T. Liu

Operating and Database Systems.
Dr. David K. Hsiao

Management Information Systems
Dr. Marshall C. Yovits
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>x</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Distributed Systems</td>
<td>5</td>
</tr>
<tr>
<td>Scope of Our Work</td>
<td>10</td>
</tr>
<tr>
<td>The Problem of Interprocess Communication</td>
<td>11</td>
</tr>
<tr>
<td>The Problem of Process Synchronization</td>
<td>16</td>
</tr>
<tr>
<td>Related Work, Contributions, and Main Results</td>
<td>20</td>
</tr>
<tr>
<td>Organization of The Dissertation</td>
<td>23</td>
</tr>
<tr>
<td>II. THE ARCHITECTURE OF DISTRIBUTED SYSTEMS</td>
<td>25</td>
</tr>
<tr>
<td>Distributed Algorithms</td>
<td>26</td>
</tr>
<tr>
<td>Distributed Control</td>
<td>29</td>
</tr>
</tbody>
</table>
## Reliability

- Distributed Software
- Distributed Operating Systems
- Distributed Database Systems

## Distributed Software

- A Distributed System Virtual Machine
- Communication Virtual Machines
- Remote Virtual Machines

## Summary

### III. MULTI-DESTINATION ROUTING ALGORITHMS

- Definitions
- Packet-Switching Architectures
- Single-Bus Architectures
- Loop Architectures
- Multiple-Bus Architectures

## III. MULTI-DESTINATION ROUTING ALGORITHMS

- Comparing Algorithms
- Broadcast Delay
- Discarding Cost
- Bit Rate

## III. MULTI-DESTINATION ROUTING ALGORITHMS

### IV. MULTI-DESTINATION PROTOCOLS

- Unreliable N-p Protocols
  - Selecting Routing
  - Addressing
- Reliable N-p Protocols
  - System Characteristics
    - Communication Systems
    - Processing Systems
- Best-effort-to-Deliver Protocols
  - The Basic Mechanisms
  - Differences with Reliable
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Types of IPCs</td>
<td>12</td>
</tr>
<tr>
<td>2. Differences Between Long-haul and Local Communication Architectures</td>
<td>98</td>
</tr>
<tr>
<td>3. Programming Models</td>
<td>131</td>
</tr>
<tr>
<td>4. Some Distributed Programming Languages</td>
<td>136</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Virtual Machines of a Distributed System Virtual Machine</td>
<td>45</td>
</tr>
<tr>
<td>2.</td>
<td>A Distributed Programming Language: A Tool for Designing Distributed Software</td>
<td>47</td>
</tr>
<tr>
<td>3.</td>
<td>Virtual Partial-broadcast Routing in Single-bus Architectures</td>
<td>60</td>
</tr>
<tr>
<td>5.</td>
<td>Virtual Partial-broadcast Routing in an Unidirectional Loop</td>
<td>64</td>
</tr>
<tr>
<td>6.</td>
<td>Imprecise Partial-broadcast Routing Using Full-broadcast Transmission in an Unidirectional Loop</td>
<td>65</td>
</tr>
<tr>
<td>8.</td>
<td>Imprecise Partial-broadcast Routing Using Partial-broadcast Transmission in a Bidirectional Loop</td>
<td>69</td>
</tr>
<tr>
<td>9.</td>
<td>Avoiding Unnecessary Repetition of Messages in Multiple-bus Architectures</td>
<td>72</td>
</tr>
<tr>
<td>10.</td>
<td>Example of Calculating the Farthest Address in an Unidirectional Loop</td>
<td>78</td>
</tr>
</tbody>
</table>
28. Single Processing System Failure  
Using the Successful Forward Technique .......... 202
There have been major advances in the sophistication of computer networks since their first appearance in the early 1960s. A good body of knowledge has been developed in the decade of the 60s on the design and implementation of centralized computer networks, where many (perhaps hundreds or thousands) of terminals are connected to a single computer system. Many techniques for reducing communication costs, for modeling system behavior, and for solving communication software problems have appeared during that time.

With the diminishing cost of processing [ROB74], the end of the 60s marks the beginning of a new era: computer communication networks. In short, this term
(computer communication networks) characterizes both communication using computers and communication among computers [MET73]. The technology of packet-switching [ROB78], as exemplified in many networks (ARPANET [ROB70, McQ77], CYCLADES [POU73]), shows that it is feasible (i.e., cost/effective) to interconnect many computer systems scattered over a wide geographical area. The success of this technology can be best demonstrated by its wide acceptance: many governments (e.g., Canada, France, England, Germany, Japan, etc.) and private companies (e.g., Telenet in the US) are implementing (and some have already implemented) such technology in order to offer the public computer communication services, much like electric or telephone services [KEL78].

In the research area, computer communication using packet-switching has also advanced during this decade in many directions: packet-switching systems using satellite channels in order to exploit its advantages have been implemented [JAC78]; interconnection among packet-switching networks and of packet-switching networks with networks of different technology have been achieved [CER78]; voice communication under packet-switching networks has been proven feasible [COH78]; and ground mobile networks using packet-switching and broadcast radio channels have been tested [KAH78].
Parallel to these developments, mostly in long-haul computer communication networks, there have been significant advances in the so-called local-area networks [LIU78, CLA78]. Basically, by restricting the geographical coverage of the network, it is possible to find new methods for interconnecting computer systems with substantial savings in cost and gains in performance (i.e., efficiency and reliability). Several local area networks have been implemented [FAR73, MET76] and some others are under current implementation [LIU78, WOL78, CLA78]. The potential of these systems can be best characterized by the growing interest of industry in developing the necessary hardware/software support for the interconnection of "geographically close" computer systems [DON79].

A closer look at all the developments mentioned above shows that computer networks have indeed succeeded in communication resource-sharing (i.e., sharing effectively communication resources such as bandwidth of channels and those of the computers used for communication). However, the area of computer resource-sharing (i.e., sharing effectively computer resources such as hardware, programs, databases, etc.) in computer networks is still in its infancy [MAM77].

The basic form of computer resource-sharing achieved so far can be called explicit resource-sharing.
Basically, this means that computer networks simply provide the capability for accessing remote resources. Thus, there is some degree of resource-sharing since many users can access the same remote resources. Unfortunately, this type of sharing has to be directed "manually" by the users. Many of us can recall the frustrating experience of using "powerful" computer networks and not being able to fully utilize many remote resources because we just do not know the peculiarities of many computer systems (and resources) of the network.

As computer networks infiltrate more and more in many sectors of our society, there will be an increased need not only for developing easier ways of using the resources available on these networks, but also on finding new techniques for implicit computer resource-sharing, e.g., networks whose computer resources are automatically handled by the system. In order to achieve this new type of resource-sharing, we have to reexamine many of the conventional approaches to resource-sharing in single systems since in these new systems the resources are spread throughout many computer systems. The systems with both communication resource-sharing and implicit computer resource-sharing can be called distributed systems.
1.1 Distributed Systems

Unfortunately, the term "distributed systems" (and equally distributed processing, or distributed computing, or distributed data processing) means different things to different people. For example, it is common for some to call distributed systems those systems with "intelligent" terminals, or several I/O processors, or front-end processors, or several general-purpose processors, or several general-purpose computers (i.e., networks), etc. Considerable effort has been put into trying to define what a distributed system really means: two workshops in distributed processing [ECK78] held at Brown University in 1976 and 1977 attempted to define the meaning of this term!

The pioneer work in the Distributed Computer System (DCS) [FAR73] shows some of the basic ingredients of a distributed system:

1. it is conceptually a single general-purpose machine, although physically a set of computer systems interconnected by a communication system (a loop);
2. it is process-oriented and all communication is by process name (i.e., no process knows where any other process is at any given time); and
3. resource allocation and the file system are distributed
(i.e., their functions typically involve cooperation of several systems).

Even more ambitious goals were set up for another distributed system, namely the Distributed Loop Computer Network (DLCN) [LIU75]. As before, they show more aspects of the structure of a distributed system:

(1) its users see a single unified computer system, although physically there is a set of autonomous computer systems (maxi, mini, or micro’s) interconnected by a communication system (a loop);

(2) its interprocess communication is by name;

(3) it provides a mechanism for calling of remote programs by name;

(4) it has a dynamically multi-linked process control structure; and

(5) it intends to provide distributed resource management, a distributed file system, and a distributed database system.

Interestingly enough, one of the most cited definitions of distributed systems provided by Enslow [ENS78] resembles the previous conception of a distributed system. He used the term "distributed data processing
systems" and gave the following components as essential for these systems:

1. multiplicity of general-purpose resources including both physical and logical resources, that can be assigned to specific tasks on a dynamic basis;
2. a physical distribution of such resources interacting through a communication network;
3. a high-level operating system that unifies and integrates the control of the distributed resources;
4. system transparency, allowing services to be requested by name only; and
5. cooperative autonomy, characterizing the operation of all resources.

Another interesting view of distributed systems is given in [SAL78] and is characterized by its emphasis on decentralization. The distributed systems of interest, from this viewpoint, are collections of largely autonomous processing systems, which cooperate in a way serving to produce a coherent system. Autonomy is important because many administrative as well as technical decisions tend to be local (i.e., with a minimum of central control). Cooperation is achieved by developing appropriate interfaces among remote system components (e.g., application or systems
programs). A distributed system is then, more specifically, the result of integrating existing system components of networks by means of standard interfaces. Due to the unique characteristics of distributed systems (e.g., in terms of reliability), such interfaces are not merely "add-on" features to conventional system components; even the local system components have to slightly change to use the interfaces [REE78].

Unfortunately, the lack of "real" distributed systems makes it hard to evaluate the validity of any view. Nevertheless, there are some common characteristics of these views that we shall adopt in this dissertation as "our" view of distributed systems:

(1) Computer networks form the "basic hardware" of a distributed system. Consequently, in a distributed system there are several processing systems (called hosts in a network) interconnected by a communication system (called communication subnetwork in a network).

(2) The actual physical architecture of the communication system seems irrelevant. Whether the communication system is a 5-meter high-speed bit-serial cable or a set of interfaces arranged in a loop or a set of bit-serial buses interconnected by repeaters or even a long-haul packet-switched system, has very little impact on the
logical functions expected in a distributed system. (Of course in terms of performance this is important.)

(3) The name of the game is software! The basic difference between a computer network and a distributed system lies in their software. The cooperation among remote systems in a computer network is limited to some simple standard services such as file transfer, mail, remote terminal access, etc. The cooperation among remote systems in a distributed system is very elaborate since it involves an attempt to implement resource-sharing of multiple remote resources. This point can be characterized by saying that the hardware of a distributed system is loosely-coupled, whereas many remote software components are tightly-coupled.

The architecture of distributed systems is suitable for designing systems with high reliability, great performance, and enormous flexibility. (We say "suitable for" because it is not necessarily trivial to obtain these advantages.) Also, distributed systems present a more natural environment than centralized systems for the solution of many problems.

In summary, regardless of a particular view taken on distributed systems, most views reflect an intellectually challenging software task with great potential benefits and
yet within reach since it is based on existing technology. In this thesis we take the view that distributed systems are highly sophisticated (long-haul or local) computer networks, whose emphasis is on implicit computer resource-sharing.

1.2 Scope of our Work

Much of the current work on distributed systems deals with the design of algorithms which in some way are related to the management of multiple remote resources of a distributed system. This trend can be easily seen by the proliferation of new algorithms for distributed concurrency control [JOH75, MUL75, THO79, ELL77, BAD78, BER78, ELL78, LAN78, REE78, S1078], distributed query processing [WON77, EPS78], distributed deadlock detection [CHA74, MAH76, GOL77, PEE77, MEN79a], distributed resource allocation [ALS76, MEN79b], and so forth. These algorithms are complex because they typically handle messages, asynchronous and concurrent events, and most importantly system failures. We are still far away from developing a coherent methodology for the design, implementation, and evaluation of these algorithms.

This thesis is not about new algorithms of the kind mentioned above, but rather on the provision of mechanisms that effectively support the implementation of these types of algorithms. The mechanisms of interest are:
interprocess communication and process synchronization. The relationship between these two problems (i.e., interprocess communication and process synchronization) in distributed systems is somewhat similar to their relationship in centralized systems: both mechanisms are typically found at the heart of system programs since they are essential for building other software.

1.2.1 The Problem of Interprocess Communication

Interprocess Communication (IPC) is the mechanism used for exchanging information between (among) processes. We use the word "process" in a very general sense: a program in execution [SAL66], or a unit of work that competes for resources. Thinking of a system as a set of processes constitutes a useful level of abstraction. Moreover, it is a very uniform view of computer system resources: processes can compete for resources, processes can be the managers of resources, and processes can even be the resources themselves. The history of IPC mechanisms helps to clarify our view of IPC for distributed systems (see Table I).

In the 60s, our main idea of computer systems was confined to single computer systems (or centralized
<table>
<thead>
<tr>
<th></th>
<th>Centralized IPCs</th>
<th>Network IPCs</th>
<th>Distributed System IPCs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>Shared Memory</td>
<td>Non-shared Memory</td>
<td>Non-shared Memory</td>
</tr>
<tr>
<td></td>
<td>Main memory:</td>
<td>Communication System:</td>
<td>Communication System:</td>
</tr>
<tr>
<td></td>
<td>Access Delay</td>
<td>Delay</td>
<td>Delay</td>
</tr>
<tr>
<td></td>
<td>Bandwidth</td>
<td>Bandwidth</td>
<td>Bandwidth</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>None</td>
<td>Communication System:</td>
<td>Communication and Processing System Anomalies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Anomalies</td>
<td>Anomalies</td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
<td>Conventional:</td>
<td>Network Services:</td>
<td>Distributed Alg.:</td>
</tr>
<tr>
<td></td>
<td>Cooperating</td>
<td>Two Remote Cooperating</td>
<td>N Remote Cooperating Processes</td>
</tr>
<tr>
<td></td>
<td>Centralized</td>
<td>Processes</td>
<td>Processes</td>
</tr>
<tr>
<td></td>
<td>Processes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Software Supported</strong></td>
<td>MULTICS IPC [SAL66]</td>
<td>TCP [CER74]</td>
<td>N-P PROTOCOLS</td>
</tr>
<tr>
<td></td>
<td>UNIX IPC [RIT74]</td>
<td>TP [ZIM75]</td>
<td></td>
</tr>
</tbody>
</table>
systems). Regardless of the number of central processors, I/O channels and memory banks, the design of IPC mechanisms was based on the assumptions that (1) there was a shared main memory, (2) the processes coexisted in the same machine, and (3) failures in the system were so catastrophic that the mechanism did not have to deal with them. Under these architectures, the delay incurred by IPC was mostly constrained by the delay of accessing the main memory, and the achievable throughput of the IPC depended mainly on the bandwidth of main memory. Typically, there was one actual system code which implemented IPC. The number of examples of these centralized IPCs (as called in [MET73]) is very large, since most systems have some kind of centralized IPC.

The advent of computer communication networks in the late 60s and early 70s brought about new problems in the design of IPC mechanisms. The source of such problems was that the processes did not coexist in the same system, i.e., they could be remote. This meant that two processes could be executing on different machines of perhaps dissimilar architectures and incompatible conventions for identifying the processes. Most importantly, such "remoteness" implied some kind of communication system between the processes. Consequently, there was a need for explicit message exchange (no more shared memory!) and for having actual IPC code at every site (no more a one-site IPC implementation!). The
problem of knowing the process destination "ID" became non-trivial. The delay and throughput of the IPC facility became highly dependent on the architecture of the communication system as opposed to the architecture of main memory.

Most of the issues in these IPC mechanisms (also called Host/Host protocols [CAR70], Process-to-Process protocols [CER74, SUN75], or Transport protocols [ZIM75]) dealt with the problem of unreliability introduced by the communication system.

With the growing interest in distributed systems since the mid 70s, new algorithms for multiple remote resource management have been continuously emerging. In general, these algorithms are a set of remote cooperating processes performing some function on remote (different or duplicated) resources. The question of interest is then: what are the new communication requirements of the processes of such algorithms? or what constitutes a suitable IPC facility for systems where it is common to have this kind of algorithms?

In addition to exchanging single-destination messages, many of these algorithms require at times the exchange of multi-destination messages. One reason for this requirement is to exploit the inherent parallelism in a distributed system. The simplest way for designing such a
multi-destination IPC mechanism is to add some logic "on top of" a point-to-point IPC. In this case, this new logic sends a multi-destination message by invoking the point-to-point IPC as many times as destinations in the message. Unfortunately, this method has serious disadvantages. First, the system has to exchange more messages than necessary, increasing the load and message delay of the system. Second, each pairwise exchange consumes independently its own resources (e.g., control information needed for reliability), when in fact the exchanges are related. In general, the first disadvantage is bad enough to avoid this kind of method. Therefore, appropriate multi-destination routing algorithms are important for the performance of multi-destination IPCs. Our first objective will be to study and propose solutions to this type of routing in several distributed system architectures.

Routing, however, is only one aspect of the problem. In particular, routing simply directs efficiently a multi-destination message to its destinations; however, this mechanism does not guarantee that a multi-destination message arrives to every destination, or that every destination only receives one copy of the message, discards damaged messages, etc. In other words, we still need (multi-destination) protocols "on top of"
(multi-destination) routing, if some levels of reliability are desired. Thus, our second objective will be to design multi-destination protocols with different degrees of reliability.

Having a very efficient software tool that is very difficult to use is of little value to an overall system design. This concern has stimulated many researchers to study structured programming, structured programming languages, etc. Thus, our third objective will be to design a mechanism by which (system or application) programmers can use interprocess communication protocols for distributed systems in a very natural and structured way.

1.2.2 The Problem of Process Synchronization

The scope of our previous problem of interprocess communication is simply "message passing" among a set of remote processes. The next level of interest is concerned with the nature of the computations performed by a set of remote processes. In particular, we want to study general synchronization mechanisms to be used by these computations in distributed systems. To clarify the significance of the problem and our level of interest, we start by characterizing the concept of "cooperation" among a set of remote processes.
A set of processes "cooperate" in a system for several reasons. First, we may have a set of independent user processes performing unrelated computations. Very often, for reasons of efficiency, these processes implicitly share system resources (e.g., processes may be unaware that they are actually sharing the only processor in a centralized system, or processes may be unaware of which processing system actually runs them in a distributed system where the load is automatically balanced). Thus, from the point of view of resources, the processes cooperate since they can share the same resources. Second, we may have a set of independent user processes performing some related computations. In these cases, the processes may explicitly share resources. The classical example is a set of user processes sharing a database. Finally, we may have a single user computation which intentionally has related processes. In this case the cooperation occurs within the user's domain (e.g., a user computation that finds the maximum element of an array by creating two concurrent tasks, each one to find the maximum of half the array, and then combining appropriately the individual solutions).

From above, we can see that our concept of cooperation encompasses the notion of concurrency. Most importantly, cooperation implies some kind of relationships among "events" generated at the different processes. In the
case of shared resources, the events are the execution of operations dealing with the shared resources. For example, if we have "n" processes sharing a stack, the events are the execution of "pop" and "push" operations generated at the processes. Even when there is not a very obvious resource being shared, as in our example of computing the maximum element of an array, there can still be relationships among events, e.g., to find the maximum of all the array we need that both events, finding the maximum of the first half and finding the maximum of the second half, finish.

We can relate this behavior of concurrent processes with the notion of program specification. A specification is a formal statement of what a program is intended to do [LIS77]. The significance of specifications is that they "measure" the correctness of a program: a program is said to be correct when it meets its specifications. In the case of concurrent programs (i.e., programs with components that can run concurrently) we can say that their synchronization specifications are those specifications describing the relationships among events generated at different concurrent components (program processes). A synchronization problem arises when events of interest in a computation are such that the model of computation does not guarantee to satisfy their relationships. In this case we need to augment somehow the
computation in order to meet the synchronization specifications. To this end, we typically use a synchronization model. Thus synchronization models are merely tools which can be used (or misused) by programmers in order to solve synchronization problems.

So far, we have not mentioned whether the underlying model of computation is centralized or distributed. The reason is that synchronization problems arise in both types of systems: relating order of events is a property of the logic of a computation regardless of whether the computation is realized by a set of processes that cooperate in a single system (centralized system) or from remote systems (distributed system).

Traditional general synchronization models such as Semaphores [DIJ68] and Monitors [HOA74] rely on the existence of a global environment. In short this means that concurrent processes can easily (i.e., it is fast and reliable to) access a common shared memory. Unfortunately, in a distributed system each remote process executes in its own local environment, making the notion of global environment non-existent. Our first objective will be to study how difficult it is to modify existing synchronization models in centralized systems for use in distributed systems. The motivation for approaching the problem in this manner is twofold: (i) it illustrates key features of a
"good" solution, and (2) the structure of some problems is such that minor modifications to traditional models suffice for a solution.

Unlike most synchronization models, the Eventcounts and Sequencer Model [REE79] is suitable for implementing in some distributed systems. Unfortunately, the implementation suggested in [KAN77] is not general enough and lacks important reliability features. Thus, our second objective will be to generalize such a "distributed synchronization model" and to add robustness to its implementation.

1.3 Related Work, Contributions, and Main Results

The problem of interprocess communication protocols for supporting multi-destination messages has not been attacked before. The most related work appeared in [DAL78], where several multi-destination routing algorithms for packet-switching networks were proposed and the problem of reliable multi-destination protocols was merely suggested as an area of future research. Thus, our work in multi-destination interprocess communication protocols constitutes a new contribution to the area of computer networks and distributed systems.
The main result in this area is the design of several levels of reliable IPC protocols: (1) unreliable level, where the protocol implements addressing and chooses the best multi-destination routing algorithm; (2) best-effort-to-deliver level, where the protocol attempts to deliver the message to as many destinations as possible, filtering duplicates and out of order messages at each destination; and (3) guarantee-to-deliver level, where the protocol ensures that every destination will eventually receive the message.

As mentioned before, multi-destination protocols are related to multi-destination routing. This type of routing has received attention only recently in long-haul packet-switching architectures [DAL77, DAL78, McO78]; however, no work has been done for local architectures. Thus, we have attacked the same problem for local architectures: single-bus, loops, and multiple-busses. Our solutions in this area also constitute a new contribution.

The main result in this area is the design of "imprecise partial-broadcast routing algorithms using partial-broadcast transmission". In short, we take advantage of the partial broadcast transmission property of some local architectures to improve the performance of the two other obvious algorithms: sending separately single-addressed messages and sending one full-broadcast
message.

The problem of how programmers should "see" interprocess communication has been a matter of controversy [SCH75]. Recently, new ideas on distributed programming have started to emerge and the issue of how programmers should best interact with IPC arises naturally. Basically, there have been two approaches to this problem: programmers use send/receive primitives [AMB77, KAH77, FEL79] and programmers use call/return primitives [WHI77, HAN78]. We rejected both types of primitives on the basis of lack of structure (first approach), difficulty for modeling several communication disciplines (second approach), and unrealistic assumptions (both approaches). Thus, our contribution consists of introducing a new communication language feature, called "remote-operation", as an alternative to the previous ones. We show that many communication disciplines can be best modeled by using this paradigm.

The problem of synchronization in distributed systems started to receive attention only in the last couple of years. Basically works in this area fall into two major categories: (1) specialized solutions to hard problems, and (2) synchronization models for simple problems. The best examples of the first category are solutions to distributed concurrency control problems in distributed database systems. Our main interest, however, corresponds to works
in the second category. Our main contribution in this area is the generalization of the implementation of the Eventcounts and Sequencers Model, adding robust features to deal with communication and processing system failures.

1.4 Organization of The Dissertation

In Chapter 2 we introduce basic concepts on distributed system software. Several software components are explained and the main issues in their design are discussed. In the last part of this chapter, we "assemble" these components to form a complete architecture of a distributed system. This chapter serves as introductory material for those who are not familiar with the typical problems in distributed systems.

Chapter 3 deals with the topic of multi-destination routing. We concentrate mainly on local communication system architectures, although we also mention the differences between this problem in long-haul architectures and local architectures. We develop several routing algorithms and compare their performance under different performance measures. In Chapter 4 we design several multi-destination IPC protocols. We explain in detail the mechanisms used by each type of protocol.

Chapter 5 addresses the question of how best to
use IPC protocols. We introduce the domain of distributed programming and show how the concept of remote-operation can handle many communication disciplines. In Chapter 6 we move into the problem of process synchronization. This chapter explains in detail the new features of distributed synchronization and one traditional synchronization model is modified for use in distributed systems. Chapter 7 deals with distributed synchronization models. We generalize a previously suggested implementation for a low-level model and propose the basic ideas of a new high-level model. Finally in Chapter 8 some conclusions of this work are drawn and suggestions for further research are given.
THE ARCHITECTURE OF DISTRIBUTED SYSTEMS

The purpose of this chapter is to present an overall software architecture of a distributed system. The motivation for developing such an architecture is twofold: (1) although there is currently a substantial amount of work in different areas of distributed systems, very little has been said about "how to put all pieces together", and (2) a good understanding of the typical issues in distributed system software is necessary to design basic mechanisms (such as interprocess communication and synchronization) for such systems.

This chapter is organized as follows. In the first three sections we introduce basic concepts of distributed systems such as distributed algorithms, distributed control, and reliability. In Section 4 we define and discuss several software components of...
distributed systems. We call these components "distributed software". Finally, in Section 5 we "assemble together" all these components into a distributed system virtual machine.

2.1 Distributed Algorithms

To start our discussion of distributed systems, we shall explore the concept of the so-called "distributed algorithms." Interestingly enough, there is no precise definition of what this term really means.

For many years we have become familiar with sequential and parallel algorithms. In a sequential algorithm each step of the algorithm is carried out (i.e., executed) one at a time. In contrast, several steps of a parallel algorithm can be carried out simultaneously.

Implicit to the notion of an algorithm is the notion of some "entity" which carries out the steps. In fact, the ambiguity of a step strongly depends on what the interpreting entity is, e.g., a step such as "01011101" can be very ambiguous for a person and very clear for some CPUs. We argue below that some properties of the interpreting entities make an algorithm become distributed.

The notions of centralization and distribution are closely related to the notion of a "physical boundary." An interpreting entity and a set of steps are within the same
physical boundary if the interpreting entity can potentially execute any step of the set. A centralized algorithm has only one physical boundary. Any of the algorithms mentioned above can be a centralized algorithm. On the other hand, in a distributed algorithm there are several physical boundaries (i.e., some interpreting entities cannot execute some steps because they "belong" to some other interpreting entities). Most importantly, the uniqueness of a distributed algorithm (otherwise there would be as many unrelated centralized algorithms as physical boundaries) is achieved by explicit exchange of information among physical boundaries. Any sequential or parallel algorithm can also be a distributed algorithm!

To make more precise the above concepts we can take programs as a particular way of describing algorithms. In this case the steps are instructions and the interpreting entities are processors. (If the instructions are high-level language statements one can think of processors as virtual processors of the language.) The notion of physical boundary represents the familiar concept of closeness between instructions (stored in some memory) and processors. The simultaneous execution of steps in a parallel algorithm requires the existence of several processors (yielding a parallel program). Other programs may or may not actually execute in parallel; however, they
describe a potential parallelism. These programs are called concurrent programs. All programs running under architectures where processors can execute any instruction are centralized programs.

In contrast, a distributed program has several groups of instructions (to be called "program components"), each group being only executed by its associated processor(s). The exchange of information among groups is realized by a mechanism which can "move" information, e.g., a communication system. This mechanism is extremely important because it introduces the notions of space (i.e., information becomes distant when it cannot be accessed directly) and time (i.e., information cannot be moved instantaneously). Two program components separated by a communication system are said to be remote to each other.

A sequential program can be distributed if at some point in the computation the execution continues in a different physical boundary. For example, imagine a sequential program calling from one site a subroutine located at a different site. The most interesting distributed algorithms are, however, those which exhibit concurrency, i.e., concurrency among program components.

Some distributed algorithms have the property that every remote component executes the same logic. These algorithms can be called "homogeneous". In some cases,
robust distributed programs exhibit such property: by being "equally" important, some program components can still perform a computation upon failure of some other program components. Notice, however, that our notion of distributed algorithm is not related to robustness.

2.2 Distributed Control

The notion of control of an algorithm can be abstracted into the notion of "state of a computation" (i.e., both terms refer to the same concept). Since a computation is the dynamic counterpart of a program, control of a program is a dynamic notion. This state or control specifies the "progress" of a computation at a given point in time. For example, let us take a program which generates ten prime numbers and computes a function on them; the function is such that we first have to generate the numbers. Initially (before any prime number has been calculated) the progress contains very little information, e.g., few initial values. Somewhere in the middle of computing the numbers, its progress contains the information of which numbers have been generated so far. Just before finishing, the entire progress is simply a number, e.g., the result of computing the function.

In (centralized) concurrent and parallel programs,
It may be even more difficult to specify what information precisely constitutes the progress of a computation. Typically, if these programs modify shared data structures, it is necessary to include the state of the data structure as part of the progress of the computation. We define centralized control as the control of a centralized computation, i.e., the control of a computation that only runs in a single processing system.

The control of a distributed computation can be thought of as a "vector" of states, each dimension representing the state of the program component executing at a processing system. Another important notion in the control of a distributed computation describes how the individual states (the dimensions) relate to each other. These relationships define the "overall" progress.

Consider the case of a program which calls a remote subroutine in the middle of the computation (this is a sequential distributed program). The overall progress of the program resides in a single site (the main program site) before and after the subroutine is called. However, when the subroutine is being remotely executed, the control is somehow "split". Most importantly, the overall progress depends on both individual progresses.

A different situation occurs in a distributed program where execution starts at one site to compute some
partial result, and then the partial result is passed to another site which in turn computes the final result. In this case the control "moves" from one site to another. The overall progress is equal to individual progresses at different times.

A distributed algorithm (or program) has centralized control when the overall progress of the algorithm always resides in a single site. If a distributed algorithm does not have centralized control we say that it has distributed control. In our example of a program calling a remote subroutine, the program is distributed (locus of execution crosses site's boundaries) and its control is distributed (the overall progress does not reside at all times in a single site). Unfortunately, this program can be very unreliable (e.g., when it needs both systems up for continuing progress). If we could call simultaneously two identical subroutines located at different sites, and simply use the first result, the distributed program has the property that if the progress of the computation at one subroutine site is destroyed (or is unavailable) the overall progress can still be preserved (i.e., it is more reliable than in the previous case). In the example of "moving" the control across sites, the reliability of the algorithm is also interesting: If we have n sites and the control moves sequentially from site to site, it is only necessary to have
one site up (the one executing) at a time in order to continue overall progress.

A distributed program can have distributed control and still be very unreliable. Similarly, a distributed program can have centralized control and still be very reliable! As a rule, however, distributed programs with distributed control are very reliable. The important point here is that distributed control and reliability do not denote the same concept.

2.3 Reliability

In general, "reliable software" implies its correctness and robustness [YEH76]. An algorithm is correct if it meets its functional specifications. An algorithm is robust (fault-tolerant) if it can continue to deliver some level of service even when faced with a hostile environment (e.g., bad input data, or hardware failures). A "reliable distributed algorithm" should be correct and robust. Correctness is a property that any algorithm should have regardless of whether it is centralized or distributed.

Most of the work in "reliable centralized algorithms" deals with software validation (e.g., correctness, testing). Robustness of centralized algorithms has received relatively little attention (see [RAN78] and
(LEV77) for a few works dealing with robustness of these algorithms). By contrast, most of the work in "reliable distributed algorithms" deals with robustness and very little attention has been paid to their correctness (see [LAM78a] for an example of a formal proof of correctness of a distributed algorithm). Unlike the types of failures considered in robust centralized algorithms (e.g., bad input data, overflows, addressing errors, etc.), the types of failures considered in robust distributed algorithms are more catastrophic, e.g., complete main memory loss of a processing system, or unreachability of a processing system.

An interesting property of "reliable distributed algorithms" is that a major part of their functional specifications include the behavior of the algorithms under system's failures.

As pointed out in [SV079], it is difficult to define quantitative measures of reliability for distributed algorithms, especially when such reliability affects the integrity of data. The alternative is to characterize the types of failures of a distributed system and check whether a reliable distributed algorithm can tolerate such failures.

2.4 Distributed Software

So far we have introduced several basic concepts
in distributed systems; however, we have not mentioned any specific algorithms or functions to be performed by these systems. At first, it seems that one can simply take any conventional software function (such as processor management, memory management, database management, etc) and a new problem in distributed systems arises. Unfortunately (fortunately?), this is not true since the performance of the system can be severely degraded. Performance is a complex issue that encompasses reliability and efficiency. Many times the gains in reliability can be defeated by the increase in overhead. Ideally, distributed system software should be efficient and reliable.

We call "distributed software" the set of programs (algorithms) used for the management of remote resources. The nature of the resources determines the type of distributed software. Two major types of distributed software can be identified: distributed operating system software and distributed database system software.

2.4.1 Distributed Operating Systems

An operating system is simply the set of programs that manage resources such as processors, processes, main memory, secondary memory, files, and so forth. In a centralized system all resources coexist in the same system.
Thus, the operating system itself coexists with the resources in the same system. In a distributed system resources can be spread throughout several processing systems so that the management of such resources can also be distributed.

Clearly, the existence of a physically distributed system implies that resources are distributed. However, this does not mean that the management of all resources should be distributed. For example, allocating pages of main memory tends to be an entirely local function, whereas retrieving a file can be a distributed function (if files can be located at any site). In general, the management of resources becomes distributed either because of external requirements (e.g., users want their files at their local sites but they still want to access remote files), or because of purely internal performance considerations (e.g., the system balances its load among several sites to increase throughput).

In order to give examples of distributed operating system functions, it is useful to look at operating systems from two points of view: as resource managers, and as virtual machines.

In the first, we take operating system resources and determine whether these resources can be managed in a distributed fashion. The following are some examples:
(1) **Distributed Job Management.** In this case the resources are processing systems and the function is to allocate such resources to jobs. This function is equivalent to a static "load balancing" problem, i.e., allocate in advance processors of processing systems to jobs. A more difficult problem is to assign processing systems dynamically.

(2) **Distributed File System.** In this case the resources are files and the function is to support file system operations on a collection of distributed files. The typical issues in the design of this function are: how to locate files (catalog problem), where to allocate files (allocation problem), when to move files (migration problem), etc.

In the second view, a programmer "sees" at any level an augmented machine (level 0 is simply the bare hardware) that provides an environment for building further machines (e.g., upper levels). A distributed operating system should have levels that support the implementation of distributed resource management. This view explains the role of interprocess communication and synchronization in a distributed operating system.
2.4.2 Distributed Database Systems

The most well-known examples of distributed software arise in the design of distributed database systems (DDBs). In a DDB the actual data are spread throughout different processing systems in order to increase accessibility and reliability, to improve response time and throughput, to satisfy user needs, and so forth. Accessibility implies that almost every "portion" of data in a DDB has to be duplicated at least at some other processing system.

Clearly, if a DDB is a read-only database, it is advantageous to duplicate the entire data at many (perhaps all) processing systems. However, even with this type of database, total duplication may not be possible if storage cost is too high. As a result, the designer may only duplicate parts of the entire database at different processing systems. In this case, it is possible that some queries may have to collect data from different processing systems, thereby introducing the problem of Distributed Query Processing (DQP).

If a "distributed query" (that is a query whose answer set is spread throughout several sites) refers to large quantities of data, the main concern of a distributed algorithm for processing such queries is to minimize
movements of large quantities of data across site boundaries. Some work has been done along these lines [EPS78, WON77]. A more complicated problem (that has not been attacked yet) consists of adding reliable features to a distributed query processing algorithm.

Another dimension of the DDB problem is introduced when updates to multiple copies of data are possible. In this case, it is necessary to ensure that an update always "sees" a consistent version of the data. On one hand, copies of data should indeed have identical values. (Of course at a given point in time during the update copies may have different values; however, if no other updates are received by the system after the update the copies should converge to identical values.) This type of consistency is usually called "mutual consistency". On the other hand, every copy should perform the updates in the same (equivalent) order as any other copy. This corresponds to the so-called "internal consistency" (equally required in centralized systems). In order to maintain both types of consistency a new mechanism has to be introduced: Distributed Concurrency Control (DCC).

Many solutions have been proposed for this problem. These solutions vary depending on the assumptions and performance goals of the system. For example, some solutions rely heavily on the assumptions that update
transactions can be classified in advance [BER78], or that updates never have to read data [JOH75], or that the database is entirely duplicated at every node [ELL77, TH079], etc. With respect to performance goals some solutions tend to be very reliable sacrificing either response time [TH079] or concurrency [ALS76, MEN79b].

2.5 A Distributed System Virtual Machine

The overall software structure of a distributed system can be described using the concept of virtual machines (VMs). Both centralized and distributed systems can be viewed as powerful VMs. Unlike centralized virtual machines, a distributed virtual machine (DSVM) is composed of a Communication Virtual Machine (CVM) and several Remote Virtual Machines (RVMs).

2.5.1 Communication Virtual Machine

The purpose of the CVM is to perform all the message-passing functions demanded by the RVMs. The actual physical realization of this machine depends on the combination of many factors: cost, geographical dispersion of RVMs, reliability requirements, etc. Most of the design issues of this machine are fairly well understood since to a
great extent there is not much difference between the so-called communication subnetwork for computer networks and our CVM for distributed systems. The differences will become clear in subsequent chapters.

Typically, a CVM consists of a complex of hardware and software. A basic hardware component is the circuitry and/or physical medium through which signals travel. This is usually referred to as the transmission medium. Other hardware components may range in their complexity from simple interfaces to sophisticated computer systems used for communication; we shall call them communication processing nodes. The important characteristic of these latter components is that they are physically apart.

The most important software components of a CVM are protocols, i.e., programs which implement the conventions and rules by which remote components can exchange information. Due to the variety and complexity of CVM architectures, there are many types of protocols and, within a CVM, there is typically a hierarchy of protocols.

Other important software components for many CVMs are routing algorithms, i.e., programs which decide how best to direct messages within the components of the CVM. With all this software and hardware, a CVM resembles a special-purpose DSVM for message-passing. This last point is interesting because there is great resemblance between
the design of routing algorithms and the design of many algorithms for distributed systems: many routing algorithms are distributed algorithms with distributed control; the implementation becomes a distributed program with many program components (one per node) exchanging messages and trying to cope with communication and node failures, i.e., it is very reliable.

Although there are many types of CVMs, current technology has created an important division of architectures: long-haul CVMs and local CVMs. (In turn, they make the difference between long-haul and local distributed systems.) We shall briefly mention below some protocol and routing characteristics of these architectures which will help our discussion in the next two chapters.

**Long-haul CVMs.** The most popular architecture for long-haul CVMs is packet-switching. These machines are complex and expensive; however, this is natural since they satisfy the requirement of extending to large geographical areas. A typical protocol architecture for a packet-switching CVM consists of three major levels of protocols: process communication protocols, access communication protocols, and node-to-node communication protocols.

The highest level corresponds to process
communication protocols (i.e., the producers and consumers of messages are processes). Typical examples of these protocols are Host-to-Host [CAR70], TCP [CER74], and Transport Protocol [ZIM75]. It is convenient to have several types of process communication protocols since upper levels of software may have different communication requirements. For example, a file transfer application needs a process communication protocol that enforces sequencing of messages, detects duplicate messages, and uses mechanisms (such as buffering) that maximizes throughput. An application such as inquiry/response needs a process communication protocol that detects duplicates but without the complexity of enforcing sequencing (the application itself may enforce sequencing if no message is sent unless the answer of the previous message is received). Finally, an application such as voice communication needs a process communication protocol with some timing features (relative timing of portions of speech is important to reconstruction of "chunks" of speech at a receiver) and some loss of messages can be tolerated (speech information is highly redundant). We shall argue in Chapter 4 that different interprocess communication protocols are also needed to support distributed system software.

The levels of access communication and node-to-node communication are similar in that both are
typically "link control protocols" (i.e., the producers and consumers of messages are fixed units of hardware). Examples of these protocols are Host/IMP [McQ77] and IMP-to-IMP [McQ77] protocols in the ARPANET. The access communication protocol "enters" messages into the communication system; the node-to-node protocol in conjunction with routing "moves" messages throughout the communication system.

Local CVMs. There are many types of local communication system architectures (we shall introduce them in the next chapter). Unlike the long-haul architecture, the protocol architecture of these systems only consists of two levels: process communication protocols and access communication protocols. Process communication protocols are similar to the ones used in the long-haul architectures (although some simplifications are possible since local environments are less "hostile" than long-haul environments). Since messages are not stored and forwarded in these architectures, node-to-node protocols do not exist. Access communication protocols become extremely important since the transmission medium is typically shared by all communication processing nodes. Examples of these access communication protocols are the so-called "transmission mechanisms" in loop architectures [LIU78, CLA78] and many
types of Carrier Sense Multiple Access protocols in single-bus and multiple-bus architectures [DON79, JEN78, MET76].

2.5.2 Remote Virtual Machines

The purpose of RVMs is to carry out the actual data processing algorithms. One can think of a computer network as a DSVM whose RVMs have few distributed functions. In fact, these functions are only services such as file transfer, remote terminal access, mail, etc. Clearly, these are distributed functions since a service requires cooperation (typically between two programs) in order to perform its duty. Thus, in these kinds of RVMs the distributed functions are very simple.

When more sophisticated functions are distributed, the system is no longer a computer network but rather a distributed system. Typically, the structure of a RVM has a set of local and distributed functions. These correspond to Local Virtual Machines (LVMs) and Cooperating Virtual Machines (COVMs) within RVMs (see Figure 1).

The functions performed at LVMs are those which do not require remote cooperation either because of performance considerations or because of their own nature. Examples of these are: main memory allocation, access to secondary
RVM: Remote Virtual Machine
CVM: Communication Virtual Machine
LVM: Local Virtual Machine
COVM: Cooperating Virtual Machine

Figure 1. Virtual Machines of a Distributed System

Virtual Machine
storage, and system or application programs that belong exclusively to a local system.

Certainly, the most interesting component of a RVM is the COVM. Unfortunately, very little is known about elegant and coherent designs of such machines. Some systems have a COVM that performs one or few functions. For example, the RSEXEC system [THO73] in the ARPANET supports a multihost file system. In this case the COVM consists of special processes at each host (called RSEXEC and RSSER) that interpret commands and naming conventions, and most importantly, can cooperate with remote processes. Another example is the National Software Works (NSW) [FOR78] in the ARPANET which provides a tool for software development. In this case each COVM consists of two processes: the Works Manager which validates and allocates (local or remote) resources and the File Package which provides a multihost file system. Both types of processes may cooperate with remote processes. The other two types of processes, the Foreman (interface for resources) and the Front End (interface for users), can be considered part of LVMS since they do not involve cooperation with remote processes.

Figure 2 shows, what we believe is, a cleaner and more coherent method for designing COVMs. The key to this approach is the development of a distributed programming language, which helps to implement algorithms for different
Distributed Software Level

Distributed Algorithms

Distributed Programming Language

Low-Level Mechanisms
Distributed IPC
Distributed Synchr.
Fault- Naming
mechanisms
mechanisms
mechanisms
Tolerance mecha.
mechanisms

Figure 2. A Distributed Programming Language:
A tool for Designing Distributed Software
distributed functions. (We shall explain in more detail some aspects of such a language in Chapter 5.) This language provides a single common tool when many apparently different mechanisms such as interprocess communication, synchronization, protection, naming, and reliability, are unified.

This idea is not new since high-level languages have been used for the design and implementation of centralized system software. However, for distributed software, each one of the mechanisms (interprocess communication, synchronization, etc.) is different from its counterpart in centralized systems, thereby making their integration into a single language a very challenging problem.

Given such a high-level language as a tool, it is possible to implement many distributed functions such as distributed deadlock detection algorithms, distributed concurrency control algorithms, distributed query processing algorithms, etc. Another important role of such a distributed programming language is to interface with distributed applications. Unlike systems applications (like distributed query processing), user applications are heavily problem-solving oriented. It is expected that some properties of a distributed system can be exploited and lower-level problems can be largely ignored. As an example,
consider the case of "knowledge-based systems" often used in artificial intelligence problems. Sometimes the "knowledge" may be geographically dispersed, or in some others the nature of the problem-solving task can be better modeled as a set of loosely-coupled modules solving problems based on partial information. These problems can be more naturally implemented in distributed systems.

2.6 Summary

In this chapter we have introduced basic concepts related to distributed system software. We made a distinction among three widely used (but often misused) concepts: distributed algorithms, distributed control, and reliability. An algorithm is distributed when its associated computation executes in more than one site. Distributed control, on the other hand, tells us about the overall progress of the computation. If such progress is not always in the same site, the control of the algorithm is distributed. Finally, reliability of a distributed algorithm describes the capability of a computation to "survive" communication and processing system failures.

We introduced the concept of distributed software. In short, distributed software describes the set of distributed algorithms used in distributed systems for the
management of remote resources. We gave examples of distributed software in the areas of operating systems (distributed operating systems) and databases (distributed database systems). In the last part of the chapter we discussed layers of software in a distributed system (a distributed system virtual machine).
In the previous chapter we discussed several functions performed by distributed systems. We learned that some of these functions deal with the management of multiple remote resources. In some cases remote resources are functionally equivalent, whereas in some other cases remote resources are identical (remotely replicated). Typically, in order to exploit the inherent parallelism in a distributed system, the functions that manage such remote resources generate messages directed simultaneously to several sites (several destinations). The purpose of such messages is irrelevant when they are viewed from the communication system level. Also, the multiplicity (i.e., the number of destinations) of those messages is beyond control of the communication system. Thus, it is necessary for communication systems to direct efficiently multi-destination messages to any number of destinations.

In this chapter we shall consider the problem of
multi-destination routing in several distributed system architectures. The architectures of interest (those that have become or are becoming very popular) are: long-haul packet-switching architectures; and local-area architectures such as single-bus, loops, and multiple-bus architectures. We shall concentrate mainly on local architectures since this problem has not been previously attacked for this type of architectures.

This chapter is organized in the following way. In Section 1 we introduce some terminology. Section 2 discusses very briefly the problem on long-haul architectures. Since several algorithms have been already proposed for this type of architecture, we simply emphasize those aspects that contrast sharply with the other type of architectures. In Sections 3, 4, and 5 several multi-destination routing algorithms for local architectures are explained. Suggestions for implementation of these algorithms are also given. Finally, Section 6 compares some multi-destination routing algorithms under different performance measures.

3.1 Definitions

A good characterization of routing and how it
relates to some other mechanisms is given in [SH078]:

"The name of a resource indicates what we seek, an address indicates where it is, and a route tells us how to get there."

Most of the work in routing algorithms deals with point-to-point or single-destination routing (i.e., how to direct a message from a source to one destination) over packet-switched networks. Our interest here is in multi-destination routing algorithms (i.e., how to direct a message from a source to several destinations). We shall call full-broadcast routing a particular case of multi-destination routing intended for all destinations and partial-broadcast routing a case of multi-destination routing intended for many (but not all) destinations.

Routing algorithms should be distinguished from transmission properties of communications systems. The type of transmission of a communication system indicates how many destinations "hear" a transmitted message ("hearing" a message means that a destination can potentially copy the message). If we have a line between two computer systems, the line is a communication system which uses point-to-point transmission. There are, however, some communication systems which use full-broadcast transmission (i.e., a message is always heard by all destinations) or partial-broadcast transmission (i.e., a message can be heard
by some but not all destinations). In these systems, point-to-point routing is typically implemented by letting a destination copy a message only when the address of the message matches the address of the destination; otherwise, although the destination hears the message, it does not copy the message.

In communication systems that use point-to-point transmission, "how to get there" (i.e., routing) involves mainly selecting point-to-point transmissions. In these architectures, "hearing" a message implies copying a message. In contrast, local communication architectures that use full-broadcast or partial-broadcast transmissions, the mechanism of "how to get there" involves mainly which destinations should "hear" the message and/or what should be done when "hearing" a message.

We next examine the problem of multi-destination routing in four popular communication system architectures.

3.2 Packet-switching Architectures

The architecture of interest corresponds to a long-haul packet-switched communication system, much like the one used in the ARPANET. For the design of multi-destination routing algorithms, the important characteristics of this architecture are: (1) distributed
topology with store-and-forward of messages, (2) point-to-point transmission between the nodes (the packet switches), (3) large computational power at the nodes, (4) limited bandwidth, and (5) message delay composed mainly of transmission and queueing delays at intermediate nodes. These characteristics contrast sharply with the characteristics of the subsequent (local) architectures.

Under these characteristics it is possible to design elaborate multi-destination routing algorithms. Characteristics (1) and (2) show why these algorithms are needed; (3) tells us that there is indeed "room" at intermediate nodes for implementing such routing; and (4) and (5) typically shape the performance of such algorithms.

As an example, let us consider the so-called "Multi-destination Addressing Algorithm" in [DAL78], or "Broadcasting Algorithm Using an N-bit Vector" in [McO78]. In these algorithms a multi-destination message has a bit map, each bit indicating whether the message should be sent to a destination. Also, the algorithms are built using point-to-point routing information, e.g., at each node there is always a "best route outgoing link" for every possible destination. The algorithms basically do the following: (i) upon arrival of a message to a node, a copy of the message is transmitted over each outgoing link that belongs to the set of "best route links" corresponding to the
destinations indicated in the bit map, and (2) each one of the copies is transmitted turning "off" the bits in the bit map corresponding to the destination for which the link was the "best route link".

Other algorithms can be found in [DAL77, McQ78, DAL78]. From the point of view of protocols, two important observations can be made about these algorithms:

(1) Multi-destination routing algorithms are not completely reliable. A multi-destination packet, in general, may arrive at only some of its intended destinations (e.g., if nodes fail at critical times). Thus, it is necessary to have reliable multi-destination IPC protocols in conjunction with multi-destination routing. (This is analogous to having reliable point-to-point IPC protocols "on top of" point-to-point routing, as suggested in [DAL78].)

(2) Due to qualitative considerations (e.g., modifying existing routing algorithms, complexity of implementing new algorithms) and quantitative trade-offs (e.g., efficiency, reliability), it seems unlikely that there will ever be a single multi-destination routing algorithm for handling single-destination messages as well as multi-destination messages with "few", "many", and all destinations. Thus, it is necessary in these
systems to choose among different routing algorithms.

3.3 Single-bus Architectures

These are local communication systems composed of several interfaces (each of which is potentially attached to a computer system) sharing a single bit-serial bus. Examples of these architectures are HXDP [JEN78] and a single-segment ETHERNET [MEI76]. In contrast with the previous architecture, the design of multi-destination routing algorithms has to consider the following system characteristics: (1) single-bus topology without store-and-forward of messages, (2) full-broadcast transmission medium, (3) not too much computational power at the interfaces, (4) high-bandwidth, and (5) message delay mainly composed of the access delay to the transmission medium.

With these characteristics, it is unnecessary to have multi-destination routing algorithms similar to the ones used for packet-switched architectures. Since messages are not stored at intermediate nodes, multi-destination routing is source-based, i.e., completely done at the source node. Clearly, full-broadcast routing is trivial in these architectures since they use full-broadcast transmission. For multi-destination messages of several (but not all)
destinations, there are basically two possibilities.

First, the message can be sent using separate single-addressed messages. This can be called *virtual partial-broadcast routing*. In general, this type of routing is acceptable when there are "few" destinations. There are some differences on the behavior of this algorithm between a packet-switching architecture and a single-bus (and in general, local) architecture: (1) each separate message (one for each destination) in a packet-switching network does not directly contend with messages from other systems while accessing the communication system, whereas in a single-bus architecture each separate message competes directly with messages from other systems while accessing the bus; (2) the multiple separate single-addressed messages in a packet-switching architecture can be simultaneously in transit (i.e., "pipelined"), whereas in a single-bus architecture there can only be one message at a time in the communication system. This suggests that this algorithm tends to degrade worse in a local architecture as the multiplicity (number of destinations) of the message increases.

Second, if the communication system has many interfaces and the number of destinations in a multi-destination message is very large (say the total number of destinations minus some threshold), it becomes
attractive to discard messages. In this case, rather than sending "many" (and the same) single-addressed message, the protocol can choose to use an *imprecise partial-broadcast routing* algorithm, where the message has to be discarded at some destinations.

Perhaps the optimal method for discarding is to have enough built-in hardware at the interfaces so that each interface only copies the message when the group of destinations (say a bit map of destinations) includes the interface address (interface bit position is "on"). Another method is to perform the previous checking logic after the message has been copied. In general, this latter method is more flexible and can be adapted to any existing system. The disadvantage, of course, is the unnecessary use of storage. Figures 3 and 4 show both routing strategies on single-bus architectures.

3.4 Loop Architectures

The communication system of these architectures consists of interfaces arranged in a loop. Typical examples of these architectures are DLCN [LIU78] and DCS [FAR73] with unidirectional loops, and DDLCN [WOL78] with two loops of different directions. For the design of multi-destination routing algorithms, the following characteristics are
SEND MESSAGE TO 3, 8, 11

Figure 3. Virtual Partial-broadcast Routing in Single-bus Architectures
SEND MESSAGE TO 2,3,4,5,7,8,11,12,13

Figure 4. Imprecise Partial-broadcast Routing Using Full-broadcast Transmission in Single-bus Architectures
important: (1) loop topology without storing messages, but with forwarding (i.e., check-and-forward), (2) full-broadcast or partial-broadcast transmission, depending on whether the transmitter or receiver removes the message, (3) limited computational power at the interface, (4) high-bandwidth, and (5) message delay composed mostly of access delay to the transmission medium.

The implementation of full-broadcast routing under these characteristics is again trivial. The algorithms for partial-broadcast routing depend on the address recognition mechanism at the interfaces. One addressing mechanism (e.g., the one used in DCS) uses an associative memory so that a message is copied by the interface if there is a match between the destination field of the incoming message and any of the stored values in the memory. If the destination bits of the message are stored at several associative memories, it is possible to send a message to several destinations with one fixed-length destination field. In this case, partial-broadcast routing is automatically achieved provided that the system has a flexible mechanism for handling the values of the memories. A property of this mechanism is that the destination bits can simply be names (not addresses), making completely transparent to the sender the location of its receiver(s). The main problem of this mechanism, however, is the
difficulty in building reliable exchange of messages (e.g., when the sender desires to selectively retransmit a multi-destination message that has only been acknowledged by some of the destinations, it is unable to do so because it cannot distinguish among receivers).

Another addressing recognition mechanism (e.g., in DLCN and DDLCN) simply compares the destination bits of an incoming message with a built-in destination address (the interface address). In this case, similar arguments can be used (as in the previous architecture) for implementing a virtual partial-broadcast routing and an imprecise partial-broadcast routing which uses full-broadcast transmission (see Figures 5 and 6).

A new imprecise partial-broadcast routing can also be designed for these architectures, if one takes advantage of the partial-broadcast transmissions inherent to loops. Basically, for architectures in which a destination node removes the message, it is possible with minimum changes to copy a message at every interface between a source and a destination. Intuitively, there are situations in which the number of destinations is not large enough to justify one full-broadcast transmission and many discards, and it is not small enough to justify sending separately single-addressed messages. For these cases, the final destination of a message can be adjusted to be the "farthest away" from the
SEND MESSAGE TO 5, 7, 10

Figure 5. Virtual Partial-broadcast Routing in an Unidirectional Loop
Figure 6. Imprecise Partial-broadcast Routing Using Full-broadcast Transmission in an Unidirectional Loop
source. By copying the message at each intermediate interface, the intended set of destinations get the message (plus perhaps some interfaces between source and the farthest destination, which are not part of the destination set). Figure 7 shows an example of this strategy.

The implementation of this new imprecise partial-broadcast in DLCN consists of the following parts:

(1) Given a set of destinations dest(1), ..., dest(j), where j > threshold and j < total # destinations - threshold, it is possible to calculate the farthest destination relative to a given Loop Interface Unit (LIU),

\[
\text{liuadd}^* \leftarrow \text{farest}(\text{dest}(1), ..., \text{dest}(j))
\]

(2) The message explicitly indicates that it is not a full-broadcast, nor point-to-point (for example, using two bits),

(3) The handling of each type of message at interfaces other than the source becomes:
Figure 7. Imprecise Partial-broadcast Routing Using Partial-broadcast Transmission in an Unidirectional Loop
if full_broadcast then copy
    forward
    exit

if imprecise_partial_broadcast then
    if myliuadd=liuadd* then copy
        exit
    else copy
        forward
        exit

if point_to_point then
    if myliuadd=liuadd then copy
        ack
        exit
    else forward
        exit

The above implementation only acknowledges single-addressed messages. It is undesirable to extend such "automatic acknowledgement" to each destination of a multi-destination message since too much traffic would be generated (at a very low level). A nice compromise can be made in the case of imprecise partial-broadcast by making only the farthest away destination to acknowledge the message.

In the case of DDLCN, where under normal operation there are two unidirectional loops of opposite directions, this new imprecise partial-broadcast is basically the same. However, since DDLCN has two directions, it is possible that a set of destinations generates two farthest addresses, each one on a different direction (see Figure 8). Finally, it is interesting to note that this new multi-destination routing can simulate full-broadcast routing in DDLCN by choosing two
Figure 8. Imprecise Partial-broadcast Routing Using Partial-broadcast Transmission in a Bidirectional Loop
farthest addresses (one for each direction) such that the entire loop is covered. This property can be used to consume equal share of bandwidth on each channel for full-broadcast messages.

3.5 Multiple-bus Architectures

The additional busses in these architectures are interconnected by the so-called "traffic-filtering repeaters" or "packet filters" [MET76]. These (unbuffered) repeaters simply pass messages from one bus to another. To a great extent, the characteristics of these architectures are similar to those of single-bus architectures. Thus, virtual partial-broadcast routing algorithms and imprecise partial-broadcast routing algorithms using full-broadcast transmission also apply here.

Repeaters can be used to avoid the situation that more interfaces than necessary "hear" a message. This can be accomplished, for example, in the following way. Let us assume that an interface address is composed of a segment number and an interface number. Then, it is possible to add at each repeater a small associative memory containing information on which segment belongs to each one of the subtrees defined by the repeater. Upon arrival of a message from one subtree, the repeater can quickly decide (by
checking the segment number of the destination field of the message) whether the message should be passed to the other subtree.

In Figure 9 we show an example. If a message arrives at repeater R3 from segment S4, then a direction bit '1' can be interpreted as "pass the message", whereas '0' can be interpreted as "do not pass the message". Conversely, if a message arrives at the same repeater from segment S6, the meaning of the direction bits is reversed. Thus, a message from (segment 7, interface 2) to (segment 5, interface 21) arrives at repeater R3 through segment 6, and since its destination is in segment 5 (direction bit '1'), it is not passed. However, the same message with destination (segment 2, interface 15) is passed by repeater R3.

As a result of this "selective repetition", the communication system exhibits partial-broadcast transmission. For the case of point-to-point routing, only the interface that recognizes its address in the message destination field actually copies the message; the others (although "hearing") do not copy the message. For the case of imprecise partial-broadcast routing, every interface that "hears" the message eventually copies it. Although at first it seems that there may be many farthest away addresses for an arbitrary set of destinations, only one should be used if
Figure 9. Avoiding Unnecessary Repetition of Messages in Multiple-bus Architectures
unnecessary duplicates are to be avoided. For example, two farthest away addresses are likely to share a common bus as part of their trees so that any destination located at the common bus will receive the message twice!. Therefore, the destinations that are not covered by the farthest away address have to generate separate single-addressed messages. Consequently, imprecise partial-broadcast using partial-broadcast transmission is of little help in these architectures: the best we can do is to eliminate unnecessary propagations (and therefore, collisions) of messages by proper design of repeaters.

3.6 Comparing Algorithms

In this section we compare various partial-broadcast routing algorithms in local architectures. The performance of partial-broadcast algorithms for long-haul architectures has been described in [DAL77, McO78, DAL78].

In particular, we consider the following algorithms: (1) partial-broadcast routing using separately point-to-point transmissions, (2) imprecise partial-broadcast routing using full-broadcast transmission, and (3) imprecise partial-broadcast routing using partial-broadcast transmission. We shall refer to these
algorithms as A1, A2, and A3, respectively. Clearly, the "optimal" multi-destination routing algorithm (say A0) for local architectures consists of modifying the addressing recognition mechanism of the interfaces so that in one transmission those destinations indicated in a bit map and recognized by hardware copy the message.

The following notation will be used in our discussion:

- \( N \): total number of nodes
- \( n \): number of destinations in a multi-destination message \((0 < n < N)\)
- \( m_d \): average message delay
- \( c_d \): average delay in calculating farthest away address(es)
- \( c \): cost of discarding a message

The performance measures of interest are: (1) broadcast delay \( BD \), the total delay over all \( n \) destinations, (2) discarding cost \( DC \), the number of discards, and (3) Bit Rate \( BR \), the rate of bits transmitted per multi-destination message. Another (qualitative) measure of interest is difficulty of implementation.

Some differences in the choice of metrics for these algorithms in long-haul and local architectures are worth mentioning: (1) in long-haul architectures it is important to measure the number of messages transmitted per
link (say in units of message-hops per broadcast); however, in local architectures this measure is unimportant since once the message enters the communication system there is typically one path to its destination; and, (2) in long-haul architectures it is inefficient to discard messages since bandwidth is scarce; however, in local architectures is a reasonable alternative, although its cost is not negligible.

3.6.1 Broadcast Delay

For a multi-destination routing algorithm which simply sends separately single-addressed messages, the broadcast delay is at least \( n \) times the delay of each single-addressed message,

\[
BD(Al) \geq n \times m_d
\]

This is only a lower bound since \( m_d \) also increases with \( n \) (i.e., more separate transmissions imply more load). Since the average message delay \( m_d \) in a local architecture is mainly composed of access delay (e.g., negligible intermediate and propagation delays), the broadcast delay of this algorithm degrades worse in local architectures than in long-haul architectures. The reason for this is that in
local architectures there is no "pipelining effect", i.e., in a long-haul architecture the access delay is not a dominant component of total message delay so that separate copies of the same message are typically in transit simultaneously.

In an imprecise partial-broadcast algorithm using full-broadcast transmission, the broadcast delay is simply the access delay of the only message transmitted,

\[ BD(A2) = m_d \]

Thus, the delay is independent of the number of destinations.

For the imprecise partial-broadcast using partial-broadcast transmission, the broadcast delay is the same as in A2 (\( BD(A3) = m_d \)). However, there is some "hidden" delay \( c_d \) since it takes time to calculate the farthest away address(es). The nature of \( c_d \) depends on the architecture of interest. For example, in loop architectures it is possible to find algorithms that only examine once each destination of an arbitrary set of destinations for calculating the farthest away address(es). Let us show how this can be done in DLCN and DDLCN.

Suppose that in DLCN the direction of the loop coincides with the direction for which the numbers of the
(sequentially numbered) interfaces increase, i.e., messages pass from interface number \( i \) to interface number \( i+1 \) \((i=1,\ldots,N)\) and from interface number \( N \) to interface number \( 1 \). If we call \( \text{LOWER}(i)=\{1,\ldots,i-1\} \) and \( \text{SD} = \{\text{dest}(i),\ldots,\text{dest}(n)\} \), \((\text{dest}(j) \in \{1,\ldots,N\}, j \in \{1,\ldots,n\})\), then the farthest away destination of the set \( \text{SD} \) at interface \( i \) is given by,

\[
\text{farthest}(i,\text{SD}) = \begin{cases} 
\max\{\text{dest}(j)\} & \text{if } \text{SD} \cap \text{LOWER}(i) \neq \emptyset \\
\max\{\text{dest}(j), \forall \text{ dest}(j) < i\} & \text{otherwise}
\end{cases}
\]

A program to calculate this address can be,

```
proc farthest(i,SD)
    max_of_lower, max <- 0
    lower_is_empty <- True
    for each dest in SD
        if dest > i then
            if dest > max then max <- dest
            else lower_is_empty <- False
        if dest > max_of_lower
            then max_of_lower <- dest
    end
    if lower_is_empty then return(max)
    else return(max_of_lower)
end proc
```

An example of this calculation is given in Figure 10.

In DDLCN, there are two directions which can be followed by a message. Furthermore, a message does not have
Figure 10. Example of Calculating the Farthest Address in an Unidirectional Loop

\[ i = 9 \]
\[ SD = \{ 10, 11, 13, 14, 15, 2, 3 \} \]
\[ LOWER(i) = \{ 1, 2, 3, 4, 5, 6, 7, 8 \} \]
\[ LOWER(9) \cap SD \neq \emptyset \]
\[ \text{Farthest}(i, SD) = 3 \]
to pass through more than "half" of the number of interfaces. Thus, for a given set of destinations at node i, it is possible to find two farthest away addresses, one on each direction. Fortunately, under these characteristics it is still possible to find the addresses by examining only once the set of destinations:

Suppose the interfaces in DDLCN are sequentially numbered counterclockwise. For each interface i, it is possible to define half(i) as the number of the interface in its "opposite side" of the loop, e.g., if there are even number of interfaces the opposite is the one which has equal number of interfaces in between of them on each direction, and if there are odd number of interfaces the opposite is one of the two which have at most one extra interface in between of them on both directions. Each interface number then belongs to one of the following sets:

\[
S1 = \{ 1, \text{half}(i) \} \\
S2 = \{ 2, \ldots, \text{half}(i) - 1 \} \\
S3 = \{ \text{half}(i) + 1, \ldots, N \}
\]

To show the algorithm for selecting the addresses, we take an interface i from the set S3. (The algorithm for interfaces at other sets is similar or even simpler.) If we call "right" the counterclockwise direction and "left" the clockwise direction, the interface i has to give the following answers:
\[
\text{right}(i, SD) = \begin{cases} 
\max\{\text{dest}(j) \} & \text{if } \text{dest}(j) \notin \{1, \ldots, \text{half}(i)\} \\
\max\{\text{dest}(j), \text{est}(j) \notin \text{half}(i)\} & \text{otherwise}
\end{cases}
\]

\[
\text{left}(i, SD) = \min\{\text{dest}(j), \text{est}(j) > \text{half}(i)\}
\]

Notice that it is possible that either one of the answers does not exist, e.g., when all destinations belong to the same side of the loop relative to \(i\).

A program for this address selection can be the following:

```plaintext
proc farthest_addresses(i, SD)
    farthest_left, farthest_right <- False
    left_min <- N+1
    right_max1, right_max2 <- 0
    for each dest in SD
        if dest > i then
            if dest > right_max1 then /* keep max between i and N*/
                right_max1 <- dest
            else if dest > half(i) then /* keep min between half(i)+1 and i*/
                if dest < left_min then
                    left_min <- dest
                else
                    if dest > right_max2 then /* keep max between i and half(i)*/
                        right_max2 <- dest
        end
        if right_max2 = 0 then farthest_right <- right_max2
        else if right_max1 = 0 then
            farthest_right <- right_max1
        end
        if left_min = N+1 then farthest_left <- left_min
    return (farthest_right, farthest_left)
end
```
It is possible that after examining the set of destinations, one of the directions in DDLCN has only "few" (i.e., less than some threshold) destinations. Consequently, it is possible to refine further the algorithm so that it "preprocesses" the destinations before it attempts to find farthest addresses. If there are only "few" in one side, the algorithm may use virtual partial-broadcast in that side and imprecise partial-broadcast in the other side. An example of this calculation is given in Figure 11.

3.6.2 Discarding Cost

The total discard cost of each algorithm is given by,

\[
\begin{align*}
\text{Discard\_cost(A1)} &= 0 \\
\text{Discard\_cost(A2)} &= c*(N-1-n) \\
0 &\leq \text{Discard\_cost(A3)} \leq c*(N-1-n)
\end{align*}
\]

To estimate c one has to consider the size of messages vs. the available buffer at the nodes (interfaces), and the probability that several sources contain such nodes as part of their destination set. If c is very small (e.g., if the addressing mechanism is such that it checks the bit map of
i=13

half(1)=5
SD= \{14,15,16,1,3,9,10,11\}
right_max1= \max( \{14,15,16\} \cap SD) = 16
right_max2= \max( \{1,2,3,4,5\} \cap SD) = 3
left_min= \min( \{6,7,8,9,10,11,12\} \cap SD) = 9
farthest_right= 3
farthest_left= 9

Figure 11. Example of Calculating Farthest Addresses in a Bidirectional Loop
addresses in hardware before copying the message, or if storage is scarce), partial-broadcast using farthest addresses is not needed. In general, however, this is not the case so that we want to minimize such a cost by using A3. The actual total cost of A3 depends on how sparse the set of destinations is.

3.6.3 Bit Rate

In a local architecture (high bandwidth) the effect of transmitting some additional bits (a bit map) is negligible. Thus, for comparison purposes we can assume that each transmitted message has on the average the same number of bits. If we read \( m_d \) in units of time per bits of a message, we have:

\[
\begin{align*}
BR(A1) &= 1/(n \times m_d) \\
BR(A2) &= 1/m_d \\
BR(A3) &= 1/m_d
\end{align*}
\]

3.7 Summary

In this chapter we have considered the problem of
multi-destination routing in local communication system architectures. We characterized several distributed system architectures in terms of how multi-destination routing arises from them. The optimal solution in local architectures corresponds to an "all-hardware implementation" where the address recognition mechanism "understands" bit-maps. If we are willing to design the hardware in such a way this solution is certainly the best we can do. If we have a system whose addressing recognition mechanism only "understands" single-addresses and all-addresses, we can use the techniques developed in this chapter to implement multi-destination routing. In particular, loop architectures lend themselves to implementing partial-broadcast routing using partial-broadcast transmission. We showed that it is easy to implement such technique in DLCN and DDLCN. For multiple-bus architectures the best we can do is to use partial-broadcast transmission to eliminate unnecessary repetitions (repetitions that can cause unnecessary collisions).
CHAPTER 4

MULTI-DESTINATION PROTOCOLS

In the previous chapter we attacked the problem of multi-destination routing in several distributed system architectures. Routing, however, does not guarantee reliable exchange of multi-destination messages. For example, if some bits of the message are altered due to noise in the communication system, routing does not know about the damage. Furthermore, routing does not tell which destinations received the message, nor does it guarantee that each destination receives at most one copy of the message. These reliability problems are handled by process communication protocols.

In this chapter we design several multi-destination protocols, each one offering a different level of reliability. We call such protocols n-process communication protocols, or n-p protocols as initially suggested in [PAR78]. Our discussion emphasizes those design aspects that differ from conventional
single-destination (point-to-point, or 2-p) process communication protocols.

This chapter is organized in the following way. Section 1 is devoted to the lowest level n-p protocols. We call this class of protocols "unreliable protocols" since they offer minimum reliability: only damaged messages are discarded. In Section 2 we introduce two levels of reliability: Best-effort-to-Deliver (BD) and Guarantee-to-Deliver (GD). The use of each type of protocol is discussed. Section 3 explains differences in the system characteristics of long-haul and local architectures. Understanding such differences is important because they determine different protocol strategies depending on the architectures. Section 4 discusses in detail how BD protocols work. We show how to overcome some problems that can arise if processing systems fail at critical times. Section 5 is devoted to GD protocols. Finally, in Section 6 we present a general protocol architecture for distributed systems.

4.1 Unreliable N-p Protocols

This class of n-p protocols implements a facility by which a process can send a message to multiple remote processes. Damaged messages are discarded; however, no
attempt is made to acknowledge messages, nor to retransmit messages. The objective of this type of protocol is to provide a minimum-overhead facility which can be used either by processes of distributed algorithms (where the logic of the algorithm requires minimum-overhead and can take care of the unreliable situations) or by more reliable n-p protocols. The main functions of these protocols are (1) selecting an appropriate multi-destination routing algorithm, and (2) implementing process addressing.

4.1.1 Selecting Routing

This function consists of deciding on the most appropriate multi-destination routing algorithm for a multi-destination message. The choice is important because it can change the format of the message. For example, given an arbitrary set of destinations in an unidirectional loop architecture, this function first decides whether to use a virtual partial-broadcast routing algorithm or a partial-broadcast algorithm with full-broadcast transmission. In case that none of the above is the best choice, it calculates a farthest away address and uses a partial-broadcast routing algorithm (with partial-broadcast transmission). In its simplest form, the choice is based on fixed parameters (e.g., if the number of destinations is
less than 10% of the total number of destinations, use virtual partial-broadcast; if the number of destinations is within 10% of the total number of destinations, use partial-broadcast routing with full-broadcast transmission. Other choices may be based on dynamic information (e.g., if the communication system is lightly loaded, use virtual partial-broadcast regardless of the number of destinations of the message).

4.1.2 Addressing

In general, each processing system has its own conventions for naming its local processes. Therefore, the protocol uses a system-wide addressing mechanism that is locally translated at each processing system. Each element of this system-wide addressing space is a triple: (processing system number, algorithm number, port number). For each set of remote processes working on the same distributed algorithm (e.g., a set of cohorts), we assign the same algorithm number. Thus, every process automatically knows the address of its cohorts. Port numbers can be used to specify different message streams within a given process (e.g., if a concurrent program executes in a single processing system then the program may belong to the same system-wide distributed algorithm with
one algorithm number, and we can still identify each local concurrent process using port numbers).

The discussion above describes the addressing mechanism; however, it does not tell us how to put addresses in the actual messages or how to process the addresses. In general, we can assume the following three address-related fields for actually transmitted multi-destination messages:

1. source-address field, which encodes the address of the source process;
2. bit-map destination field, which contains one bit per processing system and indicates the destinations of the message; and
3. qualification field, which specifies further for which algorithm number (and port) the message is intended.

Figure 12 shows these fields.

Not all communication systems that implement partial-broadcast can accept this convention. In particular, those communication systems that always identify remote entities by name (e.g., DCS) cannot use it. In these systems, the "addressing" mechanism is simpler, but its effectiveness relies heavily on the handling of the associative memory: when to load it, what to store, etc.
Figure 12. Address-related Fields for Multi-destination Messages
However, the convention described above is adequate for many other systems.

Two observations should be made with respect to the destination bit-map:

1) The length of this field grows linearly with the total number of destinations. This is not expensive in local distributed systems since there is bandwidth in excess. The only problem in these architectures is when the length is so large that it forces long messages to be split into short messages (e.g., if there is some restriction on the size of messages acceptable by the communication system). In this case and in long-haul distributed systems, it may be useful to compress the field by defining subfields (e.g., if there are many consecutive 0s, we could replace them by a subfield whose content is the number of 0s), or change the radix representation of the field.

2) For imprecise partial-broadcast, a receiver simply checks a bit within the map to find whether the message is intended for it. Thus, the cost of discarding a message after copying it is not mainly in finding out that a message has to be discarded, but rather in tying up storage.
4.2 Reliable N-p Protocols

Network applications using reliable process-to-process protocols do not have sophisticated error recovery. Typically, if a file is transferred from one system to another and there is a failure at the receiver's site, the application may simply start all over again. In contrast, cohort processes of distributed algorithms typically perform error recovery as an integral part of their logic. As a result, the nature of reliable exchange of multi-destination messages in distributed algorithms may not be so apparent since one can always assume that part of the algorithm deals with the unreliable situations.

The motivation for reliable exchange of multi-destination messages can be best explained by considering what happens after a message is received at a destination. Once a message arrives at a receiver process, it is processed by the receiver. Thus, we can say that a multi-destination message intends to generate "actions" at multiple receivers. This "action-oriented" view of the message exchange problem means that unreliable n-p protocols provide a tool for generating multiple remote actions.

Unreliable n-p protocols are needed regardless of whether all reliability is delegated to the algorithmic level or some reliability is also made available at higher
protocol levels. In the first case, unreliable n-p protocols are the only protocol mechanism available to the algorithms, whereas in the second case more reliable n-p protocols can use unreliable n-p protocols for their implementation. This is analogous to implementing reliable process-to-process protocols using unreliable datagram protocols [ZIM75].

Unfortunately, delegating all message exchange responsibilities to the algorithms implies that the algorithms themselves may have to be involved with purely message exchange matters. For example, they may have to filter duplicated messages, since performing an action more than once may compromise the correctness of an algorithm (e.g., incrementing a value more than once!); or they may have to enforce order of messages since generating actions in some order other than the request order may also compromise the correctness of an algorithm (e.g., "read" and "change" actions in two messages may produce a different effect than the sequence "change" and "read").

Reliable n-p protocols can be used for solving these problems. These protocols can also be used when some assumptions can be made about the reliability of the actions themselves. In particular we shall consider two types of reliable n-p protocols:
(1) **Best-effort-to-Deliver Protocols** (BD protocols). This class of protocols try their best to make sure that as many destinations as possible receive a multi-destination message. Since the operation of transferring a multi-destination message under unreliable communication and processing systems can take arbitrary amount of time, it is necessary to limit the time for which the protocol makes its "best effort". Thus, at a sender's site the protocol has a bound on the time within which it has to give an answer. The sender process expects to know which destinations received the message and (if possible) which destinations did not receive the message. The nature of that answer depends on the characteristics of the communication and processing systems.

This class of protocols can be used to prevent the kind of duplicated or out of order actions described before. Another use of these protocols is when it is expensive to perform actions (e.g., a message whose content demands a complex update). In this case, when a process (of a distributed algorithm) reaches the point where it is indeed safe to propagate an action to some or all of its cohort processes, it can use a "fairly" reliable protocol (a BD protocol) so that more destinations receive the message and more chances are that the
receivers perform the actions concurrently. Finally, these protocols can also be used if the actions generated by messages have high (local) reliability; i.e., once the message is received by a destination the action is eventually performed due to the action's own built-in reliability. In this case an answer is important because the sender can assume that receiving a message at a destination is equivalent to performing the action.

(2) Guarantee-to-Deliver Protocols (GD protocols). This class of protocols guarantee to a sender process that its multi-destination message will eventually arrive at every destination. A receiver process of this class of protocols is assured that once it gets a message, it has a "safe" copy of the message. This means that if a receiver fails during processing of the message, upon becoming active again it still has the same initial copy of the message. The receiver has the responsibility of deleting the copy since only the receiver knows when the copy is not needed any more.

This class of protocols can be used to ensure that no destination misses its opportunity of performing the action, even if its site has failed or is unreachable at the time when it should first attempt to perform the
action.

In the following sections we describe in detail the design of these protocols. Since their design depends on characteristics of the distributed system architecture, we shall consider in the next section those characteristics that are relevant to reliability.

4.3 System Characteristics

For unreliable n-p protocols, the characteristics of the processing systems are unimportant since it does not matter whether the processing systems get the message, or whether they have failed. (Notice that at this level a sender is not expecting an answer). Lacking the capabilities for guaranteeing delivery, detecting duplicates, or preserving order, such a level does not need extra control messages, nor does it need control information kept by the protocol itself. The only important characteristic of the communication system is routing, as discussed before.

On the contrary, reliable n-p protocols add some reliable features to the transfer of multi-destination messages. In order to masquerade unreliable features, it is necessary to understand those anomalies and constraints
introduced by communication and processing systems.

4.3.1 Communication System

Some characteristics of communication system architectures determine the strategies for (and even the existence of) solutions to problems of unreliability. Understanding these characteristics is extremely important because, in general, a protocol can do very little unless we make some assumptions. (For a nice description of what process-to-process protocols can not do see [SUN75].) Thus, matching a set of assumptions with some real-life system characteristics determines the applicability of a protocol. Table 2 contrasts some important characteristics of long-haul (packet-switched) and local communication system architectures.

The bandwidth of a local architecture is typically orders of magnitude larger than the bandwidth of long-haul architectures. A multi-destination message in a long-haul architecture can be lost so that some destinations may not receive the message. By contrast, a multi-destination message in a local architecture is less likely to get lost since, for example, there are no intermediate nodes in these architectures. Message transit time is also different in long-haul and local architectures. When several
<table>
<thead>
<tr>
<th></th>
<th>Long-haul</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bandwidth</strong></td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Message Loss</strong></td>
<td>Possible</td>
<td>Unlikely</td>
</tr>
<tr>
<td><strong>Message Transit Time</strong></td>
<td>Variable</td>
<td>Constant</td>
</tr>
<tr>
<td><strong>Alter Message Order</strong></td>
<td>Possible</td>
<td>Unlikely</td>
</tr>
<tr>
<td><strong>LD,FD</strong></td>
<td>Large</td>
<td>Small</td>
</tr>
<tr>
<td><strong>Delay</strong></td>
<td>Significant</td>
<td>Negligible</td>
</tr>
</tbody>
</table>
multi-destination messages are sent from a source, a long-haul architecture can alter their order at some destinations, if the multi-destination routing algorithm can direct messages through alternative paths. In a local architecture, a multi-destination routing algorithm is typically source-based and fixed with respect to each destination; therefore, the communication system itself does not alter the order of a sequence of messages.

Another difference is noted on multi-destination message delay. For these messages, each destination usually receives the message at different times. Let us call first-destination the destination that happens to receive the message first, and last-destination the destination that happens to receive the message last. A last-destination delay (LD) can be defined as the time elapsed between submission of the message and its arrival to the last-destination, and a first-destination delay (FD) can be defined as the time between submission of the message and its arrival to the first destination. Of course, in theory LD (and consequently FD) can be infinite since no multi-destination routing algorithm can guarantee completely that each destination of a multi-destination message receives the message. The interesting points, however, are on magnitudes and relationships of these delays.

In a long-haul architecture, a message delay is
composed of transmission delays (low-bandwidth), propagation delays (large distances), and queueing delays (competition with concurrent traffic after a packet is stored at intermediate nodes). Delays for multi-destination messages are also composed of these terms. Furthermore, the difference between FD and LD is not negligible because it can also be composed of similar terms. (The nature of these delays depend on the actual multi-destination routing algorithm). In a local architecture, the delay is basically composed of access delay and the magnitudes are typically smaller. Also the difference between LD and FD is almost negligible. (The only case where this is not true is when there are many destinations and virtual partial-broadcast is used.)

4.3.2 Processing Systems

The anomalous behaviour of interest in a processing system is any failure that can destroy either the control information maintained by the protocol, or the content of messages (e.g., memory loss). In general, these types of failures can happen regardless of whether the distributed system is local or long-haul.

We shall assume for some reliable n-p protocols the existence of two types of storage at the processing
systems: unsafe storage and safe storage. (These are equivalent to volatile and stable storage in [LAM76]; however, we use them for storing protocol information.) The important difference between these two types of storage is that once a safe storage is successfully written its content can be recovered in spite of failures; however, information in unsafe storage can be lost at any moment.

4.4 Best-effort-to-Deliver Protocols

As mentioned before, these protocols implement some reliable features in the transfer of multi-destination messages. In this section we shall explain in more detail the internal mechanisms used by these protocols.

4.4.1 The Basic Mechanisms

Some of the typical steps performed by these protocols are as follows. A process submits a multi-destination message. By noticing the source process "id", the protocol automatically recognizes the destination address of each receiver since all destinations are cohort processes (i.e., they have the same algorithm number) and the source indicates which are the destination processing systems (either explicitly by naming each one or implicitly
by naming a predefined group).

The protocol at the sender's site waits for individual acknowledgements after transmission. At transmission time the protocol can use the unreliable n-p protocol. To keep track of which receivers have acknowledged, the protocol at the sender's site has an "Acknowledgement Bit Map". When this map matches the destination bit map, the multi-destination message has been successfully delivered by all destinations. If not all destinations have acknowledged within some time bound (e.g., the retransmission interval), the protocol at the sender's site retransmits the multi-destination message to only those destinations present in the destination bit map, but missing in the acknowledgement bit map. (Note that in an addressing recognition mechanism using associative memory this selective retransmission is not possible). This operation can be repeated up to some maximum number of times (e.g., the maximum number of retransmissions). Should this limit be reached, the protocol at the sender's site gives an answer to the sender process.

4.4.2 Differences with Reliable Point-to-point Protocols

Some basic differences between reliable point-to-point protocols and best-effort-to-deliver
protocols are worth mentioning:

(1) If the (only) destination processing system in a point-to-point protocol fails, the protocol has to stop the exchange of messages. In contrast, some destination processing systems may fail in a multi-destination (BD) protocol and the protocol can still exchange messages (the reason is the level of reliability guaranteed by best-effort-to-deliver protocols).

(2) The notion of "connection" (i.e., an association between a sender process and a receiver process) in point-to-point protocols gets extended into the notion of "association group" (i.e., an association between a sender process and a set of receiver processes).

(3) Connections are assigned dynamically in point-to-point protocols. In a way, this reflects the type of software for which connections are intended, e.g., network services that are requested dynamically by users. Consequently, opening and closing connections are important. In contrast, association groups are used by processes of distributed algorithms that typically multiplex users at the algorithmic level, e.g., there is only one distributed concurrency control algorithm that handles any number of concurrent users. Thus, association groups can be thought of as always being
"opened".

(4) Connections can be "reused" in reliable point-to-point protocols, e.g., the same connection can be used by different users at different times. Thus, in architectures with arbitrary delay, it is important to initialize in a reliable fashion the sequence numbers used by connections [GAR77]. Furthermore, selecting an initial sequence number may not be trivial. By contrast, since association groups are not dynamic (for the reasons explained in the previous point), they are not reused. Consequently, initializing or selecting sequence numbers is not a major issue. However, a different synchronization problem with respect to sequence numbers arises in association groups. We shall discuss this problem later on (section 4.4.4).

4.4.3 Well-behaved Communication Systems

Suppose we have a distributed system where messages are never lost or duplicated by the communication system, and messages are always delivered in the same order as they are sent from a source. The processing systems attached to this well-behaved communication system, however, are prone to failure.
In practice, there may be systems that match very closely such characteristics. For example, DDLCN has high built-in reliability in its communication system so that it is very unlikely for messages to be lost (because of the double loop), or duplicated (because of the lack of store-and-forward). Nevertheless, its processing systems are subject to failures.

Damaged messages in these systems can be handled by sending negative acknowledgements. (Since messages are never lost, an acknowledgement is always received by a source.) In local architectures under light traffic, it is possible to find a bound on message delay. Such a bound simplifies the mechanisms used by a BD n-p protocol. In particular, the protocol does not need sequence numbers. Furthermore, under these assumptions this class of protocols can give an exact answer to the sender, i.e., it can tell what truly happened at each destination. In this case, the answer tells which destination received the message and which destination did not get the message because it had failed (i.e., it was "dead") at the time. This can be accomplished because no-answer from a destination can be unambiguously interpreted as a processing system failure.

Figure 13 illustrates a typical transmission scenario using this kind of BD n-p protocol. Notice that even if a processing system fails after it receives the
PROCESSING SYSTEM

MULTI-DESTINATION MESSAGE

FAILED PROCESSING SYSTEM

Figure 13. N-p Protocol with Bound Delay
message, the protocol reports an exact answer.

4.4.4 Ill-behaved Communication Systems

Communication systems usually have internal mechanisms to recover from message loss. These mechanisms are low-level protocols that retransmit a message in case no internal acknowledgements are received; interfaces in local architectures may retransmit a message when interfaces do not acknowledge, or in long-haul architectures there is typically a node-to-node protocol that retransmits messages. However, there is always a limit on the recovery attainable by using low-level protocols, e.g., an interface in a local architecture may fail at a critical time, or a node in a store-and-forward long-haul architecture may fail after storing a message and before forwarding it. In these cases, higher level protocols should be designed to deal with message loss. (Recall that this discussion refers to multi-destination messages.)

In local architectures, a message loss typically implies that there is a catastrophic failure (e.g., a bus failure in a single-bus or multiple-bus architecture, or an interface failure in loops). Such failures cannot be recovered by simply retransmitting the message (a high-level protocol can do very little in these cases). Long-haul
architectures, however, can recover from message loss in many cases by retransmission at a high-level protocol level. The reason is that retransmitted messages can go through alternate routes.

The protocol mechanism to deal with message loss is then "retransmission on time-out". Unfortunately, retransmission in architectures with variable message delay can cause message duplication, if a message is retransmitted just before the acknowledgement of the previous attempt arrives. In this case, the protocol itself can generate duplicates. To solve this problem, the protocol has to use sequence numbers.

Since a BD n-p protocol sends one message at a time (recall that we defined the protocol such that the interaction is similar to "inquiry/response" protocols: send a message, wait for answer, send the next message, wait for its answer, etc.), the size of the sequence number space is two. At the sender's site the protocol maintains one Next Sequence Number (NSN) and at each receiver's site it maintains an Expected Sequence Number (ESN). At each receiver's site, the protocol rejects messages whose sequence numbers differ from its ESN (see Figure 14).

For each algorithm consisting of N remote processes, we could have \(2^{N-1}-N\) different NSNs at each protocol site since this is the number of possible
Figure 14. Detecting Duplicates
destination groups with more than one destination. Furthermore, each protocol site could have to maintain as many as \(((N-1) \times (2^{N-2}-1))\) different ESNs since this is the number of possible destination groups to which a site can belong. Fortunately, it is very unlikely that every process wants to exchange messages to every possible destination group at a given point in time. However, this potential growth of control information suggests that dynamic mechanisms should be provided for creating it and destroying it.

By creating this information when it is first used, the protocol has to maintain information only for association groups which are actually used. Under local architectures (short message transit time) it is possible to select 0 as the default initial value for NSNs and ESNs. If too many sequence numbers have to be maintained at a given time, the protocol may have to destroy some association groups, e.g., by selecting the less frequently used.

Another problem of interest is what happens when a processing system becomes active after a failure. Since the control information (sequence numbers) in general can be lost when a processing system fails, it is important to synchronize the sequence numbers and not simply use the default values (see Figures 15 and 16).

For the NSNs, the synchronization consists of deleting all
Association Group= $(S(0), d(1), d(2), d(3))$

Figure 15. Need for Synchronizing NSN After a Processing System Becomes Active
Figure 16. Need for Synchronizing ESN After a Processing System Becomes Active
ESNs at remote sites where a process of the processing system is the source process of an association group. For the ESNs, rather than assuming a default value, the protocol takes the first arriving sequence number as the ESN (1). Only when the first operation (on NSNs) completes, the processing system can start accepting or sending multi-destination messages. An example of this synchronization is shown in Figures 17 and 18.

In Figure 17, the site of S(0) becomes active and the protocol sends a "delete" message in order to delete all ESNs where a process of this site is a source of association groups. If other sites are dead, there is no problem because those sites will eventually synchronize when they become active. (In case messages are lost, the protocol can simply send the delete message several times.) In Figure 18, the site of d(1) becomes active and the protocol will simply accept the first SN of a message as the ESN.

4.4.5 Answers

An important point about these protocols is the

(1) For this technique to work in architectures that may have messages of long transit times, it is necessary to use a mechanism such as incarnation numbers [GAR77].
Figure 17. Synchronization of NSNs
Figure 18. Synchronization of ESNs
"quality" of answer that they can give to a sender process. This is important since based upon an answer the sender process may generate a variety of actions (e.g., it may include those destinations that did not answer in an error recovery mechanism). In general, the sender process should not assume anything about those destinations that did not positively acknowledge the message.

As mentioned before, in a distributed system with a well-behaved communication system, the protocol can give an exact answer. If there is no way of distinguishing between a processing system failure and a message loss, an answer gives very little information with respect to those destinations that did not acknowledge, e.g., after reaching the maximum number of retransmissions it is possible that the protocol cannot tell whether a processing system failed during the attempts or was dead during all the attempts or was dead at all. Typically, in long-haul architectures by increasing the number of retransmissions the probability of message loss will decrease. In local architectures, increasing retransmissions and not getting answers means that a catastrophic failure has occurred. Thus, in this case it is easier to distinguish between message loss and processing system failure.

In some architectures, it is possible to isolate a processing system failure from some other anomalies by using
information external to the protocol, e.g., a system with (lower-level) built-in software logic which constantly tests the status of a system by sending "are you alive?" messages. Also, some processing system failures may be scheduled (e.g., maintenance) so that remote processing systems know in advance the down-time of a system. In these cases, the protocol can discriminate message loss from a processing system failure.

4.5 Guarantee-to-Deliver Protocols

These protocols are highly reliable and do not have a low delay requirement. In order to achieve such degree of reliability, it is necessary to use reliable storage, i.e., Safe Storage (SS). The basic steps in this class of protocols can be summarized as follows:

(1) **Commit Phase.** A sender process submits a multi-destination message. At the sender's site, the protocol stores a copy of the message in SS, initializes an Acknowledgement Bit Map (ABM) kept in SS, and picks up the sequence number from the NSN (also in SS). Upon successful completion of this phase, the protocol at the sender's site can give an answer to the sender process ensuring that one copy of the message will eventually
arrive at every destination.

Should failure occur at the sender's site before the answer is given, no problem arises since the protocol has not yet been committed to the sender process. To avoid having copies of messages which are not committed to a sender, it is possible to make the previous local operations "atomic" [LAM76], i.e., either all steps or none are carried out.

(2) Propagation Phase. After a message, its sequence number, and its ABM reside in SS, the protocol sends the message to its destinations. Every time an acknowledgement is received, the ABM is updated. The protocol uses a time-out mechanism to regularly retransmit the message to only those destinations which have not acknowledged. When all destinations acknowledge, the copy of the message is erased from the sender's site.

Should a failure occur at the sender's site, no problem arises because all critical information has been safely stored in SS. When the sender's site becomes active again, it continues checking the ABM, retransmitting the message and resetting the time-out until all destinations receive the message.

(3) At a receiver's site, the protocol keeps a copy of the message in SS. Since the Expected Sequence Number is
also stored in SS, the protocol can filter duplicates. (As in process-to-process protocols, duplicates are also acknowledged to recover from loss of acknowledgements.) The message can be given to the destination as soon as it is safely copied in SS. Eventually the receiver process deletes the copy of the message. Should a failure occur at the receiver's site after the message has been received, it is up to the receiver process either to reprocess the message or to delete it.

In contrast with BD protocols, GD protocols handle simultaneous messages on the association groups. Thus, the size of the sequence number space can be greater than two. The techniques for detecting duplicates and enforcing sequencing are similar to the techniques used in point-to-point protocols. Figure 19 shows a transmission scenario for this class of protocols.

Although the protocol described above illustrates one implementation strategy for the propagation phase, there are other ways of achieving the same effect yielding a different performance. For example, the described strategy exploits parallelism since at all times it uses multi-destination protocols (e.g., an unreliable n-p protocol); however, if the sender's site fails when it has not finished, all the destinations which have not received
Figure 19. A Guarantee-to-Deliver Protocol
the message are affected. An alternative to this approach is to propagate the message sequentially throughout the set of destinations. In this case, after the first destination receives the message, the sender's site can delete its copy and the reliable propagation continues depending at different times on the availability of different sites.

4.6 A Protocol Architecture

A general protocol architecture for these systems is shown in Figure 20. The region of "distributed-software processes" corresponds to processes (system processes) that cooperate with multiple remote processes to manage remote resources. Several process communication facilities are available to them: point-to-point message exchange facilities, multi-destination message exchange facilities, and special-purpose facilities.

The message exchange facilities can be called "application independent" since they just aim at passing messages. Point-to-point facilities are necessary since distributed algorithms also require exchange of single-addressed messages. The multi-destination facility corresponds to the protocols presented before.

Services (e.g., file transfer) may also be
Figure 20. Architecture of IPC protocols for Distributed Systems
necessary! whether these services can be used directly by
users or by distributed-system processes is irrelevant as
far as protocols are concerned. Other special-purpose
protocols suitable for distributed systems can also be made
available to distributed-software processes. In particular,
we shall see in later chapters that some synchronization
mechanisms are good candidates for being used at this level,
much like any other service protocol.

4.7 Summary

In this chapter we have designed three types of
multi-destination protocols, each one offering a different
level of reliability. The first level (unreliable n-p
protocols) offers very low reliability. This level is
analogous to a datagram level in point-to-point protocols.
The new interesting feature of this level is the selection
of appropriate multi-destination routing algorithms. The
next level (best-effort-to-deliver n-p protocols)
corresponds to a "medium" reliability protocol. These
protocols make an effort to deliver a multi-destination
message to as many of the destinations as possible,
filtering duplicates and enforcing order. Depending on the
distributed system architecture, the mechanisms used by
these protocols can be simplified. We also discussed the
differences between these protocols and reliable point-to-point protocols. The third level (guarantee-to-deliver n-p protocols) is a high reliability protocol. We showed how to ensure such reliability by using reliable storage.
In the previous chapter we have designed several multi-destination interprocess communication protocols for distributed systems. These software mechanisms are necessary for supporting many types of distributed algorithms. A very important consideration in any software system deals with the way users "see" the system. Having a very efficient software tool that is very difficult to use is of little value to the overall system design. Thus, in this chapter we address the question of how to use such an interprocess communication protocols. In answering this question, we enter the domain of programming languages since it is by means of languages that programmers ultimately use these protocols.
The need for integrating interprocess communication protocols into high-level programming languages has been recognized by network designers [SPR78] as well as language designers [FEL79]. Two approaches to this problem have been proposed: (1) to provide send/receive primitives at the programming language level [FEL79, AMB77], and (2) to generalize the call/return constructs for remote procedures [BRI78, WHI77]. We shall show in this chapter that none of these approaches is appropriate for writing distributed software.

This chapter is organized in the following way. In the first four sections we introduce the notions of distributed models of computation, distributed programs, and distributed programming languages. The domain of distributed programming is compared with that of existing programming models: sequential programming, concurrent programming, and real-time programming. In Section 5 we introduce the basic idea of our mechanism, namely the notion of "remote-operations". Section 6 shows how this concept coupled with other features of an object-oriented language can handle many communication disciplines.

5.1 Centralized Programs

The notions of sequential and parallel algorithms
are very familiar to all of us. Since a program is a way of expressing an algorithm, it is possible to talk about **sequential programs** and **parallel programs**. Programs are executed by a **model of computation**. Such models can be physically realizable (e.g., PDP-10) or physically unrealizable (e.g., Oracular Turing Machine). In real-life programs, the underlying model of computation has to be physically realizable.

Typically, the execution of a parallel program implies the existence of a real parallelism during the execution of some instructions. Other programs may or may not be actually executed in parallel; however, they possess a potential parallelism. These programs are called **concurrent programs**. Parallel programs are concurrent programs; however, not all concurrent programs are executed in parallel. For example, it is possible that the execution of instructions of different "modules" of a concurrent program can be interleaved (still one at a time) in some unpredictable way.

All these types of programs have traditionally run under a **centralized model of computation**. In this model any processor can typically execute any instruction of the program. Thus we can say that sequential, parallel, or concurrent programs can be **centralized programs**.
Recent advances in communication, computer network, and hardware technologies have provided us with another physically realizable model: a distributed model of computation. In contrast with the previous model, however, not all processors can execute all instructions of a program in a distributed model of computation. Under this model, a program is viewed as a set of program components, each one being stored and executed by a different processing system. The only way of communication among the components is by means of an exchange of messages. Programs under this model can be called distributed programs. A distributed program can also be sequential, parallel, or concurrent.

A very important component of a distributed model of computation is a subsystem by which messages are transported: the communication system. Changing some properties of the communication system may result in different distributed models of computation. Whether we assume no message loss, no communication system partition, finite message delay, bounded message delay, and so forth, can make a big difference in the properties (and even the existence) of computations under the model. For example, running a program on a PDP-10 where all program modules happen to communicate in the form of messages, indeed
reflects a distributed model of computation; the model in this case has the properties of no message loss and negligible message delay. On the other hand, running a program under the Actor model of computation [HEW77] or in such a microcomputer network as in [BRI78], also reflects a distributed model of computation; the model in these cases assumes finite (but variable) message delay.

5.3 Distributed Programming Languages

In the same way that a sequential, parallel, or concurrent program for a centralized model of computation can be written using one or several programming languages, any distributed program can be written using one or several programming languages. In particular, we are interested in languages that allow us to write a distributed program for a very "realistic" distributed model of computation. By a "realistic" model we mean that the distributed systems under consideration do not have perfect communication or processing systems. We seek a language that can abstract very nicely such unavoidable features of distributed systems as message handling, message delays, unavailability of processing systems, and failures of processing systems.

The need for a distributed programming language is becoming more apparent with the advent of "distributed
software", as explained in Chapter 2. Typically, the description of this software is done either in plain English or by means of some graph-like notations. Unfortunately these descriptions are often ambiguous or too far from the domain of the system, leaving plenty of room for mistakes when they are actually implemented.

5.4 Programming Models

Writing a distributed program in a distributed programming language is different from writing programs under existing programming models. Table 3 contrasts some well-known programming models based on features that become important in distributed systems (some terms have been borrowed from [WIR77]). The language features of interest are: concurrency, communication, synchronization, time, and fault-tolerance.

Concurrency means that a language provides some way for letting groups of instructions execute concurrently. Such a group of instructions is typically called a "process". The next two features relate to the interactions of such processes: communication captures the mechanism by which a language allows the exchange of data between (among) processes, and synchronization encompasses language features
Table 3. Programming Models

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>TYPICALLY REQUIRED IN</th>
<th>TYPES OF PROGRAMS</th>
<th>EXAMPLE OF A LANGUAGE</th>
<th>CONCURRENT</th>
<th>COMMUNICATION BY</th>
<th>SYNCHRONIZATION</th>
<th>TIME DEPENDENCY</th>
<th>FAULT TOLERANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENTIAL</td>
<td>All Systems</td>
<td>Sequential Program</td>
<td>FORTRAN</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Little Concern</td>
</tr>
<tr>
<td>CONCURRENT</td>
<td>Operating Systems</td>
<td>Multi-program</td>
<td>Concurrent</td>
<td>Yes</td>
<td>Shared Data</td>
<td>Monitors</td>
<td>No</td>
<td>Little Concern</td>
</tr>
<tr>
<td>(PARALLEL)</td>
<td></td>
<td></td>
<td>PASCAL</td>
<td></td>
<td>Objects</td>
<td>Conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL TIME</td>
<td>Process Control</td>
<td>Real-Time Program</td>
<td>MODULA</td>
<td>Yes</td>
<td>Shared Data</td>
<td>Interface Modules</td>
<td>Processing Time</td>
<td>Some Concern</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Objects</td>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISTRIBUTED</td>
<td>Distributed Operating,</td>
<td>Distributed Program</td>
<td>?</td>
<td>Yes</td>
<td>?</td>
<td>?</td>
<td>Processing Time</td>
<td>Major Concern</td>
</tr>
<tr>
<td></td>
<td>Data Base, and File</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message Delay</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
available to programmers for enforcing some order of events. The time feature denotes language tools which allow programmers to manipulate time. Finally, fault-tolerance refers to whether the language permits programmers to handle anomalies of the underlying model of computation. In particular, the last feature is designed to cope with both communication system anomalies (e.g., some systems become unreachable from other systems because of a communication system partition) and processing system anomalies (e.g., main memory loss), both of which are important issues in distributed system software. Notice that this feature is different from those language constructs for catching "residual software errors" as in [RAN75], or to a great extent, from "exceptional condition mechanisms" as in [LEV77].

5.4.1 The Sequential Model

Sequential programs are typically needed in all systems. Traditional programming languages such as Fortran and Algol have been used for many years as tools for expressing sequential algorithms. The evolution of these languages into "structured" languages that support abstraction mechanisms has been an important contribution to the development of programming languages (e.g., CLU [LIS77],
The issue of concurrency does not arise in this programming domain, nor do communication and synchronization. Languages for this model do not normally allow programmers to express time-dependent computations, since this is not needed for most sequential programs. Finally, failures of the types described above are not handled by these languages, nor should they be. For example, if there is a power failure that destroys the content of main memory, the program is simply executed all over again.

5.4.2 The Concurrent Model

The next programming model is for concurrent programming. Although in our table we have included features available in Concurrent Pascal (BRI75), other forms of languages for concurrent programming exist (BRY78). Of particular interest are concurrent programming languages where communication is done by means of messages (AMB77, KAH77). In these languages, avoiding mechanisms that rely on global state information (easy to implement in a centralized model of computation) seems a first step towards suitable program constructs for distributed model of computations. Unfortunately, the underlying models for which these message-based languages have been designed (or
proposed) are too idealistic. Typically, these models assume that a message always arrives at a destination, or a processing system is always reachable and never fails. These assumptions are unsatisfactory if the resulting language is to be used for writing robust software (i.e., programs able to withstand the types of failures mentioned above).

5.4.3 The Real-time Model

Another programming model is for real-time programming. The most distinguishable characteristic of this domain is that programs are time dependent. In addition to concurrency, communication and synchronization requirements, real-time programs depend for their correct behavior on the execution speed of processors [BRI78, WIK77].

5.4.4 The Distributed Model

This model presents an unusual combination of all features mentioned above. First, a language has to provide for concurrency since a distributed system is a concurrent architecture. Second, communication requires that the language be interfaced with interprocess communication
protocols, since these protocols are the only mechanism provided by the system to exchange messages. Third, synchronization mechanisms in the language cannot rely on global environments and have to be fault-tolerant. Fourth, distributed programs are time dependent, not only on processing time but also on message transmission time. Finally, for realistic environments, the language has to provide tools for handling anomalous behavior of both communication and processing systems (I).

The design of a language for such an unusual environment is a very difficult task. Table 4 illustrates some of the current trends in languages to deal with this environment. In the next sections, we shall focus on the communication features of a distributed programming language. Such features represent the programmers' abstract view of interprocess communication protocols. Interestingly enough, we shall also see how other languages features (e.g., time, fault-tolerance) are very much interrelated in a distributed system. We shall assume the existence of an

(I) A classical example of a distributed program is a program to implement communication protocols. Typically, all program components are remote and the same (this is a nice property because the programmer writes a component once and then many copies are made). Each component has a sending discipline and a receiving discipline; there are many concurrent events; communication is by messages; time is important for reliability reasons; and failures of communication and processing systems are part of the specification of the program.
### Table 4. Some Distributed Programming Languages

<table>
<thead>
<tr>
<th>LANGUAGE</th>
<th>MECHANISM</th>
<th>PROTOCOL</th>
<th>MULTIPLE TRANSFERS</th>
<th>TIME-OUT</th>
<th>LANGUAGE CONSTRUCTS</th>
<th>ERROR HANDLING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure-Call [WHI77]</td>
<td>Call</td>
<td>2-Process</td>
<td>Awkward</td>
<td>No</td>
<td>No</td>
<td>Not Discussed</td>
</tr>
<tr>
<td>Pascal-Like [BRI78]</td>
<td>Call</td>
<td>2-Process</td>
<td>Awkward</td>
<td>No</td>
<td>Guarded Regions</td>
<td>Not Discussed</td>
</tr>
<tr>
<td>EPL [MAY78]</td>
<td>Send</td>
<td>Not</td>
<td>Not</td>
<td>No</td>
<td>No</td>
<td>Not Discussed</td>
</tr>
<tr>
<td>PLITS [FEL76]</td>
<td>Send</td>
<td>2-Process</td>
<td>Awkward</td>
<td>Yes</td>
<td>No</td>
<td>Discussed (Still Open Questions)</td>
</tr>
</tbody>
</table>
object-oriented language in our examples.

5.5 The Notion of Remote-Operations

Although ultimately the communication between (among) remote program components has to be in the form of messages, it does not seem appropriate to have message exchange primitives at the disposition of programmers. Good programming practices suggest that programmers should rather have meaningful operations at their level, leaving to the compiler the job of generating automatically the low-level message exchange operations. (After all, a message is exchanged because "something" is to be done at a remote location!). Thus, a basic observation can be made about distributed programming:

Having message exchange operations (e.g., send and receive) at the programming language level is analogous to having memory access operations (e.g., store and load) at the programming language level: both sets of operations simply reflect an architectural characteristic that, although needed at the code level, does not have to be visible at the programming language level, and probably should not be.

In order to avoid low-level message exchange operations at the programming language level and at the same time to have an equivalent expressive power to that provided by the low-level message exchange operations, the language
must provide constructs to define operations on remote data objects or operations corresponding to remote procedures. Both types of operations can be called *remote-operations*. The distinguishable characteristic of such an operation is that it may not finish; i.e., once it starts it is possible that no answer is received within some time bound. Thus a remote-operation also captures the notions of arbitrary delay and system failures.

5.6 Communication Disciplines

In this section we shall show how remote-operations provide a uniform way of handling many communication disciplines. A communication discipline models a type of interaction between (among) remote program components.

5.6.1 Action-propagation

Suppose a program component at site 0 wants to change the value of a certain key stored in a table at site 1, and the actual searching and updating procedures are executed at site 1. Furthermore, assume that this operation occurs within the logic of a program such that it is unimportant at the time of the invocation whether the
operation is actually successfully executed, e.g., the program may perform some action at a later time to find out whether the operation has indeed been executed. This is an action-propagation communication discipline with minimum requirement on reliability.

Using the message exchange paradigm with low-level message exchange operations in a programming language, the communication may look as follows (2):

```/* at site 0*/

  ...

  type message=record operation:char;
     key:char;
     newvalue:integer;
  end

  var m:message;
     ...

  m.operation<-update_op;
  m.key<-some_key;
  m.newvalue<-some_value;
  send(m) to P;

  ...
```

(2) The examples show the source-program version of the code actually executed at each site; however, we are not addressing the question of how to get from the source-program level to the code-program level. One possible answer is to assume that (a) there is some kind of "distributed compiling system," where different program components can be compiled at different sites; and (b) before execution all bindings between message exchange operations and ports used by interprocess communication are made.
/* at site 1*/

var table: array [1...n] of record
    name: char;
    value: integer;
end record

/* similar message type declarations as in site 0*/

receive (m) from P0
if m.operation=update_op then
    update(m.key, m.newvalue);

Notice that the programmer has to actually specify all message related operations, e.g., build the message, send the message, receive the message, and decode the message. The nature of send operations requires the programmer to specify the "name" of some point of reception for the message, e.g., the "id" of the entire program component. If there are several points of reception, several names have to be associated. In any case, these names typically do not contribute to the clarity of the program. Similar remarks apply to receive operations. Another subtle point is when a send operation is considered to be finished: after the message is successfully received or as soon as the interprocess communication mechanism starts sending the message. Whatever the choice (which is
even more complex for multi-destination messages), it adds very little clarity to the domain of solution to a problem.

Consequently, the programmer "gets involved" in operations which, although needed, do not contribute to the actual logic of the solution. Using the remote-operation paradigm in a distributed programming language, this communication discipline may look as follows:

```c
/* at site 0*/

remote_object table at site sitename;
operation update(key:char, value:integer)

update(some_key, some_value)
```


The definition of a data object can include what its "home" site is. It may also be desirable to create multiple remote copies perhaps with different representations at different sites. At the site of the remote operation (site 0) we have included a definition of which objects are remote, where they are, and which operations will be attempted. If we do not include this information, the interface to the interprocess communication mechanism becomes very complex. On the other hand, putting in at least some information may result in extra checks at compile time (e.g., parameter types).

The advantage of this remote-operation paradigm is its clarity of programming since all operations have meaning within the solution. Of course, the compiler has to
generate appropriate code to interface with the interprocess communication mechanism. In this example, the automatic expansion at the site of the remote-operation resembles the previous solution using the message exchange paradigm, since the parameters of the remote-operation are simply fields of a message sent to the site where the remote object resides. At the site of the remote object, the actual code of a data object gets augmented with a "receive" primitive and appropriate code to perform the operation.

Finally, notice that for this example, a distributed programming language with only "call" primitives to remote procedures forces the sender to wait for a reply.

5.6.2 Single-request/Single-reply

Suppose now that we have to solve a similar problem but with a request-reply communication discipline. In this example, we assume that rather than updating, the program component at site 0 wants to read the value of a key. Furthermore, we also assume that there is a bound on the time for which the programmer at site 0 is willing to wait for the answer. (It is important for programmers to be able to manipulate this type of time-dependence; otherwise, the programmer is forced to ignore an important characteristic of distributed systems.) Using the message
exchange paradigm, this communication discipline may look as follows:

```plaintext
/* at site 0*/

```message`` = record
  operation: char;
  key: char;
  answer: integer;
end

var m: message;

on time_out(timer_id) do
  /* action to follow on time-out */
end

m.operation <- search_op
m.key <- some_key
send(m) to P1
set_timer(timer_id, bound)
receive(m) from P1
  /* no time-out */
clear_timer(timer_id)
```

```
The important observation now is that with this type of solution, the programmer is forced to design time-out constructs independent of the message exchange operations. Trying to incorporate timing within the semantics of "send" does not help since this can only encompass transmission time (and the relevant time should include processing time).

An alternate solution using the remote-operation paradigm may look as follows:
/* at site 0*/

remote_object_table at site sitename:
operations
    update(input-> key: char, value: integer)
    read(input-> key: char, output-> value: integer)

read(some_key, some_value) if unfinished /* time-out action */

/* at site 1*/

data_object_table
representation array [1...n] of record:
    name: char
    value: integer

operation update(a_key: char, a_value: integer)
    /* searching and updating logic*/

operation read(a_key: char, a_value: integer)
    /* searching logic
     (perhaps shared with update*/

end data_object

The difference between the "update" and "read"
operations is that the former has been constructed without output parameters. This automatically tells the compiler that in the "read" case a send/receive pair has to be generated since a reply is expected. The expanded code takes a similar form as in the solution using send/receive primitives. This discipline corresponds to the (only!) discipline modeled by the call/return approach.

We have appended an "ii_unfinished" clause to the remote-operation in order to model the time-out condition. The important feature of this timing mechanism is that it gets integrated naturally into meaningful operations. The example shows the clause being associated with one remote-operation; however, one can easily extend it to a group of operations. In this case, the syntactic form resembles that of program structures for exceptional condition handling.

5.6.3 Multiple-request/ Multiple-reply

Another important communication discipline relates to multi-destination messages. The intent of a multi-destination message is to cause several operations on remote systems using only one invocation, e.g., a multiple request. As in the first discipline, there may be no response associated with such a "multiple operation".
Should the operation have the requirement of high reliability (e.g., every remote system should perform the operation), very reliable multi-destination protocols have to be used (the last example of this section illustrates this requirement). On the other hand, if a response is expected, there are some typical patterns of response that can be readily assisted by the language.

One case is a first-reply type of answer. This is a situation where any answer is equally acceptable; subsequent answers are of no interest. For example, if the table of the previous examples is duplicated at several sites, a "read" operation can be performed on all sites with a copy. The purpose of such interaction can be to speed up the computation (total delay is the delay of the first answer) and to increase the chances of receiving an answer at all (more sites have to be unreachable or dead in this case). A solution within the remote-operation paradigm (the solutions using send/receive primitives will not be shown since they are clearly awkward) can be as follows:
This solution is similar to the previous single-request/single-reply case, with three modifications: (1) at the site of the remote operation, we include the site names of the sites that contain copies of the remote object (alternatively, other "clues" can be given to the compiler in order to generate the appropriate send's and receive's for finding locations of the object); (2) the invocation always refers to all sites unless otherwise specified (e.g., "read" refers to all sites, whereas "sitename.read" refers to the "read" of site "sitename"); and (3) the first-reply pattern is recognized by the compiler by noticing that the return parameters of the invocation correspond to the output parameters of each single operation.

This example also shows the use of unreliable multi-destination protocols. In this case, the read command is directed to several sites without major requirements on
reliability (less overhead!). The reason is the nature of the operation, e.g., the operation finishes when the first answer arrives.

The other typical pattern of answers corresponds to a *multiple-reply-on-condition* type of answer. This is a situation where replies are sequentially examined until some condition is satisfied. For example, suppose we want to implement an "atomic operation" on multiple copies of a data object. An atomic operation means that either all the copies perform the operation or none perform it. A typical solution to this problem is to use the so-called "two-phase commit protocol" [LAM70, OWA78]. In short, this protocol first tries to reach a commit point (phase 1) where all copies have agreed to perform the operation. If some site refuses no operation is performed. If all sites agree the operation is performed at all sites (phase 2). A slight variation of such a protocol is shown below in a program using the remote-operation paradigm.
I at a site of a data object */

data_object  table  
representation....  
synchronization....  
/*other properties of object*/

operation  fake_update(.....,answ)  
/*store update in safe storage and answer*/

operation  do_fake_update(...)  
/*do actual update based upon  
info. previously stored in safe storage*/

operation  undo_fake_update(...)  
/*delete update from safe storage*/

/*at site of atomic operation*/

proc  atomic_update(...)  
remote_object  table  at  site....  
remote_operations  
do_fake_update(...)  persistent

answers:stream_of  answ:boolean
  :
  fake_update (... answers)
  do while (#_answers < total.#_answers)  
    if answ = false  then  undo_fake_update(...)  
      return;
      else  #_answers<-#_answers+1
    end
  /* local commit */
do_fake_update(...)  
return1
end

The first important point in this example is
the use of the "stream_of" data type (in a way similar to the one described in [BRY78]). Once we invoke a "fake_update" operation, the answers from several sites start arriving asynchronously and processing can start. If an answer is negative, no more answers from the stream are of interest (this can be signalled by a "flush" operation on streams). Every time a new element from the stream is referenced, the program is suspended if it has not arrived yet.

The use of multi-destination protocols is also shown in this example. The "fake_update" illustrates the use of a best-effort-to-deliver protocol. In this case, we would like to have a message exchange facility that makes sure that as many relevant sites as possible have a chance to produce an answer ("vote"). Clearly, it is beyond the scope of the protocol what the outcome of the answer is; however, if an unreliable multi-destination protocol is used, it is possible that positive answers cannot be given just because the request for answer ("voting") did not arrive.

This example also illustrates some interesting points about fault-tolerance. For the design of very robust software, it is desirable to specify within the definition of a data object some fault-tolerant properties such as what kind of storage is used for
storing the actual content of the object and reinitialization conditions after site failures. Another fault-tolerant feature relates to the reliability of remote-operations. In the example, the "do_fake_update" operation has the property of "persistence". This means that the operation finishes when it is guaranteed that eventually every relevant site will attempt to perform the operation (e.g., if a site with a copy of the data object has failed at the time of the invocation, the request for the operation will arrive anyway at the site as soon as it comes up). This kind of property can be offered by using a guarantee-to-deliver interprocess communication protocol.

5.7 Summary

In this chapter we have outlined some preliminary ideas on a distributed programming language. The major conclusions are (1) the domain of distributed programming is more complex than existing programming models; (2) we need a language for writing complex programs arising in "distributed software"; and (3) message exchange operations at the programming language level can make programs more obscure than necessary.
Notice that we have not implied that all distributed algorithms can be more naturally expressed without send/receive primitives. As a rule, the more complex the algorithm (i.e., more concurrency among remote program components, more reliability features incorporated in the algorithm) the more beneficial it is to get rid of low level primitives. Conversely, very simple and basic distributed algorithms can be expressed naturally using low-level primitives.

To support this basic observation, we have introduced the concept of remote-operations. We have shown how this concept models several communication disciplines found in distributed software. Furthermore, we have shown that this is a powerful way of using interprocess communication protocols, since it fits into other abstract tools of programming languages.
CHAPTER 6

DISTRIBUTED SYNCHRONIZATION

In the previous chapters we have attacked problems related to the exchange of multi-destination messages. In particular, we have proposed solutions to the problems of routing of multi-destination messages (Chapter 3), protocols for the exchange of multi-destination messages (Chapter 4), and abstract views and usage of multi-destination protocols (Chapter 5). The proposed mechanisms are the basis of a powerful interprocess communication facility for distributed systems.

Such facility, however, is merely concerned with "message passing" among a set of remote processes. The next level of interest deals with basic patterns of cooperation among a set of remote processes. As explained in Chapter 1, many computations that exhibit concurrency require harmonious cooperation, if the computations are to be
correct. The overall problem of enforcing an orderly cooperation is called synchronization. Most of the work in synchronization assumes a centralized model of computation [AND77]; our interest here is on synchronization for distributed models of computation, e.g., where the cooperating processes are remote.

The purpose of this chapter is to explore the nature of synchronization in distributed systems (distributed synchronization). In Section 1, we introduce some basic concepts. Section 2 examines in retrospective some centralized synchronization models, some characteristics of which help to clarify issues in the design of distributed synchronization models. In Section 3 we explain in detail important features of the problem of synchronization in distributed systems. Finally, Section 4 discusses possible extensions of a centralized synchronization model to a distributed system.

6.1 Fundamentals

A synchronization problem describes some ordering of events required by computations. For example, "mutual exclusion" problems are synchronization problems where a set of events cannot overlap in time. In "producer/consumer" problems, some events (producer events) can happen a fixed
number of times ahead of some other events (consumer events). There are many types and versions of synchronization problems. As we proceed through the thesis, we shall have occasions to indicate which synchronization problems are common to both centralized and distributed systems and which synchronization problems are new in distributed systems.

Any synchronization problem can be solved in a "self-contained" manner; that is, the computation is modified to enforce the order of events without any system help. For example, the solution to the "mutual exclusion" problem in centralized systems can be obtained using only STORE and LOAD operations [DIJ68]; similarly, it can be obtained in distributed systems using SEND and RECEIVE operations [FEL77]. However, such solutions are very elaborated and only apply to a particular problem. The alternative is to use a synchronization model.

A synchronization model is a system mechanism that helps computations to enforce the ordering of events necessary for their correct behavior. If a synchronization model is designed to facilitate synchronization of centralized computations, we shall call it a centralized synchronization model, or centralized synchronization mechanism. On the other hand, if a synchronization model is designed to facilitate synchronization of distributed
computations, we shall call it a distributed synchronization model, or distributed synchronization mechanism.

Typically, a synchronization model can be used to solve several types of synchronization problems. Some synchronization models are only "better" than others for some problems:

"I find that they (referring to several centralized synchronization models) are all equivalent in the sense that each of these can be used to solve arbitrarily scheduling problems from a logical point of view. But for a given problem, some of them lead to more complicated and inefficient programs than others. So they are clearly not equivalent from a practical point of view." [BRI73]

In general the following are some of the criteria that can be used to judge a synchronization model:

(1) Simple to Use. A synchronization model should help to produce a solution in a very natural way. Some synchronization models have very low-level interfaces (e.g., assembly-like), whereas some others have high-level interfaces (e.g., associated with a data type of a high-level language).

(2) Good Performance. Performance encompasses efficiency and reliability. If a synchronization model induces too much delay or cannot recover from some failures, it is of very little use.
(3) Suitable for Verification. We should be able to prove formally some assertions about the behavior of a synchronization model. This is important to help verification of the entire computation.

In the next section we shall examine in retrospective some centralized synchronization models, emphasizing in particular the first criterion. In centralized synchronization models performance is not a major issue; however, we shall learn in this chapter how performance becomes an important consideration in distributed systems. The area of verification of distributed computations is still in its infancy (see [LAM78a] for an example).

6.2 Centralized Synchronization: Retrospective

There has been many centralized synchronization models developed over the past 14 years. Certainly, there are many ways in which one might classify such models. Since these models are merely tools to solve synchronization problems, it is extremely important how such a tool is viewed and used by programmers. Thus, we shall classify them into two types according to how they are perceived by programmers: low-level interface models and structured interface models.
6.2.1 *Low-level Interface Models*

These models are characterized for being independent of programming languages. As a result, programmers use these models through very low-level interfaces (i.e., assembly-like). Typically, these models consist of an interface (primitives) for interacting with computations and an internal logic (Synchronization Control Program SCP) for actually executing the primitives. The SCP normally manipulates some synchronization objects (e.g., bits, variables, message buffers, etc.). These synchronization objects have to be powerful enough to capture relevant information about events. To solve a synchronization problem means that the initial program has been carefully augmented with interactions to (primitives of) the synchronization model. The problem-solver maps the semantics of events of interest into the synchronization objects.

The *semaphore synchronization model* (P/V model) [DIJ68] is perhaps the most widely known centralized synchronization model. Processes interact with the model by means of primitives (e.g., P and V). The synchronization objects are semaphores (i.e., variables which record the "happening" of events by increasing or decreasing their integer values). The SCP, which manipulates semaphores (and
other non-observable objects such as queues), includes the logic to update semaphores, to maintain queues for semaphores, to block and awaken processes, to guarantee mutual exclusion to the access of semaphores, to implement a "fair" scheduling policy, etc.

The message-buffer synchronization model [BR173] is another interesting (yet different) low-level interface model. Processes interact by means of primitives (e.g., SEND and RECEIVE). The "happening" of events is recorded by storing/retrieving messages to/from a buffer. The SCP includes the logic to move/remove messages to/from the buffer, to deliver messages in the proper order, to block and awaken processes, etc.

The previous description suggests that the buffer is a separate global entity. An alternative to this strategy is to associate message buffers with receivers. This minor change has the property that buffers belong to processes and, as a result, they are not separate global entities.

There are many other low-level centralized synchronization models. Some of those models are slight variations of others; however, all have the same pattern of low-level interfaces and a SCP manipulating some synchronization objects.
6.2.2 **Structured Interface Models**

These models are characterized for being embedded in a high-level (systems) programming language. The language has constructs to express concurrency, i.e., it is a concurrent programming language. Typically, a resource can be modeled as a user-defined data type in the language. Therefore, the problem of sharing a resource is equivalent in the language to the problem of sharing a user-defined data type. The main idea is then to localize synchronization in the shared data type, i.e., the shared resource. In doing so, the solution to a synchronization problem becomes highly structured.

The most common of these models is the type `Monitor` in Concurrent Pascal [HOA74, BRI75]. A resource shared by concurrent processes is simply an instance of a monitor type. The operations on the resource, executed as part of the processes, can be the procedures associated with the monitor. The execution of the monitor procedures is guaranteed to satisfy mutual exclusion. Furthermore, since it is possible that processes may want to be suspended during the execution of a monitor procedure, a "condition" type is also introduced. As a result, some of the burden of synchronization is automatically passed to the run-time environment of the language, e.g., blocking and awakening
processes, making sure that every monitor procedure has exclusive access to permanent data structures, etc. It is interesting to note that in order to implement these language features, the run-time environment has to use low-level interface models.

Another example of this type of models is Path Expressions [CAM74, HAB75a, HAB75b]. In this case, synchronization is completely localized in the type definition (as opposed to being spread throughout the data type operations as in the case of monitors). A path expression defines the permissible order of operations on a data type. It is possible to define through path expressions relations among events such as sequentaility, selection, repetition, conditionals, priorities, parallelism, etc. Thus, the programmer has only to understand how to map the synchronization specifications of the data object into a path expression, leaving to the compiler the job of generating the appropriate code to enforce the semantics of the path expression. Again, a low-level interface model is used in order to implement path expressions [CAM74].

6.3 Features of Distributed Synchronization

As we mentioned before, the computations of
interest are modeled as a set of cooperating processes. In a centralized system such processes coexist in a single processing system. The communication among processes is typically made by sharing main memory and is almost instantaneous. Furthermore, reliability is unimportant since a failure becomes so catastrophic that upon its occurrence the computation is aborted and started all over again.

In a distributed system some processes may be remote. "Remoteness" means that processes have to communicate via messages. Since messages travel through communication systems, the delay in process communication may be arbitrary. Furthermore, a distributed system has a nice property of "failure isolation". This means that one or even more processing systems may fail while the others remain up. Consequently, the computations and the synchronization models themselves have the potential for being very reliable.

From the above we can see that delays and reliability are two important characteristics of distributed systems. How will these characteristics influence synchronization?

6.3.1 Delays
The first implication of delay deals with efficiency. In a distributed system it is expensive to construct global states for (distributed) computations. By the time a global state is constructed (examined or updated) the computation may have already changed its state. Alternatively, "freezing" a computation may be unacceptable if delays are long.

Another implication of delay relates to the notion of time in the system. Time is important because synchronization problems are essentially problems of enforcing time relations among events. Thus, the system has to "understand" what it means for events to be "ordered" (e.g., when an event happens before another event). In a centralized system there is one physical time scale, e.g., the time defined by the system's clock. Furthermore, the absence of delays implies that recording the happening of events takes no time, e.g., the system can "know" about events as soon as they happen. In a distributed system, each processing system has its own physical time scale. Typically, it is undesirable that the system uses "absolute time" (1) for defining orders of events. Instead, orders

(1) this would be the time obtained by synchronizing perfectly the clocks of all processing systems and by having the clocks running at the same rate of some clock C. This clock C is trusted by all users as defining the "real" time. For practical reasons, this may be difficult to achieve.
are usually built based on the observation that, if process A sends a message to process B then the events in A prior to the transmission of the message happened before the events in B occurring after reception of the message. This property can be used to build logical clocks, partial orders, and total orders independent of physical time [LAM78b]. As a result of internal delays and internal definitions of time, two remote events may happen "simultaneously" as seen from the outside of the system; however, one may happen "after" another as seen by the system. Conversely, two remote events may happen one "after" the other as seen from the outside; however, they may be "concurrent" (e.g., neither one happened "before" the other) within the system.

Delays and time also influence notions of scheduling in distributed systems. In general, synchronization problems imply some kind of policy to be followed in order to handle simultaneous events. For example, if a computation allows concurrent "writes" to a file, the requests may be required to be served FIFO (First-In-First-Out). In a centralized system, we can use a synchronization model that helps to record the issuance of requests (e.g., before a "write" request a P-operation is executed, if we use the semaphore synchronization model). Since the model itself dispatches waiting processes in FIFO order, the solution works. More importantly, this solution
works because the time spent recording a "write" attempt is negligible in a centralized system.

In a distributed system where the requests and the file can be remote, this problem and its solution exhibit a different behavior. Suppose we solve the problem using a central site (say the file's site) for deciding which request can execute. In this case, we note the following:

(1) If the synchronization mechanism uses "absolute time", all requests should be timestamped and no request from a given site can proceed unless it is known at the central site that no earlier request from any other site has been made. Consequently, a long delay in making a request known to the central site affects other requests.

(2) If the synchronization mechanism at the central site simply serializes the requests in order of their arrivals, the requests may not be served FIFO. Although requests are served FIFO with respect to their arrivals at the central site, they are not necessarily FIFO with respect to their issuances due to internal delays.

Alternatively, we can solve this problem by using a synchronization mechanism that does not reside in a central site. For example, we may have a "token" (i.e., a special control message) circulating throughout the sites that can make requests. Before a request is executed, it has to wait
for the token. Once it gets the token, it executes and forwards the token. In this case there is no FIFO discipline with respect to the issuance of requests, nor is there FIFO discipline with respect to the arrival of requests!

6.3.2 Reliability.

The most interesting implication of reliability is that it affects the specification of synchronization problems. In a centralized system a synchronization problem is specified without concern to reliability. Synchronization problems in distributed systems reflect in their specification some behavior with respect to failures. Such behavior has to be typically captured in the form of assumptions or in terms of time. For example, let us take the problem of "mutual exclusion" proposed in [DIJ68]. (We shall examine this problem and its possible solutions in more detail in the next chapter, section 7.1.3.)

In its "centralized" version, this problem involves a number of concurrent processes, each one having "critical" and "non-critical" sections. The problem is to guarantee that at most one of the processes executes its critical section at a time. The solution must satisfy [AND77]:
(1) symmetry (no static priorities between processes); 
(2) independence of relative speeds of processes; 
(3) independence of any process stopping well outside its critical section; and 
(4) when two or more processes simultaneously try to enter their critical sections, the decision of which process will enter must always done in finite time.

The specification of this problem has to be interpreted with great care in distributed systems. If a processing system fails while a local process executes its critical section, the specification of the problem does not tell us what is to be done! One solution can be to assume that such situation never happens, or it happens so rarely that we can afford to start all over again. If this assumption is unacceptable (which is often the case for reliable systems), we have to make assumptions on the structure of the processes. In this case we can design the processes such that the critical section becomes an "atomic" operation (i.e., it can never be left on intermediate states; or similarly, it is either all done or not done at all).

Another implication of reliability relates to a new class of synchronization problems in distributed systems. Unlike centralized systems, in a distributed system it becomes attractive to duplicate remotely resources. The problems associated with an orderly modification of such duplicate resources are new synchronization problems exclusive for distributed systems.
6.4 A Distributed P/V Model

In this section we shall examine the major modifications necessary to implement a centralized synchronization model in a distributed system. In particular we take the P/V model as a specific example.

In a centralized system, this model has only one copy of its SCP and one copy of each semaphore. Since the concurrent processes in a distributed system can be remote, it is necessary to have a SCP at each remote site. The new synchronization model is then realized as a set of cooperating SCPs. There are basically two ways in which these SCPs can be designed: single semaphores vs. multiple semaphores.

6.4.1 Single Semaphores

This implementation strategy corresponds to having only one copy of each semaphore at some SCP. If the synchronization problem has the structure of a single-shared resource problem, it is clear that the semaphore should reside at the site of the resource. If the problem does not have such structure (e.g., the processes share several resources, or there is not shared resource), it becomes difficult to decide where to locate the semaphore.
The main steps for the implementation of the single semaphore solution are the following:

/* invocation */

if sem_is_local(s) then /* local invocation */
    if P then local_implem_of_P
    else local_implem_of_V
    wake_up_processes

else /* non-local invocation */
    if P then send ("P", "s") to owner_of(s)
    receive (answ) from owner_of(s)
    if has_to_wait(answ) then block_caller
    else send ("V", "s") to owner_of(s)

/* external requests */

receive(m) from any

case of action_in(m)

P:
    s ← s - 1
    if s < 0 then send ("wait", "s") to originator_of(m)
    else send ("go ahead", "s") to originator_of(m)

V:
    s ← s + 1
    wake_up_processes

Since there is only one copy of each semaphore, an
invocation of a P or V operation may be either local (i.e., the copy resides at the same site of the process invoking the operation) or non-local (i.e., the copy resides at a remote site). The problem of finding the site of a non-local semaphore can be solved by assuming that each SCP either knows in advance where each copy is kept or finds its location dynamically upon the first invocation. (Efficiency is more important than transparency at low-levels!) The site of a non-local semaphore is represented in the program as "owner_of(s)". Since each semaphore can potentially have a queue of processes from any site, the actual process id's associated with a semaphore at any point in time may be process id's of remote processes. Thus, a V operation (whether local or non-local) can potentially wake up remote processes. The function "wake_up" in the program captures this idea.

The performance of this implementation encompasses reliability and efficiency. Failures in the communication system can be handled at the process communication protocol level. In the worst case the "owner_of(s)" is unreachable; however, this is not a major problem because if the resource is located at the site of the semaphore, the resource itself is also unreachable. A more serious problem occurs if there is a failure at the resource site while a process is accessing the resource. The solution to this problem is
beyond the scope of the synchronization model (I). Unless this problem is solved by the processes themselves, it makes little sense to add robust features to the semaphores. Thus if the processes solve this problem (e.g., by implementing an "atomic" use of the resource), the semaphore can be made robust by storing the copy in safe storage.

The other dimension of performance is efficiency. Typically a computation interacts with a synchronization model because it is performing some actions on its own. If these actions generate messages, the overall message exchange (traffic of the computation plus traffic of the synchronization model) can be too large. This is a serious problem in low-bandwidth architectures (e.g., long-haul packet-switching architecture). One possible solution to this problem is to send (when possible) both types of traffic at the same time. For example, if remote processes of a distributed computation want to update a single shared file using this model, they do not have to exchange independently messages for P-operations and for the actual "update" operations. Instead, both types of operations can

(I) All these concurrent computations can be thought of as being composed of two parts: a part performing some synchronization actions and a part performing the actual data processing actions. For example, in Reader and Writers computations, one part decides when to read or when to write (synchronization) and another part actually reads and writes (data processing).
be sent in the same message. This solution requires that the P-operation does not suspend its caller.

6.4.2 Multiple Semaphores

In this implementation strategy there are several remote copies of each semaphore. One way for using these copies is the following:

\[
\begin{align*}
\text{if } P \text{ then } & \text{lock_ps_set}(s) \\
& \text{implementation_of_P} \\
& \text{unlock_ps_set}(s,\text{outcome}) \\
\text{if } V \text{ then } & \text{lock_ps_set}(s) \\
& \text{implementation_of_V} \\
& \text{unlock_ps_set}(s,\text{outcome})
\end{align*}
\]

In this case there is a copy of a semaphore at every site where there is a possible invocation. Thus, for each semaphore, there is an associated set of processing systems (called in the program "ps_set"). Since (by definition) Ps and Vs operations require mutual exclusion during the access of the semaphore, this implementation has to lock and unlock all copies for each operation. The operations "lock_ps_set" and "unlock_ps_set" perform these functions.

This solution has the worst possible performance. In terms of efficiency, every time a semaphore is referenced
all its copies have to be locked. This corresponds to the mutual exclusion of semaphores. Even if we assume a perfect (100% reliable) system, there is too much delay. As a result, the invocation should be directed towards a single site (e.g., the site of the only copy or the 'main' copy).

The other alternative is to use the copies designating one as the "master" and the others as "back-ups". Under this implementation, each invocation is always directed towards the master site (avoiding the problem of excessive delay incurred by locking); the master sites propagates the effect of the invocation in a reliable way to the back up copies; and, upon a master failure, some back up site can take over as a new master. A similar implementation has been suggested in [MEN79b] for a different problem.

6.5 Summary

In this chapter we have discussed in detail the major issues in the design of synchronization mechanisms for distributed systems. We have shown that distributed systems present an environment where delays, efficiency, and reliability are all interrelated. These characteristics affect the design of distributed synchronization models, e.g., a model that causes too much delay or cannot handle
communication or processing system failures is unacceptable. We have also shown the difficulties in implementing a conventional centralized synchronization model in a distributed system.
CHAPTER 7

DISTRIBUTED SYNCHRONIZATION MODELS

In the previous chapter we have introduced the problem of synchronization in distributed systems and we have shown possible implementations of a centralized synchronization model in a distributed system. We have learned, among other things, that the issues of reliability and efficiency play an important role in synchronization for distributed systems. In this chapter we focus our attention to specific distributed synchronization models.

This chapter is organized in two major parts. The first part discusses extensions to the implementation of a low-level distributed synchronization model (Eventcounts and Sequencers) initially suggested in [KAN77]. A key characteristics of this model is that it does not require mutual exclusion for the access of eventcounts. Unlike the primitives on semaphores in the P/V model, primitives on eventcounts can access simultaneously the same eventcount.
Thus, distant copies of eventcounts can be maintained without having to lock them for remote access. The initial proposal did not include a full general implementation of eventcounts for many cases in distributed systems and the problems of reliability, especially in the implementation of sequencers, were not considered at all. Our objective is then to propose a more complete and general implementation scheme.

The second part of this chapter investigates the feasibility of designing a high-level synchronization model for distributed systems. The model discussed gets buried in a distributed programming language, much like Monitors or Path Expressions get buried in a concurrent programming language for centralized systems.

7.1 A Low-level Model

The model of interest is the Eventcounts and Sequencers Model [KAN77, REE79].

7.1.1 Description

There are two types of synchronization objects in this model: eventcounts and sequencers. An eventcount is an object that is used to record the number of events that...
have occurred in a computation. An eventcount can be thought of as a non-decreasing integer variable. Three primitives are defined on each eventcount \( E \):

**advance** \((E)\) the process that executes this operation (called a "signaller") increases the value of \( E \) by one.

**await** \((E, v)\) the process that executes this operation (called an "observer") is suspended until the value of \( E \) is at least \( v \), i.e., at least \( v \) "advances" on \( E \) have been executed.

**v-read** \((E)\) the process that executes this operation (called an "observer") receives a value \( v \) that \( E \) must have had during the execution of this operation.

It is useful to add the following primitives based on the previous ones:

**await** \((E(1), v(1), \ldots, E(n), v(n))\) the process that executes this operation (called an "observer") is suspended until one or more of the eventcounts \( E \)'s become greater or equal to their corresponding values \( v \)'s.

**update** \((E, s)\) the process that executes this operation is simultaneously a "signaller" and "observer" (called "signaller-observer") since this operation is equivalent to

\[
\text{while } s \geq \text{read}(E) \text{ do advance}(E)
\]

A sequencer can also be thought of as a
non-decreasing integer. The only operation defined on a sequencer \( T \) is the "ticket" operation:

\[ t = \text{ticket}(T) \]

after returning its current value, its value is incremented by one, all of which occurs as a single atomic operation.

Several examples of the use of this model are given in the previously referenced papers. In general, any solution of synchronization problems using the P/V model can be rewritten using primitives of this model (e.g., different versions of Readers and Writers or different versions of Producer-Consumer problems). The reason for such equivalence is that the primitives of this model are somewhat at a lower level than P and V, i.e., one can build semaphores using eventcounts and sequencers.

7.1.2 Implementation of Eventcounts

The implementation of this model in a centralized system is discussed in detail in [REE79]. Here we are interested in their implementation for distributed systems. The basic idea for a simple case of implementation of eventcounts in distributed systems was presented in [KAN77]. We shall develop a series of increasingly more complex cases in order to reach a full general implementation. For
purposes of clarity we shall first ignore the problems of communication and processing system failures.

**Reliable System.** In a distributed system there are several cases to consider for implementing eventcounts.

**Case 1.** "Signallers in one site; observers in another site."

Suppose that the computation only executes in two sites. Furthermore, let us assume that: signallers are located in one site (the signaller-site); observers are located in the other site (the observer-site); there are local implementations of eventcounts on each site; and there is a process communication protocol between the two sites (i.e., there is a facility which takes care of passing messages between remote processes and it is invoked through `send` and `receive` primitives). Thus, the eventcount $E$ can be realized by two remote eventcounts $E(s)$ and $E(o)$, one on each site. (Subscript "s" indicates the signaller's site, and "o" indicates the observer's site.) Other auxiliary eventcounts $C(s)$, $T(o)$, and $C(o)$ are needed for house-keeping. In Figure 21 we illustrate the components of the Synchronization Control Programs (SCPs) at each site. Each SCP consists of several pieces of code (indicated by
Figure 21. Signallers in One Site/ Observers in Another Site
circles) which manipulate the local eventcounts (indicated by rectangles).

The following program segments show the logic of each SCP (we shall use the same conventions as in [KAN77]):

\[
\begin{align*}
m(s) &: \text{timestamp} = 0 \\
& \quad \text{value} = 0 \\
& \quad \text{cycle} \\
& \quad \text{await}(C(s), \text{timestamp} + 1, E(s), \text{value} + 1) \\
& \quad \text{timestamp} = \text{read}(C(s)) \\
& \quad \text{value} = \text{read}(E(s)) \\
& \quad \text{send}(m(o), \text{timestamp}, \text{value}) \\

m(o) &: \text{cycle} \\
& \quad \text{receive}(\text{timestamp}, \text{value}) \\
& \quad \text{update}(E(o), \text{value}) \\
& \quad \text{update}(T(o), \text{timestamp}) \\

t(s) &: \text{cycle} \\
& \quad \text{receive}(\text{time}) \\
& \quad \text{update}(C(s), \text{time}) \\

t(o) &: \text{time} = 0 \\
& \quad \text{cycle} \\
& \quad \text{await}(C(o), \text{time} + 1) \\
& \quad \text{time} = \text{read}(C(o)) \\
& \quad \text{send}(t(s), \text{time}) \\
\end{align*}
\]

Therefore,

\[
\begin{align*}
\text{proc} & \quad \text{advance}(E) \\
& \quad \text{advance}(E(s)) \\
\text{end}
\end{align*}
\]

and
proc read(E)
    advance(C(o))
    time=read(C(o))
    await(T(o), time)
    return(read(E(o)))
end

Intuitively the system works as follows: E(o) and T(o) are the delayed images of E(s) and C(s), respectively. The program segments m(s) and m(o) cooperate so that a change in E(s) or C(s) is propagated to E(o) or C(o), respectively. We cannot read E(s) directly because the latest advance to E(s) may be still propagating (and we would get a value of E that E had before the read instead of during the read). Thus, t(o) and t(s) are necessary to propagate a read attempt to the signaller and permit to pick up a value that E(s) has during the execution of the read operation.

Case 2. "Signallers in one site; observers in multiple remote sites."

In Figure 22 we show an implementation for this case. (The first subscript indicates as before whether eventcounts or processes belong to observers or signallers sites; the second subscript indicates the site number.) The table C of eventcounts keeps the number of read attempts from each remote observer, as seen by the signaller. This
Figure 22. Signallers in One Site/ Multiple Remote Observers
Table is necessary in order to distinguish each remote read attempt (i.e., it is necessary to have a table rather than a single eventcount); otherwise, each change of that single eventcount would have to be broadcast to all $m(o,i)$'s, thereby making indistinguishable at the observer site a current read attempt from other observer's read attempts. Alternatively, adding process id's to distinguish individual requests serves the same purpose of the table. The logic on the SCPs is changed as follows. (We only indicate the less obvious modifications.)

1) $m(s)$: when $E(s)$ changes, it sends a message to all $m(o,i)$'s; and when $C(s,i)$ changes, it sends a message to the corresponding $m(o,i)$.

2) $t(s)$: when it receives a message from $t(o,i)$, it only changes $C(o,i)$.

**Case 3.** "Signallers in multiple remote sites; observers in one site."

Figure 23 shows an implementation. The table $T$ of eventcounts keeps the number of read attempts which have actually picked up values of $E(s,i)$ from the signallers site
Figure 23: Observers in One site/Multiple Remote Signallers
to the observers site. The table $E$ of eventcounts keeps the value of each eventcount $E(s,i)$ as seen by the observers site. The processes on the SCPs are changed as follows:

1) $m(o), m(s,i)$: when $E(s,i)$ changes, $m(s,i)$ sends a message to the observers site so that $m(o)$ updates $E(o,i)$; and when $C(s,i)$ changes, $m(s,i)$ sends a message to the observers site so that $m(o)$ updates $T(o,i)$.

2) $t(o), t(s,i)$: when $C(o)$ changes, $t(o)$ sends a message to all $t(s,i)$'s.

3) the reading procedure:

```plaintext
proc read(E) 
  advance(C(o)) 
  time=read(C(o))
  parallel_and_all
  begin
    await(T(o,1),time) 
    temp1=read(E(o,1))
  end
  . 
  . 
  .
  begin
    await(T(o,N),time) 
    tempN=read(E(o,N))
  end
  end
  return(temp1+temp2+...+tempN)
end
```

Each statement within the "parallel_and_all" statement
can run concurrently; and the "parallel_and_all" statement ends when all concurrent statements finish.

**Case 4.** "Two remote sites can be signallers and observers simultaneously."

Figure 24 shows an implementation for this case. Each SCP has a signaller and observer part. Both operations, "advance" and "read", are defined at each site. The reading procedure at each site becomes:

```
proc read(E)
    localevcount = read(E(s))
    advance(C(o))
    time = read(C(o))
    await(T(o), time)
    return(localevcount + read(E(o)))
end
```

**Case 5.** "General case of N remote sites being signallers and observers simultaneously."

Figure 25 shows an implementation. The changes should be clear after the discussion of the previous cases. The tables do not need the ith entry. The reading procedure at the ith system is:
Figure 24. Two Sites Being Signallers and Observers Simultaneously
Figure 25. A General Case
Unreliable System. So far our distributed implementation of eventcounts has ignored the problem of failures. We shall now discuss how such implementation can be made robust with respect to several types of failures in the communication and processing systems.

Recall that the SCPs described above use a process communication protocol for exchanging information. Thus, their robustness with respect to communication system failures depends on which process communication protocol they use. At first, it seems that a very reliable protocol can be used; however, very reliable protocols are expensive and unless they are absolutely necessary they should not be used. Furthermore, synchronization is another layer of software; hence another possible source of overhead.
As pointed out in [KAN77] for the case of two remote SCPs, it is possible to use a protocol that does not detect duplicates and enforce sequencing. The reason is that messages exchanged by SCPs simply update eventcounts to new increasing values; therefore, if an eventcount is updated twice to the same value (duplication) no problem arises, or if a message does not actually update an eventcount to a new greater value (out of order) the message can be discarded. Similar arguments apply for the case of multiple remote SCPs, if unreliable multi-destination protocols (of the type designed in Chapter 4) are used.

Another problem that can be introduced by a failure in the communication system is communication system partition. This case seems impossible to deal with at the synchronization model level since by definition there is only one logical eventcount (e.g., a "read" operation is performed on one logical eventcount). Thus, operating with two or more logical eventcounts (one per partition) and trying to combine their values after two partitions merge, does not work.

Processing system failures are, however, easier to handle. If the system can distinguish between a processing system failure such as memory loss and a communication system partition, it is possible to maintain at each processing system a list of remote processing systems that
are reported down. In this case all "send" primitives only
direct messages towards the operational processing systems.
Also, the "await" primitives of the "read" procedures are
protected by a time-out interval. If the time-out interval
expires, the procedure can check whether the processing
system for which it is "awaiting for" is operational or not.
Finally, when a processing system comes up again, it can
resynchronize each eventcount by simply "reading" its value
(that is, collecting the values of the remote copies).

7.1.3 Implementation of Sequencers

As defined in the model, sequencers (or "tickets")
imply some form of mutual exclusion. Before we describe the
implementation of sequencers, it is important to understand
several facets of the problem of mutual exclusion.

In general there are two levels (or types) of
mutual exclusion in process synchronization: (1) mutually
exclusive access to synchronization objects, and (2)
mutually exclusive access to resources. The first level is
a characteristic of the synchronization model itself
(application-independent). By contrast, the second level
arises at the application level. Typically, mutual
exclusion at the application level is achieved by using a
synchronization mechanism that implements a very primitive
and efficient form of mutual exclusion while accessing its objects. Furthermore, in a distributed system it is possible to distinguish several cases of mutual exclusion at the application level:

(a) In one case the resource is not located at any processing system; rather all processing systems have "immediate" access to it. For example, consider a computation in a distributed system that allocates a band of the electromagnetic spectrum or a volume of airspace in an air traffic control system [FEL77]. Another example can be single-resource sharing problems where a resource is remote to all processes and the resource is located at a system (a node) where no synchronization logic can be implemented, e.g., a local distributed system where a file is located at a "file machine node" that only accepts and executes remote requests. The interesting feature of this type of problems is that a synchronization mechanism has to be located at the processing systems of the cooperating processes and cannot be implemented at the site (if any) of the resource.

(b) In another case the resource is located at some processing system. For example, consider a distributed computation that generates concurrent updates to a
table. In this case the table is located at some processing system and consequently, it is possible to design a synchronization mechanism that also resides at the resource's site.

(c) Finally, there is a case of several duplicated resources at different processing systems. For example, consider a distributed database system that supports duplication and allows update transactions. In this case the synchronization mechanism can be implemented at the resource sites, at the process sites, or at both types of sites.

We shall discuss mutual exclusion of types (b) and (c) later on (section 7.2.3). We now return to the implementation of sequencers. We shall see that the techniques used for implementing sequencers can also be used to implement mutual exclusion problems of type (a).

**Reliable System.** Sequencers are synchronization objects whose only defined operation implies mutual exclusion. From the above, it is clear that this mutual exclusion is of type (or level) 1. Its implementation can be done by having a "circulating message" over the set of remote processes that can execute ticket operations. The content of such a message is the value of the sequencer.
The "ticket" operation consists of waiting for the message, reading its value, incrementing it by one, and letting the message go to the next site.

Figure 26 shows an example of this technique. In this case there are six remote processes, each one at processing system PS(i). Notice that a slight modification of this technique can be used for implementing mutual exclusion of type (a). In such a new version, no numbers are needed inside the message since it is only necessary to be able to recognize the message (it looks more like a "token" or special control message); each process executes its critical section (i.e., the event that has to be mutually exclusive) after receiving the message; and the message is forwarded only after the execution of the critical section.

A brief analysis of this technique under error-free conditions illustrates some of its properties. Let us define:

- \( n \): number of remote processes requesting ticket operations
- \( c_d \): average message delay between two processing systems
- \( p_d \): average processing delay in updating the content of the circulating message (or in holding the control message)
- \( t(i) \): average delay between two local ticket requests
Figure 26. Implementation of Sequencers Using a Circulating Message
We can see that the maximum waiting time for a ticket operation is equal to \( n(c_d+p_d)-p_d \). In other words, every request is satisfied within a finite interval of time. The next question of interest, however, is how "fair" is this technique with respect to the allocation of ticket requests. Clearly, ticket requests are not satisfied in order of their requests, nor are they satisfied in order of their arrivals at a central site (there is no central site!). "Fairness" in this case depends on the stochastic nature of the requests, e.g., if all \( t(i) \)'s are less than \( (c_d+p_d) \) then ticket requests are always satisfied in order of their requests. Finally, if some \( t(i) \)'s are greater than \( n(c_d+p_d) \), it is possible that the message stops at processing systems where there are no outstanding ticket requests. (This is the case of a set of remote processes where the rate of progress of some processes is very slow, or some processes finish making their \( t(i) \)'s go to infinity.)

**Unreliable System.** Our previous discussion assumed an error-free system. However, if the control message is lost the entire computation stops!

There are basically two types of failures that can cause loss of the control message: communication system
failures and processing system failures. The first type of failure suggests the use of fairly reliable protocols. Fortunately, such protocols (with retransmission to recover from internal loss in the communication system, and with detection of duplicates to prevent multiple control messages) are available in any distributed system. A more difficult problem occurs if a processing system fails after it receives the message and before it forwards the message (e.g., while executing the ticket operation if the message is a sequencer, or while having the token if the message is only a control message).

One solution to this problem is to keep a copy of the message (recall that for sequencers there is a number to "remember") until the next processing system acknowledges a successful forward. (We shall call this technique the "successful forward technique"). For example, consider three consecutive processing systems in a virtual loop: \( i \), next\((i)\), and next\(\text{next}(i)\) \((i)\). When \( i \) forwards the message to next\((i)\), \( i \) keeps a copy of the message. Then, after some

\((i)\) The usage of "next\((i)\)" instead of "\(i+1\)" indicates that failures of processing systems at times when the control message is not "close" to them should be handled separately. In these cases, their failures can be detected by periodic "are you alive?" messages. Upon such failures, the processing system is automatically taken out from the virtual loop so that "next\((i)\)" is no longer the node in some preassigned static order.
time next(i) tries to forward the message to next(next(i)). If next(i) succeeds, it sends an acknowledgement back to i so that i can discard its copy. However, if next(i) fails before it successfully forwards the message, processing system i eventually times out and forwards its copy to next(next(i)). Figures 27 and 28 show how this technique works with no failures and with single processing system failures, respectively. Dotted lines indicate an acknowledgement at the process communication level (i.e., the message arrived successfully at its destination), whereas acknowledgements on solid lines mean "successful forward." Notice that the time-out interval should be greater than 3*c_d+p_d.

This is a simple technique that is robust with respect to some failures of processing systems. If two consecutive processing systems fail very close in time while the control message is passing by, this technique may lose the ticket (e.g., if next(i) fails before a successful forward, and i also fails before it times out). Also, if next(i) fails between the time it successfully forwards the ticket and it sends back an acknowledgement to i, this technique may generate two tickets. To avoid the latter problem, a ticket may carry a "hop count" (a number that is incremented each time the ticket is stopped) so that tickets with lower hop counts than the last seen by a processing
Figure 27. Error-free Case of the Successful Forward Technique

Figure 28. Single Processing System Failure Using the Successful Forward Technique
system can be discarded.

A nice property of this technique (although it still can lose the ticket very rarely) is that it can detect the loss of the control message very quickly. This is possible since additional messages (the acknowledgements) are exchanged under normal operation. Another technique (suggested in [LAN77]) can recover from multiple processing system failures and does not add extra messages during normal operation; however, it takes longer time to detect even a single processing system failure. To remedy this last problem, it is possible to combine both techniques as follows:

1. Each processing system sets a timer (to a value greater than the maximum waiting time) every time it successfully forwards the control message.
2. Each processing system keeps a copy of the last control message it has "seen."
3. Upon time-out, a processing system tells every other processing system that it has not "seen" the message.
4. Some time later after the time-outs, the processing system with the minimum (preassigned) number in the virtual loop generates the new control message.

Part 4 makes this technique time consuming. In fact, "some
time later" means that it takes 
(n-1)*c_{d}+(n-2)*p_{d}+c_{d}+\text{max\_waiting\_time} to regenerate a new control message. Fortunately, this method will only be used if two consecutive processing systems fail at a critical time (i.e., when the first technique does not work); therefore, very rarely the system will run this type of error recovery algorithm.

7.2 Towards A High-Level Model

The previous distributed implementation of Eventcounts and Sequencers can be used to solve several types of synchronization problems. However, the primitives of such a model are assembly-like (i.e., low-level). In the same way that low-level synchronization mechanisms in centralized systems evolved into high-level mechanisms, we can expect to find "structured" or high-level synchronization mechanisms for distributed systems. High-level synchronization mechanisms have been successfully designed for centralized systems partially because of a clear understanding on both synchronization and concurrent high-level languages in the context of centralized systems. Unfortunately, very little is known to date on both areas in the context of distributed systems so as to design a very powerful high-level model. Throughout the rest of this
chapter we will attempt to characterize how this problem can be attacked.

7.2.1 The Approach

Suppose we have an object-oriented language, much like CLU [LIS77] or ALPHARD [WUL76]. For our purposes, the important feature of the language is that it supports data abstractions, i.e., user-defined data types that encapsulate their representation and can only be accessed by their defined operations. In order to model synchronization problems, we allow objects to be shared by concurrent processes.

Any resource-sharing problem can now be modeled as a problem of sharing a data object in the language, where the different uses of the resource correspond to the defined operations on the data object. The extension to a distributed system consists of allowing remote data objects to be shared by remote processes, i.e., we now have a distributed programming language. In Chapter 5 we have discussed the communication features of such a language. We shall now discuss how synchronization tools could be built into such a language.

The basic idea of the "high-level" approach is that synchronization specifications are attached to the
definition of shared data objects and the resulting constraints are enforced by the type manager of such data objects. An important characteristic of this approach is that synchronization is entirely viewed at the resource-operation level. For example, when using low-level models, a programmer inserts primitives of these models into the actual code of the concurrent processes of its program; however, these primitives are different from (and in addition to) the actual operations on the shared resources. By contrast, when using a high-level model, the code of the concurrent processes is not altered (i.e., it only contains the meaningful operations on resources). In this case, the programmer only adds to the resources some constraints on the order of execution of their operations.

This high-level approach has two important advantages in distributed systems:

(1) It is very modular. By associating all synchronization with the resources, the synchronization mechanism is confined to the sites of the resources, thereby making easier the dynamic addition or deletion of processes at different sites.

(2) It can be very efficient. As mentioned earlier, all low-level mechanisms are based on the approach that before (and/or after) the actual operation on a shared
resource, a low-level primitive is executed. Even if the low-level model does not use locking on its own synchronization objects (as with eventcounts and sequencers), a substantial amount of bandwidth and time can be consumed when used in distributed systems. Thus, a model that avoids those extra execution of primitives and simply lets the operations compete with other concurrent operations at the sites of the resources, can save time and bandwidth.

7.2.2 The Specification Language

In order to attach synchronization specifications to the definition of shared data objects, we need a specification language that permits the description of orders commonly found in synchronization problems. Two non-procedural languages have been extensively studied for specifying centralized synchronization constraints: path expressions [CAM74] and predicate calculus [LAV78].

The issues of interest to us are: (1) which features of these languages should transcend to a specification language for distributed synchronization, and (2) which distributed synchronization features cannot be captured by these languages.
Common Requirements. Some synchronization constraints arise in centralized as well as in distributed systems. In general, the same types of synchronization constraints used in centralized systems can be used in a distributed system; however, these constraints by themselves are only useful to describe problems where there is only one remote shared-resource. Let us examine how these constraints (used in centralized systems) can be interpreted in a distributed system:

1) Local Exclusions. These constraints state that the execution of some local operations on the shared object cannot overlap in time. Local exclusions arise in a variety of ways: (a) several requests for the same operation cannot be executed simultaneously, and (b) the execution of different operations cannot overlap in time. For example consider the following synchronization specification in the language of path expressions:

path push + pop end

In this case the specification is attached to a stack data object and every operation is excluded with respect to itself (case a), i.e., two concurrent "push" operations are automatically serialized. Furthermore,
both operations are excluded (case b), i.e., if a "push" operation is concurrent with a "pop" operation they are automatically serialized. The important point to remember is that exclusions in specification languages for centralized synchronization become local exclusions in a specification language for distributed synchronization.

(2) Local Priorities. It is desirable to specify by means of the language different types of priorities, e.g., scheduling disciplines. However, the priorities captured by specification languages for centralized synchronization are somehow "local" priorities in a specification language for distributed synchronization. In other words, these are the priorities that can only be calculated based on local information. For example, a "first-come-first-served" discipline is to be interpreted as "first request arriving at the resource site is the first request served by the resource." As mentioned earlier, due to internal delays in the communication system, two operations may arrive at a resource site in a different order than their request order.

(3) Data-dependent Constraints on Local Data. For many problems different synchronization constraints or priorities apply to a shared object depending on some
local values. For example, in the language of path expressions we can have

\[ \text{path} \{ \text{top}=0; \text{push}, \text{top}=\text{max}; \text{pop}, \text{push} + \text{pop} \} \text{ and} \]

to indicate that only a "push" operation is allowed when the stack is empty (\(\text{top}=0\)); only a "pop" operation is allowed when the stack is full (\(\text{top}=\text{max}\)); and both operations are allowed (although mutually exclusive) otherwise. The important point to note is that "top" and "max" are local to the stack. Such a requirement on locality is extremely important if the object is remote.

**New Requirements.** Some synchronization constraints are exclusive for distributed systems. For example, only in a distributed system it is attractive to duplicate remotely resources in order to make them more accessible. As a result, the constraints on the operations on the resource apply to every copy. More importantly, there are new relationships among operations at different copies, e.g., an operation that "updates" a resource should be done at every copy.

Another constraint deals with scheduling. As mentioned earlier, in a distributed system there are delays between the time in which an operations is requested and the time the request arrives at the site of the resource. Thus,
if operations on a resource are to be performed in order of their requests, it is necessary to explicitly indicate so in the specification language.

7.2.3 Implementation Techniques

The problem of implementing different order of operations on a single resource (single data type) is similar in both centralized and distributed systems. On the other hand, the implementation of the new orders (the new requirements discussed in the previous section) is exclusive for distributed systems. Fortunately, there are many techniques to solve the synchronization problem associated with multiple copies of a resource [TH079, ELL78, JOH75, MEN79b].

One approach is to have a built-in solution for this problem in the language. For example, if we write a distributed program that uses multiple remote copies of a data object, it is possible that the compiler automatically generates code to ensure consistency among the copies of the object by using a "majority consensus algorithm" [TH079]. Unfortunately, experience with these algorithms suggests that each algorithm tends to perform best under different assumptions (e.g., whether there is total duplication, whether reliability is more important than efficiency,
whether broadcast is cheap, etc.). Thus, the implementation of the language becomes dependent on the system architecture.

Another approach is to find a minimum set of features for the language that allow writing different solutions to the problem. Thus, this approach suggests to have sufficient expressive power with few built-in synchronization mechanisms. Unfortunately, very little is known about the total space of synchronization problems in distributed systems in order to make reasonable judgements on the expressive power of synchronization features.

7.3 Summary

In this chapter we have described in detail how to extend the implementation of the Eventcount and Sequencer model for distributed systems. We have also shown how to make the model fault-tolerant to communication and processing system failures. We pointed out that the major disadvantage of using low-level synchronization models is their potential for introducing inefficiencies (e.g., in terms of extra number of messages). We examined in detail the efficiency and reliability of the implementation of sequencers.

We also discussed some common problems in
specifying and implementing high-level synchronization mechanisms for both centralized and distributed systems. The new problems arising from distributed systems are similar to concurrency control problems for distributed databases.
CONCLUSIONS AND FURTHER RESEARCH

In this chapter we shall summarize the main results of our research. We shall also indicate some topics for further research.

8.1 Summary and Conclusions

A very important result of our research is that in order to design a coherent distributed system it is necessary to provide an integrated hardware and mostly software support at many levels of the system. Given the "hardware of a distributed system", that is, any basic local or long-haul computer-network architecture, it is possible to redesign a high-level layer of system software (such as -
file system) that implements automatic resource-sharing. However, this approach is highly inefficient because it builds "on top" of conventional computer network software. Distributed software has different requirements than conventional centralized or computer network software. In particular, distributed software exhibits the property of cooperation among multiple remote software components and has important reliability requirements. In this thesis we have shown how to design some mechanisms such as interprocess communication and synchronization in order to effectively support distributed software.

Many times a process of a distributed algorithm sends a message to several remote processes. One reason for such type of cooperation is to exploit the inherent parallelism in a distributed system. At a very low-level, this means that the communication subsystem of a distributed system has to support efficiently the exchange of multi-destination messages. Thus, we have developed several methods to route efficiently multi-destination messages in local distributed systems. We have shown how these solutions differ from existing multi-destination routing algorithms proposed for long-haul architectures. For local architectures that have conventional addressing recognition mechanisms (that is, either point-to-point or full-broadcast messages are recognized), we have shown how to route
multi-destination messages. The most interesting result was
the case of single and double loops, where a
single-addressed message can be copied at intermediate
nodes between its source and its destination, thereby
improving the performance over the other two alternatives.
Furthermore, we have suggested how this can be implemented
in loops with minimum changes on existing or proposed
implementations.

Multi-destination routing was one dimension of the
problem of interprocess communication, i.e., how to direct
efficiently a message from one process to several remote
processes. The next dimension of the problem was to add
several reliability levels to the exchange of such
multi-destination messages, i.e., the design of several
levels of multi-destination process communication protocols.
Thus, we have designed three levels of protocols (called n-p
protocols): unreliable protocols, best-effort-to-deliver
protocols, and guarantee-to-deliver protocols.

The first level provides minimum reliability,
e.g., it only discards altered messages at a destination.
The functions of these protocols (addressing and interfacing
to routing) are common to the exchange of any
multi-destination message. Making such functions a separate
protocol level allows algorithms to use a minimum overhead
facility when they can implement (if needed) some
reliability features.

The second level provides more reliability to the expense of some additional overhead. Best-effort-to-deliver protocols try to make sure that as many destinations as possible receive a multi-destination message. This level is important since some destinations may be unavailable or may fail during the exchange of a multi-destination message. We have shown how the processing systems can maintain control information (used to detect duplicates) consistently despite of their own failures. Guarantee-to-deliver protocols are another type of interprocess communication, even more reliable than the previous ones. This level ensures that every destination will receive the message as soon as it becomes available. We have shown how to design such protocols using very reliable storage.

To complete our discussion of interprocess communication, we have moved to the domain of programming languages to show how such protocols can be naturally integrated into the novel concepts of abstract programming. In doing so, we have found a new programming domain that we called distributed programming. We have shown how this domain differs from conventional programming models. The proposed communication features of a distributed programming language (e.g., remote-operations) constitute our answer to a structured use of the interprocess communication
architecture developed earlier.

Interprocess communication is the mechanism for "passing efficiently messages" among processes of distributed computations. Another important system mechanism that we have studied is the enforcement of different orders in distributed computations. We have discussed in detail the unique features of distributed synchronization. In particular, we have shown how the notions of system states, time, scheduling, and reliability, are all different in a distributed system. We have also shown the difficulties in extending conventional centralized synchronization models to distributed systems.

In the final part, we have concentrated on specific synchronization mechanisms for distributed systems. The implementation of Eventcounts and Sequencers was extended to handle more general cases in a reliable fashion. Also the problem of mutual exclusion of remote processes was shown to have a similar solution to the implementation of sequencers. Finally, the problem of designing a high-level distributed synchronization model was discussed in some length.

8.2 Further Research

We believe that there are many interesting
topics for further investigation:

1) Multi-destination Routing Algorithms. There are other architectures that we did not mention in this thesis and that can be investigated in this respect. In particular, architectures that are a combination of other architectures will be very common in the near future. How can we design multi-destination routing algorithms for such mixture of, say, several local and long-haul communication systems?

2) Multi-destination Protocols. We assumed a rather static assignment of ports for association groups. Can we build association groups dynamically and in a reliable fashion? More importantly, are there other levels of reliability needed in multi-destination protocols?

3) Distributed Programming Languages. We discussed the communication features of a distributed programming language. However, more research is needed in order to design a coherent set of fault-tolerant, timing, synchronization, and protection features for such languages.

The most obvious "next-step" is, however, the implementation of the ideas developed in this thesis. Only by actual experimentation in a running system can we
ultimately judge their impact. Nevertheless, we expect that our view of the problems and their solutions will contribute to clarify the poorly understood field of distributed systems.
BIBLIOGRAPHY


FEL79 J. A. Feldman, "High Level Programming for


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publication Details</th>
</tr>
</thead>
</table>
B. Liskov, et. al., "Abstraction Mechanisms in CLU," CACM, Vol. 20, No. 8, August 1977, pp. 584-


