

A PROPOSED IMPLEMENTATION OF COMMUNICATION PRIMITIVES
IN ADA AND CHILL FOR DISTRIBUTED SYSTEMS

by

JOHN WILLIAM UNGER

A. B. Rutgers University, 1978

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree


MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:


Major Professor

LD
2068
.T4
CMSC
1988
U54
C.2

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION	1-1
1.1 Standard Communication Methods	1-2
1.1.1 Procedure Calls	1-3
1.1.2 Message Passing	1-3
1.1.3 Remote Procedure Calls	1-7
1.2 Summary	1-9
CHAPTER 2 - SURVEY OF COMMUNICATIONS PRIMITIVES	2-1
2.1 Ada	2-1
2.1.1 The Ada Rendezvous	2-2
2.1.2 The select statement	2-3
2.1.3 The delay Statement	2-5
2.1.4 Communication Considerations	2-6
2.2 Concurrent C	2-6
2.2.1 The Accept Statement	2-7
2.2.2 Timed Transaction Calls	2-8
2.2.3 Communication Considerations	2-9
2.3 CHILL	2-9
2.3.1 Buffers	2-9
2.3.2 Signals	2-12
2.3.3 Communications Requirements	2-14
2.4 Summary	2-15
CHAPTER 3 - DESCRIPTION OF A SUPPORTING COMMUNICATION SYSTEM	3-1
3.1 Specification of System	3-1
3.1.1 Network Topology	3-3
3.2 Service Queues	3-4
3.2.1 Characteristics	3-4
3.3 Message Content	3-7
3.4 Major Software Components	3-8
3.4.1 Device Driver	3-8
3.4.2 Network Server	3-10
3.4.3 Application Interface	3-10
3.5 Sending Messages	3-10
3.5.1 Logical Name to Physical Name Mapping	3-11
3.6 Receiving Messages	3-12
3.6.1 Asynchronous Receive	3-15
3.6.2 Synchronous Receive	3-15
3.6.3 Receive Immediately	3-16
3.7 Comparison of Communication System	3-17
3.7.1 Independence of Transmission Media	3-18
3.7.2 End-to-End Data Transfer	3-19
3.7.3 Transparent Data Transfer	3-20
3.7.4 Quality of Service Selection	3-20
3.7.5 User Addressing	3-23
3.8 Summary	3-23

CHAPTER 4 - IMPLEMENTATION OF COMMUNICATION PRIMITIVES	4-1
4.1 Hierarchy of Layers	4-1
4.1.1 Assumptions	4-4
4.2 Language Interface Components	4-5
4.2.1 Run-time Routines	4-6
4.2.2 Directory Server	4-8
4.2.3 Error Handling	4-11
4.2.4 State Diagrams and Transition Tables	4-12
4.3 Ada	4-14
4.3.1 Calling an Entry Point	4-14
4.3.2 Executing Accept Statements	4-18
4.4 Concurrent C	4-20
4.5 CHILL	4-21
4.5.1 Sending Buffer or Signal Messages	4-23
4.5.2 Receiving Buffer or Signal Messages	4-26
4.6 Summary	4-27
CHAPTER 5 - CONCLUSIONS	5-1
5.1 Usefulness of State Diagrams	5-1
5.2 Impacts on Language Interface	5-3
REFERENCES	6-1
ADDITIONAL DETAILS ON THE COMMUNICATION SYSTEM	C-1
C.1 Major Software Components	C-1
C.1.1 Device Driver	C-1
C.1.2 Network Server	C-2
C.2 Miscellaneous Operations	C-4
C.2.1 Empty Service Queue of Messages	C-4
C.2.2 Cancel I/O for a Service Queue	C-5

LIST OF FIGURES

Figure 2-1. Syntax of Ada Accept Statement	2-3
Figure 2-2. Syntax of Ada Select Statement	2-4
Figure 2-3. Example of Ada When Clause	2-4
Figure 2-4. Example of Ada Count Attribute Usage	2-5
Figure 2-5. Example of Ada Delay Statement	2-5
Figure 2-6. Example of Concurrent C Syntax for Transactions	2-7
Figure 2-7. Syntax of Concurrent C Accept Statement	2-8
Figure 2-8. Example of a CHILL Send Buffer Statement	2-11
Figure 2-9. Example of CHILL Receive Expression Statement	2-11
Figure 2-10. Example of CHILL Receive Case Action Statement	2-12
Figure 3-1. Communication Network Topology	3-6
Figure 3-2. Communication System Components Interconnection	3-9
Figure 3-3. Format of SEND Operation	3-11
Figure 3-4. Format of DELETE_BUF Operation	3-14
Figure 3-5. Format of Asynchronous Receive Operation	3-16
Figure 3-6. Format of the Synchronous Receive Operation	3-17
Figure 3-7. Format of the Receive Immediately Operation	3-17
Figure 4-1. Layers of Communication	4-3
Figure A-1. State Diagram for Ada Accept Select Statement	A-1
Figure A-2. State Diagram for Ada Entry Call Select Statement	A-4
Figure A-3. Composite State Diagram for Ada Rendezvous	A-7
Figure B-1. State Diagram for CHILL Send Statements	B-1
Figure B-2. State Diagram for CHILL Receive Statements	B-4
Figure B-3. Composite State Diagram for CHILL Communications	B-7
Figure C-1. DRAIN_QUE Call Format	C-5
Figure C-2. Format of CANCEL_IO Operation	C-6

LIST OF TABLES

Table 3-1. Processor Logical Names	3-13
Table 4-1. The Three Service Layers	4-2
Table A-1. State Transition Table for Ada Accept Select Statement	A-2
Table A-1. State Transition Table for Ada Accept Select Statement (Continued)	A-3
Table A-2. State Transition Table for Ada Entry Call Select Statement A-5	-
Table A-2. State Transition Table for Ada Entry Call Select Statement (Continued)	A-6
Table B-1. State Transition Table for CHILL Send Statements	B-2
Table B-1. State Transition Table for CHILL Send Statements (Continued)	B-3
Table B-2. State Transition Table for CHILL Receive Statements	B-5
Table B-2. State Transition Table for CHILL Receive Statements (Continued)	B-6

ACKNOWLEDGEMENTS

A number of people have helped me immensely in the development of this thesis.

I wish to thank my major professor, Rich McBride, for his assistance and guidance in helping me to concentrate on the issues involved with the topic.

I would also like to acknowledge the assistance provided by my supervisor, Tom Ely, who kept prodding me to complete the thesis, and allowed me to disappear for five weeks to do it.

I especially wish to acknowledge the support I received from my wife, Diane, during my work on this thesis.

CHAPTER 1 - INTRODUCTION

This thesis examines different ways that processes can communicate. In particular, communications between distributed processes will be studied. It also examines the communication primitives that are available in some standard languages, namely AdaTM, Concurrent C and CHILL. A communication system that has been developed is also presented in the paper. This particular communication system is used because it is familiar to the author. Also, it provides facilities commonly found in communication systems for real-time, distributed computing environments. The functions of the communication system will be presented in detail.

The objective of this paper is to show how the facilities of the communication system can be utilized to support the communication primitives of the aforementioned languages. The additional functionality that would be necessary to map the communication primitives of the languages into the services provided by the communication system will be described.

This chapter discusses the communications mechanisms that are commonly employed between processes. Chapter 2 presents the standard language communication primitives, some of which are commonly available in languages that use two of the mechanisms, namely message passing and remote procedure calls. Chapter 3 describes the communications system. It presents the capabilities of the system. Chapter 4 presents enhancements that would need to be developed to use the communication system to support the standard language primitives. Chapter 5 provides the conclusions, showing

Ada is a registered trademark of the U. S. Department of Defense, Joint Ada Project.

that the communication system can be used to provide the mechanisms for communications.

1.1 Standard Communication Methods

For the purposes of this paper, a distributed computing environment is defined to mean a collection of, possibly heterogeneous, processors. The processors are connected through a communications medium. A process can be understood as the execution of a program, and its associated data structures. Processes can run on different processors.

In a distributed computing environment, processes need to communicate. The reasons for the communication can be varied, such as:

- the processes need to synchronize activities;
- a client process requests service from a server process;
- or two cooperating processes need to pass information.

These needs have been widely discussed in the literature dealing with concurrent processes and distributed processing. In particular, see [Andrews 83], [Hoare 78], and [Brinch Hansen 77].

Different methods are available to facilitate the communications between concurrent processes that share common memory. These techniques include the use of procedure calls, shared variables and monitors.

For distributed processes that do not share memory, only message passing is a viable alternative. Remote procedure calls and monitors are possible, but rely on an underlying communications mechanism which utilizes message passing.

The following sections describe the characteristics of the communications mechanisms. The advantages and disadvantages of each mechanism for distributed processing are

presented.

1.1.1 Procedure Calls

Procedure calls are a standard mechanism used to communicate between procedures in a typical sequential process. This mechanism involves passing parameters to the called procedure, which will act upon the parameters and return to the caller. A result may or may not be passed back to the caller.

The caller is blocked while the called procedure is executing. The caller expects the called procedure to return to it when it has completed its operation. If the called procedure fails, the caller expects to be terminated, or that an exception handling procedure will be invoked, if provided by the language.

Communication between the calling and called procedures is accomplished by means of the passed parameters, as well as possibly by global variables. Because it is assumed that shared memory is available, the called procedure can affect the global variables directly.

Because shared memory is not necessarily available in a distributed system, traditional procedure calls cannot be used for communication. However, because the semantics of procedure calling is familiar to programmers, it would be beneficial to maintain a semblance of it for distributed computing.

Remote procedure calling has been proposed as a mechanism for distributed systems [Sloman 87], because it attempts to maintain a close similarity to traditional procedure calling. It will be discussed in detail in section 1.1.3.

1.1.2 Message Passing

In a distributed system, when processes need to communicate, they must pass messages. This action implies that a message handling, or communication, system is implemented to support this need. In this paper, the term *communication system* will be used to indicate

this facility.

Message passing involves having the caller construct a message, and send it to the intended receiver. This operation is typically called a *send*. The receiver must actively request a message. This is identified as a *receive* operation.

According to [Peterson 83], there are a number of issues that must be addressed in a communication system. Some of the more interesting issues are:

- How do processes identify each other for communication? The alternatives are direct or indirect naming.
- Where are messages sent? They may be either sent directly to the receiving process or stored in buffers.
- Is the communication symmetric or asymmetric?

The following subsections will discuss the issues in more detail.

1.1.2.1 Process Identification When *direct naming* is used, each process explicitly identifies the process with which it intends to communicate. This requires that each process knows the name of its partner. This is called *symmetric* communication. If the name of either process changes, the other one must know about it, and make the appropriate correction.

If the sender identifies the receiver explicitly, while the receiver will accept a message from any sender, the communication is called *asymmetric*.

When *indirect naming* is used, the messages are placed in a buffer or global location, which may be called a *mailbox* [Gelertner 82] and [Kenah 88], a *link* [Solomon 79] or *port* [Balzer 71]. Each process does not know the identity of the other, but must agree upon the name of the global location. In this situation, neither process needs to know the

name of its counterpart before a message transmission. In some implementations, multiple processes may be attempting to receive or send through the same buffer [Kenah 88].

1.1.2.2 Buffering Issues Either buffers are provided to allow the communicating processes some level of asynchrony, or no buffers are used.

If buffering is allowed, the sender can resume execution as soon as its message has been copied into the buffer. The buffering may be either bounded or unbounded. If the buffering is bounded, a sender is blocked when buffers are not available, until one becomes available. True unbounded buffers are unrealistic because there is a limit on the amount of physical storage available in any actual system. An assumption can be made that the storage space available is unlimited because it should not be exhausted in normal usage.

If no buffers are used, the communicating processes must be halted while the message transfer is being completed to prevent the message from becoming corrupted, because the message is copied directly from the sender's storage area to the receiver's. If either process is allowed to execute while the transfer is occurring, it may inadvertently write over the storage area for the message. An advantage of no buffering is that the sender receives an implicit acknowledgement that the message has been received, when it continues execution.

1.1.2.3 Synchronization of the Communications Operations Either the sender or receiver or both may be blocked for the duration of the communication. The different situations are examined in the following subsections.

1.1.2.3.1 Asynchronous Send When the sender is unblocked as soon as the message is queued by the communication system, it is referred to as an *asynchronous send*. This

technique cannot be allowed if no buffering is offered by the communication system.

In this case, the sender does not know if the message has been received. If it is necessary for the sender to confirm that the message had indeed been received, it would have to request an acknowledgement from the receiver or the communication system.

This method allows for more concurrency between processes, and incurs little overhead from the communication system. It puts a greater burden on the applications programmer to ensure reliable communications.

1.1.2.3.2 Synchronous Send In a *synchronous send* operation, the sender will be blocked until the receiver actually receives the message. In this way, the sender is assured that the message was received.

While the sender is blocked, concurrency does not occur. This defeats one purpose of distributed processing.

1.1.2.3.3 Asynchronous Receive As with *asynchronous send*, an *asynchronous receive* operation allows the caller to continue execution while the operation is still outstanding. It allows greater concurrency of activities within the receiver.

The receiving process will either be informed through an interrupt, or have to periodically poll to determine when a message is received. When it determines that a message has been received, it can then process the message.

As with *asynchronous send*, this method requires greater care from the application programmer to ensure the correct operation of the process. Some communication systems do not provide this type of receive operation, because either an interrupt service is not available from the operating system, or the overhead of polling is too great for the communication system. Many other communication systems only implement synchronous receive operations, because a receiver process normally only waits for messages, and

processes a message before attempting to receive another one. This is the normal paradigm for server processes.

The VMS operating system [Kenah 88] provides interrupt services to notify processes that messages are available. These services are used in the communication system presented in Chapter Three.

1.1.2.3.4 Synchronous Receive In a *synchronous receive* operation, the receiver is blocked until a message has been delivered to it. This can be used for synchronization as well as communication.

If a message does not arrive, the process could be blocked forever. To prevent this from happening in an unreliable communication system, the receiver may provide a timeout interval.

1.1.3 Remote Procedure Calls

Remote procedure calls are similar to normal procedure calls, except that the called procedure is not in the same process as the caller. It is even possible that it is not on the same processor. But the semantics of a remote procedure call are the same as for a normal procedure call, i.e. the caller is blocked until the called procedure performs the requested operation, and returns.

This action closely models the client/server relationship, where the client calls the server, and is blocked until the server performs the requested operation, and the results are returned.

Remote procedure calls in a distributed system imply the existence of a communication system that can transmit the requests and responses. In a distributed environment, a message passing facility would be necessary to meet this requirement.

1.1.3.1 Process Instantiation Because the process that executes the specified procedure is remote, it either must exist before the call is made, or it must be created.

A separate process that exists to service all requests of a certain type will serialize them, and thereby reduce the amount of concurrency possible. Ada is a language in which the remote process exists prior to the execution of the call. Synchronizing Resources (SR) [Andrews 82] allows for processes that wait for requests and then handles them.

In ARGUS [Liskov 82], handlers are remote processes that are created to service the call. Distributed Processes [Brinch Hansen 78] also provides separate processes for each invocation. Separate processes allow for more concurrency than would be possible with a single server process. If common data is accessed by the concurrent server processes, the servers must synchronize their accesses to the common data.

1.1.3.2 Remote Procedure Call Semantics Because the processes are communicating through a potentially unreliable communication system, there is a possibility of messages being lost. As a result, the call may be serviced by the called process more than once. Remote procedure call semantics can be defined by one of the following two types.

1. *Exactly once semantics*

The called procedure will be executed exactly once, and the results will be returned to the caller only once. This implies that duplicate messages will be eliminated by the communication system.

2. *At least once semantics*

The called procedure is executed at least once.

The first type of semantics is more ideal because it correctly emulates the normal, sequential procedure call. It is also the most difficult to implement, and incurs a higher overhead, because the communication system needs to be more concerned about the

successful transmission of messages. It implies a reliable communication system, that will guarantee message delivery, despite failures in transmission.

1.2 Summary

This chapter has discussed the need for communications in distributed systems. Different mechanisms for communication between processes have been presented, and their advantages and disadvantages for use in distributed systems have been discussed.

The next chapter will present the communication primitives available in some languages, in preparation for a discussion of a proposed implementation of these communication primitives in a distributed system.

CHAPTER 2 - SURVEY OF COMMUNICATIONS PRIMITIVES

This chapter examines the language constructs that are available in Ada, Concurrent C and CHILL for interprocess communication. The semantics of each construct will be presented. Issues that arise as a result of using the constructs in a distributed computing environment are discussed for each language.

All of the languages discussed provide facilities for concurrent processing, but some of these facilities do not lend themselves to distributed processing, especially when reliability and real-time processing are important considerations.

The constructs available in Ada for interprocess communication are presented in Section 2.1. The constructs defined in Concurrent C are discussed in Section 2.2, while Section 2.3 provides a description of the constructs in CHILL.

2.1 Ada

Ada is a language that was developed in response to the needs of the U. S. Department of Defense. A language was desired which would be able to be used for any defense project. Ada was designed to be employed in systems as diverse as payroll systems and real-time flight control systems in fighter aircraft. The goal of Ada was to provide a single language that all programmers would use, and reduce the problem of maintenance of programs that were written in little known, less maintainable languages.

As a result of these goals, Ada has become a large language with many different constructs. Some of the constructs, such as the *rendezvous*, utilize new concepts that did not exist previously in other languages.

The part of the language that is of interest in this paper is the tasking model for

concurrency, and in particular the *rendezvous* mechanism for synchronization and communication. A task is similar to the concept of a process in other languages. It is a sequential program part that may execute concurrently with other tasks.

The following subsection will discuss the *rendezvous*, and its implications in a distributed system.

2.1.1 The Ada Rendezvous

Communication between tasks is accomplished primarily by means of the rendezvous mechanism. In its simplest form, two tasks must be willing to communicate for this event to occur, hence the name *rendezvous*. If only one task is ready, it is blocked awaiting the other one. A *rendezvous* can be considered a remote procedure call in which the caller may be delayed longer than expected.

There are variations on the mechanism to allow either side to discontinue waiting for the *rendezvous*.

Data is passed from the caller to the called task by the parameters in the entry call. The called task can pass information back to the caller in the parameters from the call that are specified as *out* or *in out* in the entry definition.

A task initiates a *rendezvous* by calling an entry declared in another task. The entry is declared in a task specification in the same manner as a procedure in a package specification. An entry call takes the following form:

```
task_name.entry_name( actual_parameters );
```

The called task must execute or be waiting on an **accept** statement within the entry to have the rendezvous continue. The syntax of an **accept** is shown in Figure 2-1.

```
accept entry_name( formal_parameters ) do
    statements
end entry_name;
```

Figure 2-1. Syntax of Ada Accept Statement

Any sequence of statements may be executed in the body of an **accept**, including other **accept** statements.

The rendezvous is complete when the end of the **accept** block is reached. At that time, the caller is allowed to continue execution.

The naming in a rendezvous is *asymmetric*, because the caller must explicitly identify the task with which it is to rendezvous, whereas, the called task does not know which task called it.

2.1.2 The select statement

The **select** statement allows alternative blocks of statements to execute. Only one of the alternatives will execute each time the **select** statement is encountered.

The syntax of the **select** statement appears in Figure 2-2.

The **or** and **else** alternatives are optional. If the keyword **when** occurs in any alternative, the alternative is considered conditional. Otherwise, it is considered to be unconditional, and is available for execution at any time. The value of the conditional expression in a **when** clause determines whether the associated block is ready for execution. If the expression is true, the alternative is considered *open*. A **when** clause is referred to as a *guard*.

An example of the **when** clause appears in Figure 2-3.

```

select
    .
    .
    .
or
    .
    .
    .
else
    .
    .
    .
end select

```

Figure 2-2. Syntax of Ada Select Statement

```

when I < J =>
  accept read(file:in filename, record[I]: out rectype) do
    -- read a record from the file
    -- and put it in the record buffer.
  end read;

```

Figure 2-3. Example of Ada When Clause

Any of the open alternatives may execute. The order of execution is non-deterministic, and only needs to be fair, so no task is needlessly blocked.

There is also an attribute, called the **count**, associated with each entry that indicates the number of outstanding requests. This attribute can be used within the **when** clause to provide priority for specific entries.

An example is shown in Figure 2-4.

If none of the alternatives is open, the **else** block will execute immediately.

```

when entry1'count = 0 =>
  accept entry2(. . .)
  .
  .
  .
end entry2;

```

Figure 2-4. Example of Ada Count Attribute Usage

2.1.3 The delay Statement

The *delay* statement is used to suspend a task. It can be used to specify a timeout value in a task that is waiting for a rendezvous.

An example of a call to an entry, with an alternative **delay** statement is shown in Figure 2-5. The caller will not be delayed indefinitely if the designated *entry* is not ready for a rendezvous.

```

select
  task.entry1(arg1);
or
  delay 15.0;
  -- Statements are placed here to
  -- respond to the delay.
end select;

```

Figure 2-5. Example of Ada Delay Statement

In this instance, the caller will either be able to rendezvous with *task.entry* within 15 seconds, or the statements after the **delay** statement will be executed. It must be noted that the delay is only until the rendezvous starts. Once the rendezvous begins to occur, the **delay** is canceled. The caller cannot prevent being hung indefinitely if the called task

takes inordinately long to complete the rendezvous.

A task can also use the **delay** statement as an alternative to **accept** statements to only wait a defined amount of time for a rendezvous.

2.1.4 Communication Considerations

For Ada, the communication system that implements the constructs described above must be concerned about the following issues. These issues will be discussed in more detail when an actual implementation is explored.

1. The communication system must determine the location of the entry being called. The entry can exist in a number of tasks that reside on different processors.
2. When the **delay** statement is used in conjunction with an entry call, the communication system must be able to recognize this condition, and be able to cancel a rendezvous if the delay expires beforehand.
3. The communication system must be able to queue the entry calls in the proper sequence.
4. The communication system must be able to provide the **count** attribute associated with an entry.

2.2 Concurrent C

Concurrent C was designed as an extension to the C programming language for use in concurrent programming. [Gehani 86] It is based on synchronous message passing, and supports remote procedure calls, in much the same manner as Ada. It also provides the *rendezvous* mechanism for synchronization and communication.

The terms *process* and *transaction call* in Concurrent C have essentially the same meanings as a *task* and *entry call*, respectively, in Ada. They will be used in the

discussion that follows, to more closely conform to the terminology for Concurrent C.

Concurrent C and Ada are similar, in that they both use the rendezvous method for communication. The main differences exist in the **accept** statement and in the way a timed transaction or entry call is established. In Concurrent C, the **accept** statement allows for two additional clauses, the **suchthat** and **by**. The timed transaction call will be discussed in a following subsection.

The **select** and **delay** statements, as used by the called transaction, function the same in Concurrent C as in Ada, so they will not be discussed here. The example in Figure 2-6 will be used to indicate the syntax of the statements.

```
select {
  (i < n) :
    accept any(x) {
      /* Statements to execute */
    }
  or
  (i == n) :
    accept any2(y) {
      /* Statements */
    }
    /* Additional statements */
}
```

Figure 2-6. Example of Concurrent C Syntax for Transactions

2.2.1 The Accept Statement

The **suchthat** clause is used for selecting which of the calling requests will be allowed. The **by** clause is used for ordering of the accepted calls. Each clause is optional, and can be specified in conjunction with each other. The syntax of an **accept** statement with the optional clauses is shown in Figure 2-7. The *expression* is evaluated, and if it is true (non-zero), the request is accepted. All allowable requests are accepted in FIFO order, if

```

accept entry_name(parameters)
  [suchthat (expression)]
  [by (arith expr)] {
    /* statements to be executed */
  }

```

Figure 2-7. Syntax of Concurrent C Accept Statement

no **by** clause had been provided.

When a **by** clause is specified, the *arith expr* is evaluated for all outstanding, acceptable calls. The one with the lowest value is accepted first.

2.2.2 Timed Transaction Calls

For a caller to limit the time period for a rendezvous to start, it must call the transaction with the **within** statement. This statement is similar in intent to the **delay** in Ada. The transaction call will only occur if the rendezvous occurs within the period specified. It will not cancel the rendezvous if it already started.

The format of the statement is as follows:

```

within period ? proc.trans(actual_parameters) : expr

```

This statement will allow the rendezvous involving *proc.trans* to occur, if it happens within *period* seconds. Otherwise, the transaction call is canceled and *expr* will be evaluated.

2.2.3 *Communication Considerations*

The considerations are the same as mentioned for Ada, with the addition of the following issues.

1. The ordering of the requests for a transaction may be different if the *suchthat* or *by* clauses are used in a statement.

2.3 CHILL

CHILL [CCITT 80] is a language that was developed for the telecommunications industry, for Stored Program Control (SPC) switching machines. It is intended for real-time, fault-tolerant applications. It is a high-level language that can also be used in other applications.

Processes in CHILL have the same meaning as *processes* in Concurrent C and *tasks* in Ada.

Message passing is the only means of communication in CHILL. Remote procedure calling is not explicitly available.

The constructs that can be used for inter-process communication are the *buffer* mode and *signals*. The *send* and *receive* operations are used to pass messages through a *buffer* or *signal*. The following subsections describe the constructs.

2.3.1 *Buffers*

Processes communicate by passing messages through buffers. The statements *send* and *receive* are used to pass messages. Only messages of the same mode as that given in a buffer definition can be passed through the buffer. *Mode* in CHILL can be considered to be the same as *type* in other languages.

Optionally, a buffer can be defined to have a specified number of slots to hold messages

that have been sent, but not yet received. If no value is specified for the number of slots, the buffer is unbounded.

An example of a buffer definition is given below.

```
decl buff buffer (10) int;
```

In this example, a buffer named *buff* is declared. It contains ten slots, and can only accept messages of mode int.

2.3.1.1 Sending a Message A sender deposits a message of the type allowed into a buffer. If the buffer is not full, the sender is allowed to continue processing. If it is full, the sender will be blocked until there is room.

A sender can specify a priority for the message being sent. Each message has a priority attached to it. If none is specified when a message is sent, it is given a priority of zero. The priority can be used to determine the order of delivery of messages to receivers. The highest priority message will be sent to the next available receiver. If the highest priority message is from a process that is blocked because the buffer is full, that message will be sent, before a message in the buffer.

Message passing, using buffers is asymmetric, because the sender does not identify the receiver of a buffer message. Because the sender continues execution immediately, before the message is received, the sending operation is considered to be asynchronous.

An example of a call to send a message is shown in Figure 2-8. In this example, *num* is defined as an integer variable. It is assigned a value of five, and sent through the buffer *buff* to another process.

```
dcl num int;  
dcl buff buffer (10) int;  
  
num := 5;  
send buff(num);
```

Figure 2-8. Example of a CHILL Send Buffer Statement

2.3.1.2 Receiving Messages There are two methods to receive messages from buffers. One is called a *receive expression* and the other is called a *receive case action*. A receive expression is invoked by a process calling *receive*, and specifying the buffer name from which a message is to be received. The process is blocked until a message arrives in the buffer. The value in the message is assigned to a variable of the same mode.

An example of a *receive expression* operation is shown in Figure 2-9.

```
dcl msg char;  
dcl buff buffer char;  
  
msg := receive buff;
```

Figure 2-9. Example of CHILL Receive Expression Statement

The *receive case action* statement allows for alternative buffers to be waited on, or to possibly not wait on any buffer.

An illustration of the format of the receive case action statement is given in Figure 2-10.

```

receive case set send_proc

    (buff1 in msg1):
        /* statements */

    (buff2 in msg2):
        /* statements */

else
    /* statements */

esac;

```

Figure 2-10. Example of CHILL Receive Case Action Statement

If a message is available in either *buff1* or *buff2*, then it is put in the corresponding message location, and the statements associated with the alternative are executed. The process instance value of the sender of the message received is assigned to *send_proc*.

The order of selection when multiple buffers have messages is dependent upon the priority of each message. The message with the highest priority is received first. If multiple messages have the highest priority, selection is non-deterministic.

If no buffer has a message, and an else clause is specified, the statements associated with the else clause are executed. This alternative allows the process to not be blocked when there are no messages.

The receive operation can be either *synchronous* or *asynchronous*, dependent upon whether the else clause is specified.

2.3.2 Signals

The use of signals can be for both synchronization and communication. The definition of a signal can be in either of the following forms:

signal A to receiver;

or

signal B = (int, int)

In the first case, the **signal** is only used for synchronization, because no associated message is defined. In the second case, the **signal** sends a message that consists of two integers. The first statement also shows that the identity of the destination process class optionally can be specified as part of the definition for the **signal**.

Only signals that contain messages will be discussed in this section.

2.3.2.1 Sending Signals A process sends signals by using the **send** operation, in the same way as with buffers. The difference is that a destination is specified with signals. The destination is either identified in the signal definition, or in the **send** operation. When it is identified in the signal definition, it is the class of processes that may receive the signal. An actual **send** operation names the specific process instance that can receive the signal.

Sample uses of the **send** operation follow, which rely on the example previously given for signal definitions:

send B(1, 2) to procl;

send A;

In the first example, the signal B is sent to the process instance value given by *procl*, with the parameters 1 and 2. The signal A, with no parameters, is sent to any process of type *receiver* in the second example.

A priority can also be specified when sending a signal, and the semantics for the ordering of receiving signals is the same as that for receiving messages through buffers.

As with buffers, sending signals is asynchronous.

2.3.2.2 Receiving Signals Receiving signals is performed in exactly the same manner as receive operations involving buffers.

2.3.3 Communications Requirements

The following are the issues that need to be addressed by the communication system when addressing the communications primitives in CHILL.

1. Space needs to be allocated to store the messages associated with buffers. The storage needs to be managed either in a central location, or cooperatively, because it is not known where the process is located that will request the next message. If the buffer is bounded, the storage requirements are known at compile time. Otherwise, a large storage area, such as disk storage, may be needed to handle the possible number of messages.
2. Storage also needs to be allocated for messages associated with signals. The same storage requirements are possibly needed for signals as for unbounded buffers, because no limit is placed on the number of outstanding signals.
3. The communication system needs to be able to determine when a message is available, to handle the else clause of a receive case statement.
4. The location of the receiver needs to be identified, when *signals* are used. If a specific process instance is used, the communication system can locate the process directly. When a process class is used, the communication system has to determine where potential receivers reside.

5. The communication system must be able to handle messages of different priority, and deliver the highest priority message first.
6. The communication system must be able to provide the process instance value for the *receive case action* statement, when the *set* clause is used.

2.4 Summary

The communication primitives in Ada, Concurrent C and CHILL have been discussed in this chapter.

In Ada, communications is by means of the rendezvous. Tasks wishing to communicate through a rendezvous specify an entry point as the connection between the tasks. Either a calling or caller task is blocked if the partner task is not ready to participate in a rendezvous. Tasks on either side of a rendezvous (caller or called task) can determine alternatives through a select statement. They can also abort a rendezvous attempt, either by specifying a timeout value, or an else clause in the select statement.

Concurrent C provides similar capabilities to those provided by Ada. There are two differences. One is the syntax of the timed entry call. The other difference applies to the selection of alternatives in the select statement. The *suchthat* and *by* clauses are provided to allow different ordering of waiting callers. They allow higher priority calls to be processed before lower ones.

CHILL provides communication primitives that are different from those of Ada and Concurrent C. The *send* and *receive* statements are provided, and the semantics of the statements are very close to what would normally be expected from the names. Two different mechanisms are provided in CHILL for passing messages. One is a *buffer*, and the other is a *signal*. The semantics of a signal are similar to the transaction call in Concurrent C and the entry call in Ada, because parameters can be specified in the

signal, and a destination is specified, but the sender is not blocked waiting for the signal to be received, and no result is returned to the caller. A buffer is a repository for messages, and only messages of a specific type can be placed in a buffer. CHILL does not provide for the type of synchronization in its communication primitives that is implicit in the Ada and Concurrent C primitives. The sending process in a communication is allowed to continue, even if the receiver has not yet received the message.

The communication primitives have been examined to provide a basis for a discussion of the issues that need to be addressed when attempting to implement the primitives in a distributed environment.

CHAPTER 3 - DESCRIPTION OF A SUPPORTING COMMUNICATION SYSTEM

This chapter discusses a communication system that can be used to support communication operators found in a high level language. This communication system is used because the author is familiar with its operations, having worked on its development for the past three years, and it provides many of the same features that have been proposed for other languages, such as Communicating Sequential Processes (CSP) proposed in [Hoare 78], and Distributed Processes (DP) described in [Brinch Hansen 78]. It was designed to be used in a real-time processing environment, similar to the environments for which the languages were designed.

The components of the system are presented in this chapter, as well as the characteristics and functions of different aspects of the system. Each system component will be examined in sufficient detail to allow for a discussion in Chapter Four concerning any enhancements that would be needed to support the communications primitives in the languages presented in Chapter Two.

3.1 Specification of System

As mentioned in Chapter One, the communication system that will be used as a basis for providing the communications primitives in languages already exists. It operates under the Digital Equipment Corporation (DEC) VAX/VMS operating system [Kenah 88], and utilizes DECnet [DEC 86a], [Wecker 80] as the medium for communications between nodes in sections of the network.

Because some of the nodes are connected as a VAXclusterTM, the high-speed Computer

Interconnect (CI) bus is used for communication between them. This bus is used for a number of reasons:

- the bandwidth is almost an order of magnitude higher than the Ethernet bus used by DECnet (70 Megabits per second (Mbps) compared to 10 Mbps);
- the inherent reliability of the bus, and the software managing it, is better than that of DECnet.

This chapter will discuss the operational aspects of the system, in order to provide a basis for further discussion in Chapter Four concerning the additions necessary to support the communication primitives which have been outlined in Chapter Two.

The system was developed with a goal of providing communication support for a real-time, highly reliable, distributed system. The following are the major issues that drove its development.

- The messages need to be delivered with a high degree of reliability, but guaranteed delivery is not provided. In other words, the system is to provide a datagram service.
- The messages need to be delivered as quickly as possible between processes on the VAXcluster.
- The sender is not to be blocked, waiting for delivery of the message. As a result of this requirement, the sender does not know if a message is received, unless it explicitly waits for an acknowledgement. This acknowledgement results from another message transfer between the two processes.

DEC, VAX/VMS, DECnet and VAXcluster are trademarks of Digital Equipment Corporation.

- The receipt of messages can be either synchronous or asynchronous; the receiver can choose the method.
- If a service queue exists at the time of delivery, and a receiver is not ready to receive the message, the system will store the message.
- If a service queue does not exist for a message, it is discarded.
- The sender does not necessarily need to know the processor on which the receiver is running. The sender needs to specify the receiving processor, but that processor may be a logical name that the communication system will use to determine the physical processor(s). The mapping from logical to physical name must be dynamic, because of the characteristics of the environment. This topic is discussed in more detail in Section 3.5.1.

Section 3.2 discusses the characteristics of *service queues*, which act as repositories for messages in transit. Section 3.3 describes the format of messages handled by the communication system. Section 3.4 discusses the major software components. Sections 3.5 and 3.6 present the operations which are provided to application processes. Section 3.7 compares the Communication System with a network standard.

3.1.1 Network Topology

The distributed environment consists of a number of DEC VAX processors. Microcomputers, such as the Micro-VAX II, as well as superminicomputers, such as the VAX 8650, make up the processors that communicate within the system. A major advantage of using DEC processors and DECnet results from being able to integrate processors with different capacities in the same network.

The system is not limited to the processors mentioned. Any processor that can communicate through DECnet, and is compatible with the VMS operating system can

utilize the communication system.

Figure 3-1 depicts the topology of the network. As can be seen from the figure, different communication mediums, such as Ethernet and RS-232, can be used to connect the processors. This enables the environment to be more geographically dispersed than a Local Area Network (LAN) alone would allow.

Because of the topology shown, processes can communicate through different mediums. One of the design goals is to provide as much transparency as possible to the communicating processes. This is accomplished by requiring the communication system to determine the communication path. Also, redundant paths are not utilized in the system, other than those implicitly provided by DECnet. As an example, only the CI bus is used for communication between processors on the VAXcluster, even though the communication system could also provide connections through DECnet. The reasons for not providing the dual paths were two-fold. Firstly, if the CI bus is not available, the processors would crash. They require the CI bus to be functioning for the VMS operating system processes to communicate between processors. Secondly, the DECnet path would not provide the performance needed for inter-processor communications between the VAXcluster processors.

3.2 Service Queues

Service queues are system provided buffers and data structures used to hold and control messages. They are similar to *mailboxes* [Kenah 88] and *ports* [Balzer 71].

3.2.1 Characteristics

A service queue is uniquely identified by the following characteristics:

- Processor name,

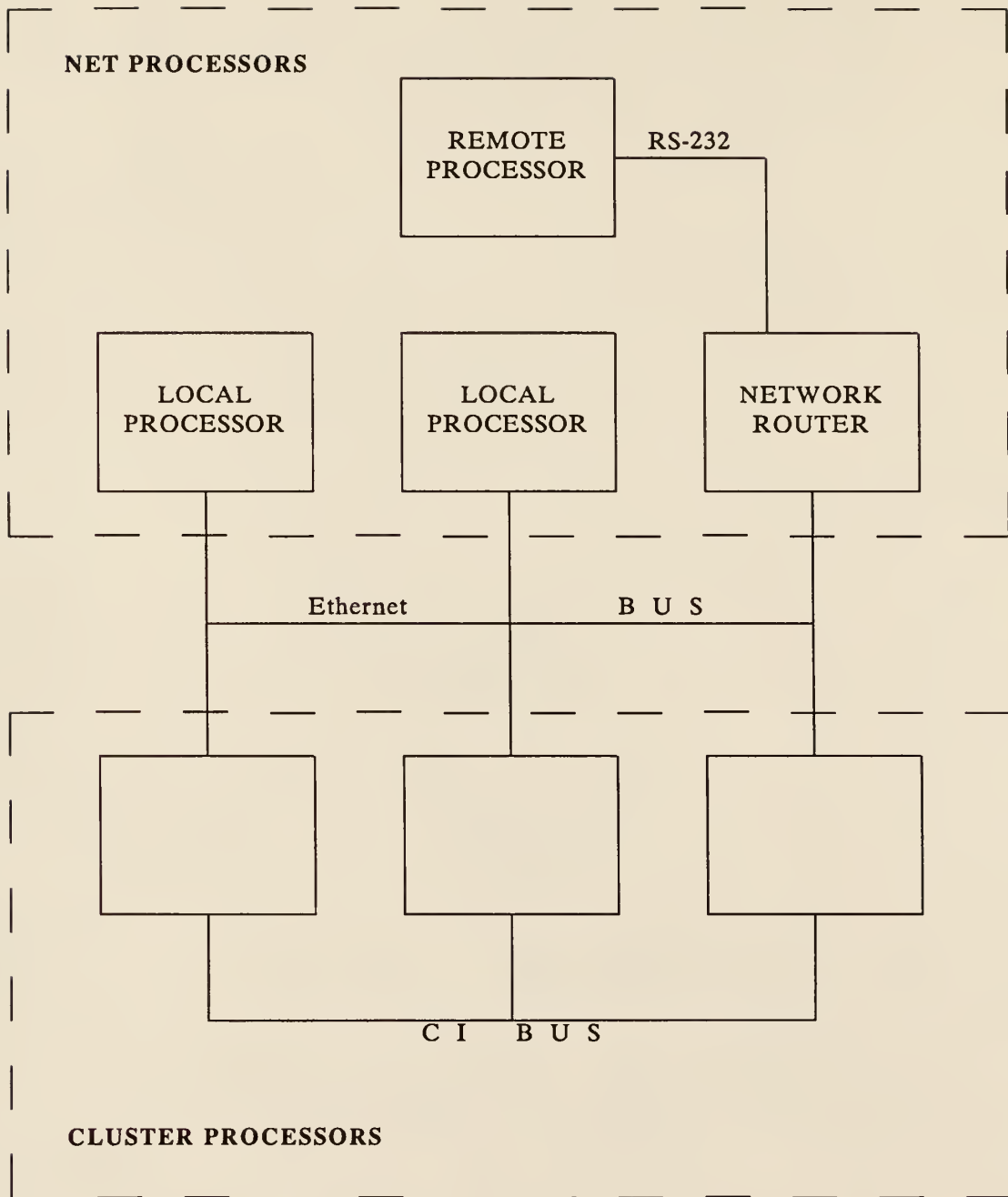


Figure 3-1. Communication Network Topology

- Service queue name,
- VMS group ID.

A VMS group is a collection of login names that are logically associated with each other. A group of login names normally share common functions in a organization. For example, all login names in the payroll department of a company would be defined to be in the same group. VMS uses the group ID of processes to control access to available resources. For instance, files can have protections for reading, writing or deleting based upon the group of the owner.

Each service queue can be uniquely identified by these values, and service queues with the same name, on the same processor, but in different VMS groups, would actually be different. In other words, a process that is operating in one VMS group, cannot send messages to a process in a different group, even if they reside on the same processor and use the same service queue name for communication.

A process sends messages to a service queue by naming it in a send operation. Refer to Section 3.5 for details on specifying how to send messages. A process connects to a service queue to receive messages by specifying the queue as one of the parameter values in a receive operation. Section 3.6 identifies the ways that messages can be received. Processes communicate by specifying the same service queues, and passing messages through it.

Service queues can also be identified as *permanent* or *temporary*. A permanent queue will exist, even if no process is connected to it. Whereas, a temporary queue will disappear when no process is connected to it. When a temporary queue disappears, all messages associated with it are also lost.

The advantage of permanent queues is that messages can be sent to them, even when no

process is currently available to receive them. This implies that the receipt of the message is important, and that timeliness of the receipt is not as critical.

Temporary queues have the advantage that messages which are intended for processes that do not exist will be discarded. In cases where the timeliness of the receipt of the message is important, and the information in the message is worthless if not received immediately, temporary queues should be used. They are also useful in circumstances where the receiver is not interested in messages that exist prior to the time it first starts up. If a temporary queue is used, no messages will be in the queue when the receiver first connects to it, thus relieving it of the responsibility of ensuring that no old, useless, messages are processed.

3.3 Message Content

Messages delivered through the communication system consists of two parts. The first part is the message header. It is created and maintained by the communication system, but is available to the application programs. The second part is the user data. This part is variable in length and is not evaluated by the communication system. The user can pass any data, in any format desired, in this area. Its interpretation is dependent upon the needs of the application processes.

The message header provides information that is useful to the application processes. The information of importance to an application is listed below.

message length Length of the message, including the header. Only this part of the buffer will contain valid information. A process must specify the maximum size for any expected message when establishing a receive operation. This field can be accessed by the application process to determine the length of a message. Because the length

of the header is fixed, the process can determine the length of the data portion of a message.

sending process The name of the sending process. This field can be useful for determining whether to accept a message.

sending process identification The process identification (PID) of the sending process.

sending node The name of the sending processor. It is useful in determining to which processor a response should be sent.

Additional information is also available in the header for the Network Server to use in the determination of the destination processor and service queue.

3.4 Major Software Components

The communication system consists of a number of components. These components will be described briefly in the following subsections. For a detailed description of the Device Driver and Network Server components, refer to Appendix C. Each component is necessary in order to provide the complete communication service for the configuration described above.

If the communication system were to be used only in a VAXcluster environment, the Network Server would not be needed. The Device Driver and Application Interface are always needed to provide the minimum service, even if inter-processor communication were not needed.

Figure 3-2 depicts the interconnection of the components and the application software.

3.4.1 Device Driver

The Device Driver is installed in the system to provide facilities to retrieve messages

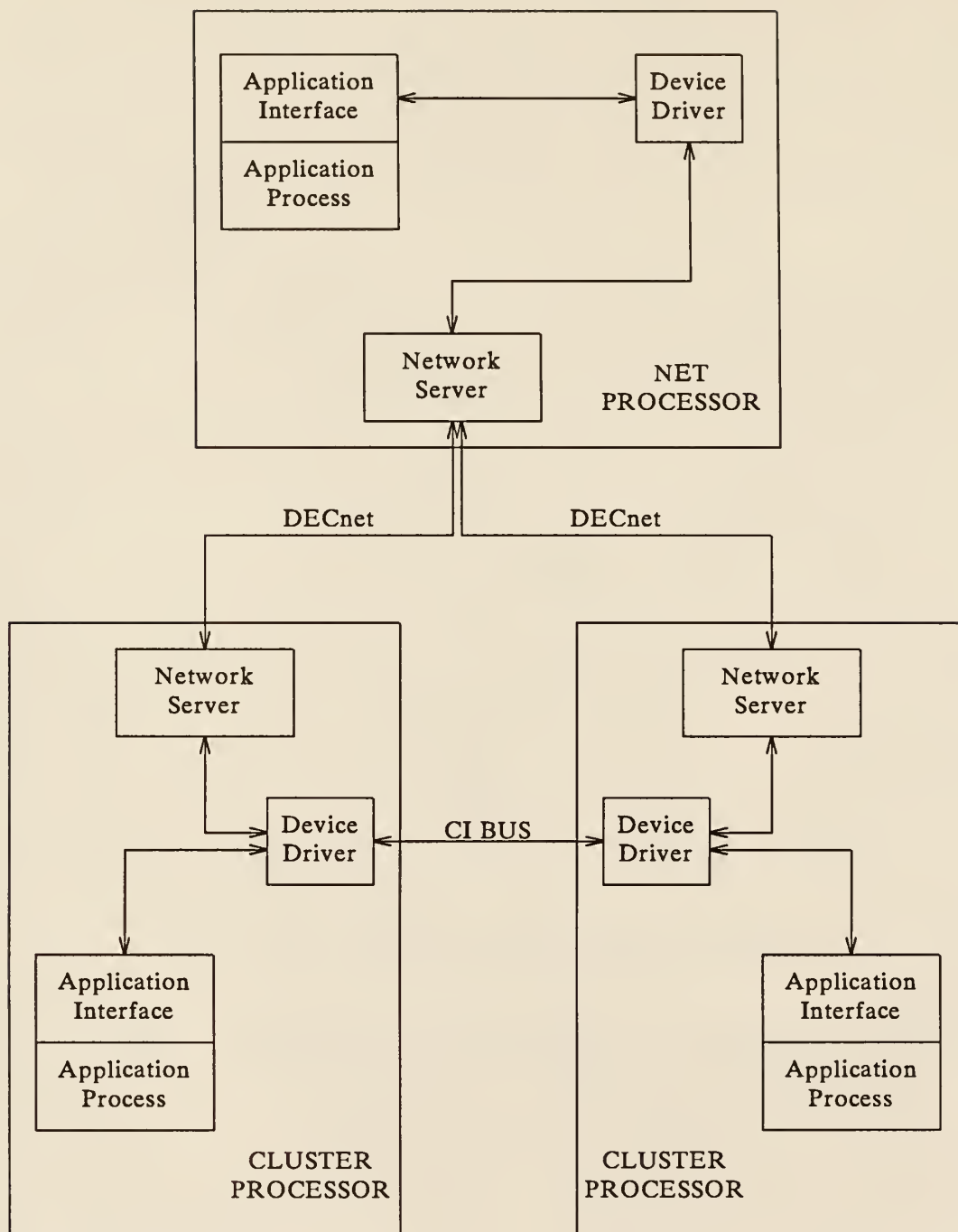


Figure 3-2. Communication System Components Interconnection

from and deposit messages into a service queue. Service queues were discussed in detail in Section 3.3.

3.4.2 Network Server

The Network Server provides functions similar to those of the Device Driver. It maintains DECnet connections to other processors. It also provides the service to send messages between processors.

3.4.3 Application Interface

This interface is linked in with the application software. It provides the interface between the communications processes and the application processes.

Application processes send and receive messages by calling these routines directly. The routines perform some initial processing before calling the Device Driver to complete the operation.

3.5 Sending Messages

There is only one available operation to send messages. It is called **SEND**¹. The parameters for the **SEND** operation, and its format, are described in Figure 3-3.

Messages are sent asynchronously. The sender is only notified if the request could not be queued on the local side. The reasons for a failure status usually signify problems that cannot be corrected by the sending process. These problems range from insufficient memory available to the device driver not being loaded. It is assumed that processes that encounter these problems will terminate themselves. Normally, these error statuses are encountered only during the development phase, and appropriate error statuses have

1. The names used in this thesis for the communication operations are not the actual names of the operations available in the Communication System. They are used to easily identify the operations performed.

SEND(*msg*, *length*, *queue*, *node*);

where:

msg address of the message to be sent.

length length of the message.

queue name of the service queue

node Destination processor name. This parameter is optional. If it is not specified, it is assumed that the message is local.

Figure 3-3. Format of SEND Operation

been provided to aid in debugging new code.

The only exception to the above occurs if the remote service queue is full. This status is returned to allow processes to exercise flow control, where it is appropriate.

The sending process can specify the node either as an explicit physical processor name, or as a logical name. The following subsection discusses logical names, and how they are mapped to physical names.

3.5.1 Logical Name to Physical Name Mapping

The communication system allows for logical names to be specified in a **SEND** operation. These logical names are mapped to actual physical processor names, based upon certain configuration information. The following discussion describes how the mapping is effected.

The processors in a network are divided into two groups. One group is in the VAXcluster, and will be called the *cluster group*. The second group is connected to the cluster group and interconnected only through DECnet; this second group will be

referred to as the *net group*. A process on a *net processor* can send messages to a process on any other processor only through the Network Server. As can be seen, the groups are mutually exclusive; a processor can only belong to one group.

A characteristic of the processes in the network is that all the processors in each group will normally contain a collection of the same type of processes. That is, a process of one type on a processor has a partner process on each of the other processors in the group.

Another characteristic is that some processes exist on only one processor in the group. All unique, individual processes exist on the same processor. The processor which holds these processes which are unique within a group is called the *coordinator processor*. These unique processes are used in coordinating the activities for the whole network. If the coordinator processor fails, all the processes are automatically migrated to another processor in the same group. This processor will then become the coordinator processor. All processors in the group are available to be selected as the coordinator processor, and one is chosen based upon an arbitration scheme, where each processor attempts to become the coordinator, and only one succeeds.

The previous definitions will be useful in the following discussion of the name mappings. The types of logical names supported are listed in Table 3-1.

3.6 Receiving Messages

Messages can be received either synchronously or asynchronously. Even with the synchronous form, the receiving process has the option to return immediately if no messages are available, or specify a timeout.

Table 3-1. Processor Logical Names

Processor Type	Description
<i>Coordinator processor</i>	<p>The process receiving the message is on the coordinator processor. The communication system is required to map to the correct physical processor name. For <i>cluster processors</i>, the name is translated by the Application Interface. For <i>net processors</i>, the name is passed to the Network Server, and may not be correct. Network Servers can reroute the message if it was delivered to an incorrect processor.</p>
<i>Other Cluster Processors</i>	<p>Send the message to the service queue on <i>cluster processors</i> other than the local one. This name will be translated to the physical names by the Application Interface, if the local processor is a <i>cluster processor</i>. If the sending processor is a <i>net processor</i>, the message will go to all <i>cluster processors</i>, and will be translated by the Network Server. The name is mapped to all processors in the configuration table that match the <i>cluster processor</i> class.</p> <p>Processes use this form to send a message to their partner processes on the other <i>cluster processors</i>. It is useful to coordinate, or disseminate, information.</p>
<i>All Cluster Processors</i>	<p>This logical name is used to broadcast the message to the specified service queue on all <i>cluster processors</i>. The mapping is done the same as for the above, except that the local processor will also receive the message, if it is a <i>cluster processor</i>.</p> <p>This mapping is useful to send the same message to the same process type on all processors.</p>
<i>Other Net Processors</i>	<p>This name is similar to the <i>other cluster processors</i>, except that the <i>net processors</i> are selected. If the local processor is a <i>net processor</i>, it does not receive the message.</p>
<i>All Net Processors</i>	<p>This is similar to the <i>all cluster processors</i>, except that the <i>net processors</i> are selected. If the local processor is a <i>net processor</i>, it also receives the message.</p>

The reason for the range of receive options is due to the nature of the applications which can use the communication system. Processes may be required to perform other functions while awaiting messages. Also, messages may be sent at random times, in response to other events, and the timing of these events is non-deterministic.

All the receive operations dynamically create buffers to hold the received messages. The default buffer size is 5K bytes, but the process can change the value on each call. The maximum size of a message which is allowed in the system is 32K bytes. If the buffer created is too small to receive a message, an error status is returned and the message is lost.

The receiving process must explicitly delete a buffer when it is finished using it. This is done with the `DELETE_BUF` operation. Its format is shown in Figure 3-4.

```
DELETE_BUF( buf );
```

where:

buf pointer to the buffer created by a receive operation.

Figure 3-4. Format of DELETE_BUF Operation

Processes can only receive messages on the local processor. When messages are sent, they are only delivered to the processor(s) specified. If a process is attempting to receive messages that are being sent to a different processor, it will not succeed.

The following subsections describe the different types of receive operations available.

3.6.1 Asynchronous Receive

A process may elect to receive messages asynchronously. This method allows the process to continue with its normal execution while awaiting a message. This is possible in VMS, because it supports interrupt handling within user mode processes, by delivering Asynchronous System Traps (ASTs) [Kenah 88]. When an outstanding asynchronous receive operation completes, the application routine specified when the operation was invoked will interrupt the normal processing to execute.

The application routine can perform nearly any operation available in the system on the received message. The only limitations occur when attempting to access data areas that may also be changed by the mainline code. The AST can interrupt the mainline code at any point, and must be concerned about changing variables that the mainline code also changes, or accesses. This synchronization is a concern of the application programmer, and must be dealt with appropriately to ensure correct operation of the process.

The format of the asynchronous receive is presented in Figure 3-5 in which the parameters to the call are described.

3.6.2 Synchronous Receive

In this operation, the process will be blocked until a message is received. It can optionally specify a timeout value to prevent it from being delayed indefinitely.

This form of the receive operation is normally used by processes that only respond to requests passed to them by messages. They have no other operations to perform. They could be considered server processes that respond to requests passed to them as messages.

Another use for this operation is to receive responses to previous messages. This is useful in ensuring that previously sent messages had been received at the destination. It

```
ASYN_RECV( que, astrtn, astprm, bufsize );
```

where:

- que* Name of the service queue. This queue is local to the calling process.
- astrtn* Address of the routine that will execute when the receive operation completes. The routine is passed the status of the call, a pointer to the buffer containing the message, and the parameter passed by the caller. The status should be checked to determine if the operation completed successfully before accessing the message.
- astprm* The parameter that will be passed to the AST routine. This parameter can be any 32 bit integer value. It is useful for identifying which invocation of the receive operation caused the AST routine to be called, when multiple outstanding receive operations call a common routine.
- bufsize* This parameter is optional, and specifies the size of the buffer that should be created to receive the message.

Figure 3-5. Format of Asynchronous Receive Operation

also allows for responses that indicate errors in the execution of an operation. For this purpose, the timeout parameter is available.

The format of the synchronous receive operation is shown in Figure 3-6.

3.6.3 Receive Immediately

This operation is used when the process does not want to receive messages asynchronously, and also does not want to wait for non-existent messages. It can be considered to be a synchronous receive operation, with a timeout value of zero.

If no messages are available, an appropriate success status will be returned to the caller.

SYNCH_RECV(*que*, *buf*, *timeout*, *bufsize*);

where:

- que* Name of the service queue.
- buf* Pointer to the buffer used to hold the message.
- timeout* The time to wait for the message to be delivered. The resolution of the time interval is 10 milliseconds. This is an optional value. If it is not specified, the process will wait forever.
- bufsize* This parameter specifies how large to make the receive buffer. It is optional.

Figure 3-6. Format of the Synchronous Receive Operation

RECV_NOW(*que*, *buf*, *bufsize*);

where:

- que* Name of the service queue.
- buf* Address of the buffer used to hold the message.
- bufsize* Size of the buffer. If not specified, the default value is used.

Figure 3-7. Format of the Receive Immediately Operation

The format of the operation is shown in Figure 3-7.

3.7 Comparison of Communication System

It would be informative at this time to compare the functions provided by the Communication System with those defined in the seven layer Open System Interconnection (OSI) standard developed by the International Standards Organization

(ISO). This standard is used because it is generally known, and provides a level of abstraction that makes it easier to discuss general functionality.

The International Telegraph and Telephone Consultative Committee (CCITT) adopted the OSI Reference Model for its development of communications standards. Reference X.213 was produced to describe the functions that are to be provided by the Network layer in the OSI Reference Model.

The Communication System will be compared to the service expected to be provided by Network layer. [CCITT 85] (also identified as X.213) defines the Network layer as providing transparent data transfer between users. The Network layer is examined because the services provided by the Communication System more closely correspond to those defined for this layer than for any of the other six layers.

The functions specified for the Network layer are described in the following subsections. The descriptions of the functions are paraphrased when the terminology of the recommendation makes the meaning unclear, without introducing the meanings of the terms used within the recommendation. When the recommendation is directly quoted, quotation marks surround the quoted phrases. The extent to which the Communication System provides each indicated function will also be described. The term *Network Service provider* is used in the following discussion to indicate any entity that provides the services defined for the Network layer.

3.7.1 Independence of Transmission Media

A Network Service provider is to provide "independence of underlying transmission media" to its users.

The Communication System provides this function by determining the route each message is to follow. Depending upon the source and destination processor, a message may travel

either over DECnet or the CI bus.

3.7.2 End-to-End Data Transfer

A Network Service provider is to provide "end-to-end" data transfer.

The Communication System is responsible for taking a message from a sender's address space, and delivering it to a receiver's address space, so it minimally satisfies this requirement.

X.213 specifies that a Network Service provider must provide connection services to its users, as part of the end-to-end data transfer. Connection establishment and release are two phases involved in providing a connection between a pair of users. The Communication System does not provide for connections. The overhead required to establish, maintain and release connections were considered too excessive for the types of messages sent between processes. Most processes only send messages to other processes infrequently.

Processes that need connections can establish them through the operations available in the Communication System. They would be responsible for establishing and maintaining their connections. Connections would generally consist of the two processes communicating through two service queues. One process would send messages to the receive service queue of the other process. This is a burden on the developers for the processes using the Communication System. For the applications planned for the Communication System, the number of processes requiring connections is small compared to the ones sending messages unidirectionally, and it was decided that the Communication System should not incur the overhead for providing the service.

Because connections are not supported within the Communication System, the phases described by X.213 for establishment and release of a connection will not be discussed.

It is more useful to discuss the data transfer phase, because the Communication System does provide this function. Within the data transfer phase, the Network Service provider is to provide for exchange of messages, either unidirectionally or bidirectionally, simultaneously. The Communication System provides for passing messages in a single direction only. But processes that maintain their own connections can initiate transfers in both directions, and the Communication System can handle this condition.

According to X.213, a Network Service provider must also be able to preserve both the sequence and the boundary of messages. The Communication System preserves the boundary of messages, because it transfers whole messages. It does not perform any packet assembly or disassembly of messages. DECnet does packetize messages that are over its maximum message size, but it guarantees to preserve sequence and message boundaries.

3.7.3 Transparent Data Transfer

A Network Service provider provides "transparency of transferred information". This requirement means that the Network Service provider does not interpret the information given it by its user.

The Communication System meets this requirement. The messages sent through the Communication System are all treated the same, regardless of content. The only processing of a message done by the Communication System is to add a header. This header is provided by the Communication System for use by the receiver of a message.

3.7.4 Quality of Service Selection

A Network Service provider is to allow selection of "Quality of Service parameters" for a network connection.

Because the Communication System is based on datagram message delivery, instead of virtual circuits, it does not provide for connection establishment and release. As a result, it does not provide for quality of service specification.

The quality of service provided is dependent upon the route selected for each message, and the type of processor on which the sender and receiver processes reside. The speed of the processors that constitute the computing system, (minicomputers vs. microcomputers) and characteristics of the communication media (DECnet vs. CI bus vs. local memory copy) strongly influence the quality of service.

The range of values for the quality of service parameters specified by the standard, that are provided by the Communication System, are given below. The parameters associated with connection establishment and release are not discussed, because connections are not used.

— "Throughput"

Throughput is defined as the number of messages passed between the sender and receiver per unit of time. According to X.213, the throughput should be specified for each direction of a connection. Because connections are not used, the throughput is equivalent for each direction.

The range of throughput is between 50 messages per second for message transmission over DECnet, and 1000 messages per second for messages sent locally on a minicomputer.

— "Transit delay"

Transit delay is defined as the time interval between the request by a process to send a message and the receipt of the message at the specified destination. Because the Communication System does not utilize connections, and a receiver

may not be available at the time a message is sent, the values presented are only the time necessary to send a message to the service queue.

The transit delay is highly dependent upon the congestion of the transmission media, which the Communication System has little control over. The delay can be from one millisecond for intraprocessor message transmissions to two seconds for DECnet transmissions.

— "Residual error rate"

The residual error rate consists of the ratio of the number of lost, incorrect and duplicate messages to the number of received messages.

The error rate is influenced more by the transmission media used by the Communication System, than by the Communication System itself. The Communication System is reliable, when considered by itself, because it provides minimal functionality. It takes advantage of the reliability provided by the transmission media. The transmission media consist of the VMS operating system and the DECnet software. The operating system and DECnet provide for error detection and correction in any message transmission. Flow control, packetizing, sequencing and message acknowledgement are also provided by them.

Although the residual error rate has not been measured quantitatively, it has been observed to be much less than one percent.

— "Transfer failure probability"

This value refers to the ratio of the number of messages sent that were not received to the total number of messages sent.

This value is partially the result of the underlying transmission media and the Communication System itself. In this case, the Communication System will

purposely discard messages. The reasons include: no service queue available to receive the message, and no path to the designated processor.

The major influence on this measure is the correctness of the implementation of the application. Messages should not be sent to non-existent service queues.

There is no quantitative measure of this characteristic. Because the Communication System is specified as a best-effort datagram service, any application using it must be aware of potential problems with lost messages, and handle the conditions appropriately.

3.7.5 User Addressing

The Network Service provider is to provide for addressing of its users.

The Communication Server provides for addressing by using a service queue. Each service queue is unique by processor name, service queue name and VMS group identifier. This addressing is not unique by process, because multiple processes can send to or receive from a service queue.

3.8 Summary

This chapter has presented a Communication System that currently exists. The specifications for this system, as well as the network topology it accommodates have been provided. The communication system had been developed to be used in conjunction with an application system and, as a result, only provides services that are considered necessary for the application.

The concept of a service queue was presented. A service queue is a uniquely identified repository of messages. Processes can send and receive a message by specifying the same service queue.

The components of the Communication System were described to provide a better understanding for how the specifications for the system were implemented. The requirements for performance, reliability and transparency of the destination led to a unique system being developed. Some of the necessary tradeoffs resulted in operations available to the user that require him to know more about the network and its topology than normally should be allowed. The user is also required to provide for guaranteed message delivery. The tradeoffs that resulted from the requirements are:

- no confirmation that a message had been delivered,
- no reliable means to determine if a processor is available to receive messages,
- message transfer times are dependent upon the path selected.

Parts of the Communication System are used for routing of messages. If a message cannot be delivered, an alternate route is tried. This allows for greater probability that a message is ultimately delivered. This routing only occurs within the Network Server. The Device Driver does not use alternate routing for messages to be sent between the cluster processors, because it is assumed that if a path does not exist from one cluster processor to another, it is because one processor is down. In that case, the message is not deliverable anyway.

The types of operations available to the user of the Communication System were described in this chapter. The normal operations for sending and receiving messages were presented. The destination processor for a send operation can either be a physical processor, or a logical name that the Communication System maps to a physical processor, or group of processors. This system is also unusual because of the number of receive operations allowed to the user. A user can choose between asynchronous and synchronous reception of messages. Also, within the class of synchronous receive

operations, a user can choose to select a timeout value, or to receive an indication that no message is available. Miscellaneous operations, such as canceling a receive operation on a service queue, and emptying a service queue of all its messages are also provided. These operations provide greater flexibility and control on the part of the users. This extra freedom also puts a greater burden on the user to ensure that the programs using these operations are correct.

The Communication System has been compared to the Network Layer of the OSI Reference Model in order to put the capabilities of the system into perspective.

CHAPTER 4 - IMPLEMENTATION OF COMMUNICATION PRIMITIVES

This chapter discusses the implementation issues involved with attempting to use the communication system discussed in Chapter Three to implement the communication primitives in Ada, Concurrent C and CHILL. As was mentioned previously, the purpose of this paper is to investigate the concerns that arise when communication primitives in standard languages are used in a distributed environment. A suggestion for an implementation is now presented to show the issues that must be tackled in resolving the problems.

Section 4.1 discusses the layers of software that are needed to support the communication primitives. The components of one of these layers, the Language Interface, are presented in Section 4.2. In Section 4.3, the issues concerning an implementation which supports the communication primitives in Ada are presented. An implementation of Concurrent C in a distributed system is discussed in Section 4.4. The implementation issues which must be addressed in order to support the communication primitives of CHILL are discussed in Section 4.5. A summary of this chapter is provided in Section 4.6.

4.1 Hierarchy of Layers

Supporting the communication needs of the communication primitives of the languages from Chapter Two (Ada, Concurrent C and CHILL), in the environment of the communication system presented in the last chapter, requires an additional layer of software to be implemented. This layer will be referred to as the Language Interface layer in the rest of this paper, and it consists of a number of components. The general aspects of the layer's components are covered in Section 4.2. The specific capabilities of

the layer, that relate to each language, will be presented in the appropriate sections of this chapter.

Table 4-1 presents the layers needed to support the communication primitives, and the responsibilities associated with each layer.

Table 4-1. The Three Service Layers

<i>Software Layer</i>	<i>Responsibilities</i>
Language Interface	This layer is called directly by the language constructs. It is responsible for all operations necessary to transform the constructs to the appropriate calls to the Communication System. It also ensures reliable communication by including any additional protocols. Any additional data structures needed to maintain state information are kept by this layer.
Communication System	This layer is responsible for the transmission and reception of messages. It provides the services described in Chapter Three of this paper.
Operating System	This layer is responsible for the normal system services provided by the VMS operating system. The communications and synchronization mechanisms of the operating system are utilized by the Communication System. DECnet and CI communications are considered to be part of the Operating System.

One reason for the layers is that the Operating System and Communication System layers

already exist, and the needed additional functionality should be added in another layer for the purposes of modularity and flexibility. Also, the implementation of the Language Interface layer would be different for each language, because each language uses different primitives for communications, and the semantics of similar primitives are different.

Each layer in the system only communicates with the layers that are directly above or below it. See Figure 4-1 for the organization of the layers. The implementation of a layer may be changed, as long as the functionality remains the same, thus there is little or no impact on the surrounding layers.

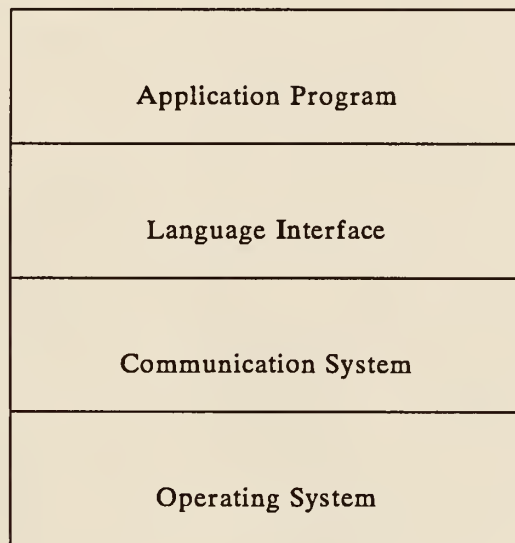


Figure 4-1. Layers of Communication

4.1.1 Assumptions

The following assumptions are used in the discussions of specific properties of the Language Interface which occur throughout the rest of this chapter.

1. Processes may exist on any processor in the network. The sender process does not explicitly know the location of the receiver process involved in any communication.
2. Each language allows for the existence of multiple senders and receivers, each possibly on a different processor. This implies that any necessary buffering is either maintained centrally, or is distributed. In either instance, the Language Interface needs to know where each message resides, so that it can be sent to a waiting receiver.
3. For considerations of fault-tolerance, the ensuing discussion must assume that processors can leave and come into the network. In other words, the network is dynamically reconfigurable. As a result of this consideration, messages that reside on processors that leave the network must not be lost. This consideration means that either the messages are stored to disk, or the buffers are replicated on multiple machines, or the messages reside on the processor of the sender until requested by a receiver. In either case, a performance penalty results, because additional processing is needed to copy the messages, and additional state information concerning the location of messages needs to be maintained.
4. For calls in a language that have arguments present, e.g. entry calls in Ada, it is assumed that checking the consistency of the number and type of the arguments between the called and calling processes, is performed by the compiler. When pointers are specified, the items referenced must be the same type for both processes, because the referenced item will be copied into and from the message.

5. Depending upon the scope rules of a language, some actions may result in intra-process communication. An example would occur when buffers or signals are defined local to a process in CHILL. In this case, assuming a process only executes wholly on a single processor, a communication through the buffer or signal is intra-process, and could be implemented directly by the compiler and run-time library, and will not be discussed in this chapter. It is assumed that all communication is between different processes, not within a single process.
6. In Ada and Concurrent C, when processes terminate, all outstanding messages sent by them are removed from the system. Also, all state information referencing the processes must be removed. In CHILL, the opposite action must occur, that is, the messages sent by terminated processes through buffers and signals must remain in the system.
7. All communicating processes are started in the same VMS login group. The Communication System only allows processes in the same group to communicate, thus allowing processes in separate groups to coexist in a network, without interaction.

4.2 Language Interface Components

The two main components of the Language Interface are the run-time routines that are linked into each process in the system, and a Directory Server. The run-time routines are used in each process to map the communication primitives provided by a language into services provided by the Communication System. The Directory Server maintains state information for all processes written in a language involved in interprocess communication. A facility with this capability is needed because the information needed for connecting processes for communication is not available in any other part of the network.

The general functions of these components are presented in the following subsections.

4.2.1 Run-time Routines

These routines are linked into each process by the linker for the language. The routines for process initiation and process termination are executed for all languages in essentially the same manner. They are responsible for maintaining state information, and connecting with or disconnecting from the Directory Server.

The activities performed by each group of routines are described in the following subsections.

4.2.1.1 Process Initialization Routines When a process starts execution, routines in the Language Interface execute to create any necessary service queues needed to allow communication with the Directory Server. A special queue must be created for each process to receive messages from the Directory Server. The queue must be distinct for each process on the processor. The name of the queue has the following format.

XXXX_DIRECTORY_RSP

The **XXXX** part of the name is distinct for each process. It could be the Process Identifier (PID) created by the operating system for the process, or the process name, depending upon the conventions used in the operating system to ensure uniqueness. The queue name also has to be able to be reproduced by the Directory Server, so it can determine the destination queue for sending messages to the process. In this paper, the PID will be used.

The Directory Server will be able to construct the response service queue name from the PID in the header of a received message. This header, as mentioned in Chapter Three, is provided by the Communication System. The VMS Operating System ensures that the

PID is unique within a processor.

4.2.1.2 Routines for Sending Messages Routines exist to determine how to send messages to other processes, when the appropriate call is made in the language. A message is sent when an entry call in Ada or Concurrent C is made, or a send operation in CHILL is performed. In the case of an entry call, the routines also wait for a response from another process.

Messages are also sent between the processes written in a specific language and the Directory Server. These messages are used to convey state information.

All messages sent by processes or the Directory Server rely upon the SEND operation provided by the Communication System.

4.2.1.3 Routines for Receiving Messages When a process wishes to receive messages from other processes, it calls these Language Interface routines. The necessary interaction of a process with the Directory Server and the Communication System will be done in this set of routines. The specifics of the actions performed are dependent upon the language, and will be presented in the sections discussing the languages. These routines would be invoked by an accept statement in Ada or Concurrent C, or a receive statement in CHILL.

These routines receive messages from the Directory Server, which convey information concerning the state of the system. For instance, the Directory Server sends a message to a process when it times out waiting for an event to occur, which was requested by the process. The routines receive messages from the Directory Server on the service queue specified in Section 4.2.1.1. Any of the receive operations allowed by the Communication System, synchronous or asynchronous, can be called by these routines.

4.2.1.4 Routines Invoked when a Process Terminates Routines exist that will be invoked when the process terminates. These routines must execute regardless of whether the termination is normal or abnormal. The routines are responsible for cleaning up any outstanding requests, and communicating with the Directory Server. The Directory Server itself maintains state information about the system.

4.2.2 Directory Server

The Directory Server is responsible for determining when processes are attempting to communicate. Because a process does not explicitly specify the location of its communicating partner, the Directory Server must include a mechanism to determine the location.

The Directory Server maintains information concerning processes that are attempting to communicate. When a process wants to receive a message, possibly as a result of an **accept** statement in Ada, it registers that intention with the Directory Server. The Directory Server returns a message to each process attempting interprocess communication which specifies whether or not another process is waiting to communicate with it.

These activities imply that the Directory Server must maintain state information about the network. The normal method used to store the information would be in a set of tables in memory, or in a disk file. The problems of maintaining this data in a fault-tolerant manner are similar to those encountered in maintaining a database in a distributed system.

Two alternatives for maintaining the data are to have a central process maintain all of the data for the network, or to have the data distributed across all, or a subset, of the processors. The advantages of the first alternative are the data is maintained in one

place, so multiple copies do not have to be synchronized, and also access to the data can be single-threaded, thus reducing the need for concurrency control. Disadvantages of this approach are: that the data could be lost if the process terminated abruptly; if the data is saved to non-volatile storage, it would still be unavailable until the process is restarted; performance penalties also result from single-threading access to the data. The server process could be multi-threaded to allow concurrent access to the data, but that would make the process more complicated, and consequently less reliable. The data could also be checkpointed to disk, so it could be recovered when the process terminated, and later restarted, but the data would still be unavailable during the interval that the Directory Server is down.

Replicating the data would resolve most of the problems mentioned in the previous paragraph concerning a single server. Multi-threading the access to the data would be easier, because multiple servers exist. If one server process terminates, the other processes are still available to maintain the data, and provide service to the other processes in the system. But this approach also has disadvantages. One is that the multiple servers must ensure that their copies are consistent. Changes in one copy must be reflected immediately in the other copies. When processes update the data, an appropriate mechanism must be used to ensure that two processes do not attempt to update the same data concurrently. Possible methods for coordinating multiple updates could include the two phase commit protocol described by [Kohler 81] for transaction processing, or employing locks on the individual entries for the state information.

Because the issues involved with a distributed database system have been explored extensively in the literature [Bernstein 81], [Date 83], [Filman 84], [Kohler 81], [Liskov 82], and are worthy of a study in their own right, implementation details of the Directory Server will not be presented. It is assumed that the state information provided

by the Directory Server to the Language Interface run-time routines is accurate, consistent and timely.

Since the Communication System used does not provide guaranteed message delivery, the Language Interface run-time routines that communicate with the Directory Server must ensure that the messages passed between themselves and the Directory Server are delivered. The state information passed in the messages is important to the operation of the communication primitives. The following subsection discusses the protocol used in communications by the run-time routines and the Directory Server to guarantee message delivery.

4.2.2.1 Message Passing Protocol Each message passed between the Language Interface run-time routines and the Directory Server must be acknowledged, to ensure that the message has been received by the destination process. To ensure that the originating process had received the reply message, a second acknowledgement message must be sent back to the second process. This method is similar to the three-way handshaking protocol for connection establishment described in [Tanenbaum 81].

Each process updates its state information only after receiving an acknowledgement. A timeout is the mechanism used in the receive operations to ensure that a process does not wait indefinitely for an acknowledgement. If a process times out before receiving an acknowledgement, it sends the message again. The timeout provided should be sufficient to allow the message to be delivered, and acknowledged, assuming a worst-case communication delay, but it should be short enough so that the process does not wait too long, because the languages being implemented are to be used for real-time applications. The amount of time to wait is difficult to determine, given the variability of the communication medium, and the desire for optimum performance required by the languages examined. It is assumed that a vast majority of the time, the message will be

delivered correctly by the Communication System, and will be acknowledged immediately. If a timeout occurs, the process resends the message. Duplicate messages can be recognized through the use of a sequence number. If a second timeout occurs, the process can assume the other process has terminated, and will execute an error recovery procedure (see the next section).

4.2.3 Error Handling

Error detection and recovery in a distributed system is very difficult. Part of the problem is that the layer approach does not allow the higher layers direct access to the lower layers to determine if problems exist. In a typical communication system, each layer assumes that the lower layers handle all errors, and provide error-free services. When that assumption is not valid, the error detection and recovery becomes more difficult. In the system presented, the lower layers, particularly the Operating System and Communication System layers, provide no error handling. So any error handling and correction must be done at the higher levels, either in the Language Interface or Application Program layer.

One approach that the Language Interface layer could take would be to raise an exception whenever a condition exists that it cannot handle. This approach can be used with Ada and CHILL, because they have exception handling facilities. It is not an ideal solution, because the language standards only specify certain exceptions which can occur when the communication primitives are used. An example of a specified exception is the PROGRAM ERROR that occurs when all guards in an Ada select statement are false.

Although the approach is not ideal, it can be used in this paper to simplify the presentation of the implementation. It could also be argued that the approach is acceptable, as long as the possible exceptions are identified, and added as extensions to the language. A discussion of these topics is provided in [Burns 87]. Although their

discussion is in the context of tasking in Ada, the issues raised are pertinent to any fault-tolerant, distributed computing system. The system provides very few facilities to help a user with the problems of error detection and recovery.

In the discussions on the implementation of the communication primitives, error cases will be ignored, to simplify the presentation. This is an acceptable approach, because the error cases can be numerous, and discussion of the processing necessary to handle them would detract from the discussion of the normal operations of each communication primitive.

4.2.4 State Diagrams and Transition Tables

Individual state diagrams and their corresponding transition tables are used to describe the state changes that occur in the Language Interface run-time routines for each communicating process. The diagrams and tables help to more concisely present the allowable states and actions associated with each communication primitive. They are used in conjunction with the discussions in the following sections to fully describe the actions that occur in the run-time routines.

Individual state diagrams indicate all possible states for a particular function, and the allowable transitions between the states. An oval is used to identify each state, while directed arcs indicate the allowable transitions. A state is identified by capital letters, and represents the current state, or location in the execution of a process. A transition indicates the event that must occur to move from one state to another in a process. In the transition tables associated with each individual state diagram, actions that are taken in a process as a result of the occurrence of an event are indicated.

Composite state diagrams are used in this paper in a form similar to that used by Sunshine [Sunshine 75] for the validation of network protocols. In a composite state

diagram, pairs of states for individual communicating processes are presented, with the allowable transitions between each pair of states.

The diagrams and tables can serve a number of purposes in the specification, design and testing phases of a project such as this proposed implementation. They can aid in ensuring the correctness of the specifications, by showing that the states and transitions necessary to satisfy the specifications are reasonable and complete. A measure of reasonableness is the number of states and transitions in a diagram. There should be enough states and transitions, so that the user can clearly understand the representation of the problem, but the number of states should not be so great that the user is confused. If the states and transitions are reasonable, then an implementation should be able to proceed easily. If they are not reasonable, the specifications may be overly ambitious or incomplete. If the states and transitions are complete, the state diagram should result in proper termination conditions. Incomplete states and transitions result in an inability to traverse the state diagram to a final state. For example, suppose a process should be able to transition from state *a* to state *b*, because of the occurrence of event *e*, but that the transition is not indicated by the specification as being valid, or is not identified. The state diagram would be able to show the need for the transition in the specifications.

Composite state diagrams are even more useful in this respect, because they are able to examine the states of two processes. The interactions between the processes become more obvious when a composite state diagram is used. The diagram helps to ensure that inconsistent pairs of states cannot be reached, and that sufficient states and transitions exist in the individual state diagrams to ensure correct representation of the interactions between communicating processes.

In the design phase, the individual state diagrams can be used to determine the modules that need to be developed. The transition tables could be expanded to determine the

design of each module. Each action specified in the transition table could be decomposed into a state diagram containing greater detail.

In the testing phase, the state diagrams could be used to determine the test cases required to test the implementation. Test cases for all, or representative, valid and invalid transition sequences could be developed. The state diagrams would be an aid in determining which test cases should produce valid results, because transitions are allowed that can reach a valid termination state.

4.3 Ada

The main form of communication between tasks in Ada is via the *rendezvous*. This concept was presented in Chapter Two. A task calls an entry point in another task. When the second task is ready to accept the rendezvous, the activity corresponding to the entry point is accomplished, and each task can continue. The rendezvous is a synchronization, as well as communication, primitive. Either task can be blocked waiting for the other one to participate in the rendezvous.

The Directory Server maintains information about the tasks that wish to communicate via a rendezvous. It is responsible for determining which tasks will participate in each rendezvous.

The following subsections present the steps involved in each task that attempts to participate in a rendezvous.

4.3.1 *Calling an Entry Point*

A task calls an entry point to indicate its interest in a rendezvous with a process that contains that entry point. The call can be made either within a select statement, or as a single statement. Since a call made from within a select statement includes the states involved when there is only a single statement, only the processing required for the select

statement will be presented.

The following steps outline the necessary actions in the Language Interface for the entry point call. Refer to Figure A-2 and Table A-2 of Appendix A for the individual state diagram and transition tables, respectively, for this operation.

1. The task, desiring an entry call, begins in the *START* state. The Language Interface run-time routines send a message to the Directory Server process to indicate its interest in a rendezvous with the indicated entry point. The task name and entry name are passed to the Directory Server process. The timeout value, if a *delay* statement is specified in an *or* clause of the *select* statement, is also passed in the message.

The run-time routine issues a call to *SYNCH_RECV* on its response queue from the Directory Server. The Directory Server sends a message back to the run-time routine, indicating whether a task is available for the rendezvous. Based upon the information, the run-time routine selects one of three possible actions. They are:

- Wait for a rendezvous, because no task is available to accept the entry call, and an *else* clause is not specified in the *select* statement. (*WR*)
- A task is available. (*SM*)
- No task is available, and an *else* clause was specified. (*EE*)

The actions involved in each alternative are described below.

2. If a rendezvous is not possible from the *START* state, and an *else* clause is specified, the task moves to the *EE* state in which the statements associated with the *else* clause are executed.

The task is then continued, in the *CE* state, with the statement following the *select* statement.

3. If no task is available, the task waits, in state *WR*, for the Directory Server to inform it when a task, with which it can rendezvous, is waiting. The run-time routines call `SYNCH_RECV` on its response queue from the Directory Server. If a **delay** was specified as an alternative, the task will receive a message from the Directory Server indicating either that a task is available for a rendezvous, or the task timed out. One of the issues with the timeout concerns when to start the timeout interval. It can either be when the entry call is made, or when the request is queued. As discussed in [Burns 87], either alternative causes problems in a distributed system, because of the special handling that results from a delay of zero, and the unclear definition in the language standard [USDoD 83].
4. If the task times out waiting for a rendezvous, the Directory Server sends a message to the run-time routine, to have it execute the statements associated with the **delay** alternative of the `select` statement. The task moves to the *ED* state. The Directory Server also removes the task from the queue of waiting tasks for the entry point.

After executing the **delay** statements, the task continues execution with the statement following the `select` statement, going to the *CE* state.
5. If a rendezvous is possible, either from the *START* state, or from the *WR* state, the Directory Server sends a message to the run-time routine, informing it of the destination for the rendezvous. The task moves to the *SM* state. The destination indicates the processor name to receive the message sent by the run-time routine. The service queue that is the target of the message is constructed from the task name and entry name. For example, assume that the entry point called is *entry1* in task *taska*, the service queue name would be:

taska_entry1

A message is constructed by the run-time routines to send to the destination service queue. The message contains the argument values from the call. If values are passed by reference, the values referenced are put in the message, instead of the pointers.

The run-time routine uses the **SEND** operation of the Communication System to transport the message.

The run-time routine calls the **SYNCH_RECV** operation of the Communication System to wait for a response from the called entry point. The service queue that it waits on is known to the other task's run-time routine from the header of the sent message. When the **accept** statements are executed by the called task, and it is ready to send a response, it uses the **PID** to construct the service queue. Assuming a **PID** of 12345678, the queue name would be:

12345678_ENTRY_RESPONSE

Because a task can only wait on a single rendezvous at any point in time, the task would only be waiting for one message on the service queue. The processor name for routing the response is also found in the message header.

6. When the response message is received, the values passed back are copied into the appropriate arguments from the call. The task moves to the *RM* state. The task is allowed to continue executing with the statements in the **select** statement immediately following the entry call.

After the statements are executed, the task continues with the statement following the **select** statement. The task moves to the *CE* state.

4.3.2 Executing Accept Statements

A task can execute an **accept** statement either as a single statement, or in a **select** statement. Because the execution of an **accept** statement from within a **select** statement is the more interesting of these possible actions, it will be examined in this section.

Refer to Figure A-1 for the individual state diagram, and Table A-1 for the transition tables describing this construct.

1. The task begins at the **START** state. The run-time routines of the Language Interface evaluate the guard for each alternative, to determine if it is true. For each true guard, the routines add the associated **accept** entry name into the message to be sent to the Directory Server. After this message has been sent, the routines wait for a response to this message, by calling **SYNCH_RECV**. The response from the Directory Server indicates whether any rendezvous is possible, and if one is ready, the entry name is given. The Directory Server decides which entry is selected when multiple ones are possible, because only the Directory Server contains the state information concerning communicating tasks. Multiple tasks may issue outstanding **accepts** for the same entry point, and the Directory Server is responsible for correctly determining which tasks communicate.

Depending upon the response from the Directory Server, the run-time routines select one of the following alternatives.

- Waiting for a rendezvous. No entry is available for a rendezvous. (*WR*)
 - No rendezvous is possible, and an else clause is specified. (*EE*)
 - A rendezvous starts. (*EA*)
2. In state *WR*, the task is waiting for another task to call one of the entry points specified in the message sent to the Directory Server. The run-time routines wait

on the task's response queue for a message from the Directory Server, by employing a call to `SYNCH_RECV`. When a task calls an appropriate entry point, the Directory Server sends a message to the task, informing it of the service queue name that has a message. The service queue name corresponds to one of the entry points the task is waiting on. The run-time routines receive a message from the service queue, by calling `RECV_NOW`. They take the arguments from the message, and put them into the appropriate variables for the entry. The statements included in the `accept` statement in the alternative of the `select` statement are executed.

When the end of the `accept` statement is reached, the run-time routines construct a response message to send back to the caller. The parameters that are to be returned to the caller are put in the message. Only those arguments that are defined as `out` or `in out` are sent back to the caller, so that the amount of data in the message can be reduced. These arguments must be copied back into the corresponding arguments used by the caller. The processor name from the original message header is used to send the message back. The destination service queue is constructed, as described in Section 4.3.1 Step 5.

The response message is sent to the caller.

The task continues with the statement following the `accept` statement in the `select`. After the statements in the `select` alternative are executed, the task continues with the statement following the `select` statement. The task goes to the *CE* state.

3. If a timeout occurs while waiting for a rendezvous, the Directory Server sends a message to the task informing it of the event. The task moves to the *ED* state. The same activity as that specified in Section 4.3.1 Step 4 for a timeout on an entry call occurs in this instance.

4. If no rendezvous can occur immediately, and an else clause is specified, the task goes to the *EE* state. The statements associated with the else clause are executed.

The task continues with the statement following the select statement, and moves to the *CE* state.

4.4 Concurrent C

Concurrent C provides basically the same facilities as Ada. The main differences are in the ordering and selection of alternative **accept** statements in a **select** statement, by using the **suchthat** and **by** clauses. The differences are mainly in the scheduling of a rendezvous, and are not of interest in this paper. As a result, the implementation of the communication primitives in Concurrent C would be similar to the implementation in Ada.

An implementation of Concurrent C in a distributed system is discussed in [Cmelik], and it would be useful to compare that implementation with the one proposed in this paper.

Cmelik et.al. implemented Concurrent C in three different configurations, but the implementation for multiple processors connected through a Local Area Network (LAN) is most useful for the comparison, because it is similar to the implementation presented in this paper. In that implementation, a Concurrent C program is contained in multiple UNIXTM processes, one Unix process per processor. A "message reading daemon" Concurrent C process exists in each UNIX process to handle messages received from the other processors. A "master" UNIX process, that is created when the Concurrent C program starts execution, exists on the processor at the hub of a star configuration of processors on the LAN. Messages sent between the processors pass through the "master"

UNIX is a registered trademark of AT&T.

processor.

One difference between the two implementations is that there does not seem to be the concept of a Directory Server in [Cmelik]. But it is not clear from the paper how the Concurrent C processes determine the destination of a message.

Another difference is that Cmelik et.al. required an extra Concurrent C process to read messages from other processors. This function is provided by the Language Interface and the Communication System layers in this implementation. Each Concurrent C process can send a message to any other Concurrent C process directly.

A third difference exists in the manner that Concurrent C processes are structured. In [Cmelik], they are part of UNIX processes, and an additional scheduler was developed to schedule execution of the Concurrent C processes in the UNIX process. In this implementation, each Concurrent C process is a VMS process, so the VMS operating system scheduler handles the scheduling of the processes. The VMS scheduling algorithms are efficient, and are sufficient for scheduling of real-time processes. The process context switching time that [Cmelik] cites as the reason for multi-tasking in that implementation is not a problem in VMS, because the context switching is done very similarly to the way mentioned in [Cmelik] for the Concurrent C processes. That is, only registers are switched, and not the whole virtual memory and page tables for each process.

4.5 CHILL

As mentioned in Chapter Two, there are two methods for processes to communicate in CHILL. One is through buffers, and the other is through signals. In either case the SEND and RECEIVE operations are used to pass messages.

Because the operations appear similar between the operations in CHILL and in the

Communication System, it could be assumed that it would be simple to map the communication primitives in CHILL into operations in the Communication System. When the primitives are examined in detail though, it can be seen that they have some complicating features that make them nearly as difficult to implement as those in Ada.

The factors that make it difficult to implement the CHILL communications primitives in the Communication System are as follows.

1. Processes can send messages to either specific processes or process classes. This requires the use of a Directory Server, like in Ada, to locate processes that are willing to communicate.
2. Buffers and signals are persistent, which means that they must exist, even if no process is willing to receive them. The Communication System assumes that if no process attempts to receive on a temporary queue, and a message is sent to the queue, the message is lost, because the queue only exists while a process is connected to it. Permanent queues could be used, but then a process would be responsible for creating all possible queues that may be needed in a system.

The Communication System also requires that messages reside on a specific processor. If the receiving process is unknown at the time a message is sent, the Communication System would not know which processor to specify for receiving the message.

This problem requires that the messages be buffered outside of the Communication System. The messages could be held in the sender process until a receiver has been found for the message, but CHILL requires that messages outlast processes. In other words, the message will exist, even if the sender terminates.

One alternative would be to have the Directory Server maintain the messages. They

could be considered an extension of the state information that it must maintain. It also gives the Directory Server more control over blocking the sender when a buffer is full.

3. The sender must be blocked when a buffer of bounded size is full. The Communication System can return a status indicating a queue is full, but the status is only returned under some conditions. If both processes are on the same processor, or they are both on *cluster processors*², the status can be returned, because it is obtained by the Device Driver in the Communication System. If one process is on a *net processor*, the information cannot be provided, because the Network Server is not given the information to provide the status. Another mechanism, such as mentioned above, is needed to ensure that the sender is blocked at the appropriate time.

The state diagrams and transition tables for the communication primitives can be found in Appendix B. The following subsections describe the activities that occur within the Language Interface for each of the communication primitives.

4.5.1 Sending Buffer or Signal Messages

The following steps are taken for each call to **SEND**. The storage for buffer or signal messages is maintained by the Directory Server.

Because the Directory Server is responsible for the message storage, the only differences between sending signal or buffer messages are concerned with the generation of the messages. For buffer messages, the message is identified as a single parameter. For signal messages, the message consists of possibly many arguments, so the message must

2. Refer to Section 3.5.1 for information on the meanings for *cluster processor* and *net processor*.

be constructed from the arguments passed in the call. The problem here is very similar to the problem with passing arguments through an entry call in Ada.

The following steps describe the operations necessary to send messages. Refer to Figure B-1 for the individual state diagram, and Table B-1 for the transition tables.

1. The run-time routines begin at the *START* state. They send a message to the Directory Server providing information on the sending action. Depending upon the structure of the `send` statement, and whether a buffer or signal is specified, different information is provided to the Directory Server to help it decide upon the action to be followed.

If the `send` operation is on a buffer, only the buffer name and the priority are provided.

If the operation is on a signal, the signal name, process instance, or process class and priority are specified.

The process changes to the *CD* state by waiting for a response from the Directory Server. It issues a call to `SYNCH_RECV` to receive the message.

Depending upon the information returned by the Directory Server, the process advances to one of three states. They are:

- Buffer is full, wait for a free slot. (*SD*)
- No receiver available, send the message to the Directory Server. (*CM*)
- Receiver available, send the message directly to it. (*SM*)

2. When the process is in the *SD* state, the buffer is full. The process must wait until a slot, or a receiver, is available to take the message. A buffer is bounded only when a length is specified when it is defined. Otherwise, the buffer is unbounded,

and will never be full.

It is possible for a process to send a buffer message directly from this state to a receiver, if the priority of the message it is attempting to deliver is higher than the priority of any other message awaiting delivery.

A process in this state, then, can move to either the *CM* state when a slot is available, or the *SM* state when it is allowed to send its message directly to a receiver.

3. When no receiver is available, the process is in the *CM* state. In this state, the run-time routines sends the message to the Directory Server to store it.

This implementation allows the process to continue, or terminate, without affecting the message, or the possibility of its delivery.

A disadvantage is that the Directory Server is responsible for maintaining the buffers and signal storage areas, and this results in performance penalties. An optimization is made by allowing a process to send a message directly to another process whenever it is possible.

After the message is sent to the Directory Server, the process continues with the statement following the *send* statement. The process moves to the *CE* state.

4. If a receiver is waiting for the message, the process moves to the *SM* state, and the run-time routines send the message to the destination process. The destination processor is specified in the message sent by the Directory Server.

The process continues with the statement following the *send* statement. It moves to the *CE* state.

4.5.2 Receiving Buffer or Signal Messages

Processes can receive messages by using either the receive expression or the receive case statement. The receive expression can only be used to receive messages through buffers, and it results in an assignment of the message to a variable. The receive case statement is more complicated, and allows for more variations. Multiple buffers or signals can be waited on, and alternative statements can be executed. The following discussion focuses only on the receive case statement. It is relatively trivial to implement the receive expression.

The following steps describe the actions taken for receiving messages through either buffers or signals. The state diagram for the receive operation is in Figure B-2, while the transition tables are in Table B-2.

1. The process begins in the *START* state, by executing the receive case statement. The run-time routines send a message to the Directory Server, requesting a message from any of the alternative buffers or signals specified in the receive case statement. The run-time routines then wait for a response from the Directory Server on its response queue. Depending upon the response from the Directory Server, one of three transitions is possible. They are:
 - A message is available immediately. (*RM*)
 - No message is available, and no else clause is specified. (*WS*)
 - No message is available, and an else clause is specified. (*ES*)
2. If a message is available, the Directory Server sends it with the response message. The Directory Server also specifies the buffer or signal associated with the message, and the PID of the original sender. The process moves to the *RM* state.

If the **set** clause is specified, the sender PID is placed in the variable specified in the **set** clause.

If the **in** clause is specified, and the message is associated with a buffer, the message is copied into the location specified. If the message is associated with a signal, the values in the message are moved into the appropriate variables.

The process executes the statements associated with the signal or buffer indicated.

After execution of the statements, the process continues with the statement following the receive case statement. It moves to the *CE* state.

3. If the Directory Server indicates that no message is available for the specified buffers or signals, and no else clause is specified, the process moves to the *WS* state.

The run-time routines wait on a message from the Directory Server, by calling **SYNCH_RECV** on its response queue.

When a message is available, the Directory Server indicates the queue to use to receive the message directly from the sender. The run-time routines receive the message by calling **RECV_NOW**. The process then moves to the *RM* state.

4. If no message is available for the process, and an else clause is specified, The process goes to the *ES* state. It executes the statements associated with the else clause.

It then moves to the *CE* state, and executes the statement following the receive case statement.

4.6 Summary

This chapter has examined a possible implementation for utilizing communication

primitives available in Ada and CHILL in a distributed system. The issues involved, and some of the more important decisions have been discussed.

State diagrams and transition tables have been used to analyze the possible states for each operation. A composite state diagram was used to examine the interactions between communicating processes in each language.

By examining the composite state diagram, the nature of the communication can be identified. For example, the synchronous nature of a rendezvous in Ada becomes very obvious in the composite state diagram. Also, the asynchronous quality of communication in CHILL can be seen by the crossing of lines between state pairs, and no obvious single line of processing as was the case with the Ada diagram.

CHAPTER 5 - CONCLUSIONS

This chapter presents the conclusions that have resulted from the author's efforts in developing the sample implementation for the communication primitives. One conclusion concerns the usefulness of the state diagrams in developing and understanding the proposed implementation. They are also useful in all phases of a development, as discussed in Chapter Four. The implications of this conclusion are discussed in Section 5.1. Another conclusion involves the impacts on the Language Interface layer when either the communication system that it uses, or the language primitives that it supports, changes. Section 5.2 presents the implications of this second conclusion.

5.1 Usefulness of State Diagrams

The state diagrams had been very useful in the development of the sample implementation presented in this paper. They describe in a succinct manner how the Language Interface routines process the communication primitives of a high level language. The terse description of states and transitions allows the entire implementation to be viewed, and the descriptions clearly depict how the transitions are made between states. Of course, the transitions and states must be chosen by the developer, but the state diagrams can help to show when paths through the states become invalid. For example, in the state diagram for Ada in Figure A-1, it is obvious that the task cannot move from the ED_a state to the EA_a state, because, when the task starts to execute the statements for the `delay` alternative of the `select` statement, it aborts waiting for a rendezvous. If the task is not waiting for a rendezvous, it cannot execute the statements associated with an `accept` statement. If a transition is specified in a design between this pair of states, it would become obvious from an analysis of the state diagram that the

transition is invalid.

The possible flow of control within the Language Interface run-time routines is shown clearly with the state diagrams. These patterns of control help to show that the structure of the routines is correct, in the sense that each possible state is represented, and all allowable transitions are shown. Of course, it is easy to provide too much detail, and obscure the structure. This is one reason why states and transitions for the error conditions are not provided in the state diagrams for the Language Interface run-time routines.

Because of the languages studied, and their intended application areas, performance is an important consideration in the design of the support facilities for the languages. State diagrams can be used to determine the maximum expected communication delay between states. The transitions between states that involve passing messages between the Language Interface run-time routines and the Directory Server can have bounded communication times. These times would be the maximum transit times for message passing within the communication system. With the maximum delay times known, the designer could determine if the performance of the system is acceptable. They are also useful in program testing, because the tester can verify that a module developed meets the maximum delay times specified, or has reached an invalid condition, because the maximum delay time has been exceeded.

Composite state diagrams are also found to be useful, particularly in examining the interaction between communicating processes. They help to determine if states in the individual state diagrams are consistent with each other. They also help to determine the level of detail needed in the individual state diagrams to correctly represent operations in the run-time routines. This feature of the composite state diagram was discovered when the author constructed the composite state diagram for the Ada constructs, and found that

a state was missing in the individual state diagram for the `accept` statement. The SM_a state was not in the original individual state diagram, but the composite state diagram needed the state to show that the statements enclosed by the `accept` statement were executed, and that the process was to send a response back to the caller. The caller would be blocked until the response message was received. Without the additional state, the composite state diagram did not correctly show the point where the caller was unblocked and allowed to continue execution.

The composite state diagram clearly shows the structure of the communication primitives for each language. Examining the composite diagram for Ada in Figure A-3, it is evident where the rendezvous occurs, and that the calling and called processes are synchronized for the duration of the rendezvous. When the composite state diagram for CHILL in Figure B-3 is examined, it is obvious that there is less synchronization between the sending and receiving processes. Very few pairs of states are not allowed in CHILL, when compared to Ada. Three state pairs out of a total of 36 are not possible in CHILL, compared to ten out of 56 in Ada.

5.2 Impacts on Language Interface

Because the Language Interface translates the communication primitives in a language to the services available in a communication system, it can be significantly impacted if either the primitives or the communication system changes. The functions performed by the Language Interface are driven both by the requirements placed upon it by the language primitives and by the capabilities of the communication system. This section reviews the potential impacts on the Language Interface, as a result of different types of changes.

It has been shown that the Language Interface can successfully support the communication primitives of Ada, Concurrent C and CHILL, using the communication system presented in Chapter Three, provided that the distributed database issues in the

Directory Server, and the error handling issues, discussed in Chapter Four are resolved. The author believes that they are resolvable, because extensive research has occurred in these areas.

We have examined two types of communication primitives in this paper. One is the synchronous style of communication in the rendezvous communication mechanism in Ada and Concurrent C, where the sender and receiver are synchronized when the rendezvous occurs. The other type is asynchronous in nature, as displayed in the communication primitives of CHILL, where only the receiver is blocked while waiting for a message. Other styles of communication, such as totally asynchronous communication, where the receiving of messages is also asynchronous, would require corresponding changes in the Language Interface.

Presumably, the Directory Server would not change drastically, because the mapping service provided by it would probably be required by other languages. The flexibility offered by the incorporation of the Directory Server is a major feature in the design. It requires the Language Interface to be concerned about the location of a destination process, and relieves the application programmer of that burden.

Changes in the communication system would also impact the Language Interface. A more reliable communication system may be used, but the overhead required to ensure message delivery may be greater than the overhead that results from the current design. In other words, the reliable communication system may provide more reliability than is actually needed, or, because of its general applicability, may provide features that are not needed by the Language Interface. These unnecessary features may incur an undesirable performance penalty. Because the languages that the Language Interface supports are intended for real-time applications, performance is an important consideration for the run-time routines.

The communication system used in the proposed implementation provides a number of different types of receive operations. It allows asynchronous and synchronous receive operations, as well as alternatives in the synchronous case, such as timeouts and no wait receive operations. If another communication system is used that does not offer this variety of receive operations, the Language Interface would have to change.

In essence, any change in the communication system, that does not offer the same capabilities in performance or allowable operations as the current Communication System provides, would cause changes in the design of the Language Interface. Depending upon the capabilities in the new communication system, it may be possible that the Language Interface may not be able to meet its requirements, particularly if the performance of the replacement communication system is not acceptable, or the operations available do not allow the Language Interface to function properly.

REFERENCES

- [Andrews 82] Andrews, G. R., "The distributed programming language SR - Mechanisms, design, and implementation.", *Software: Practice and Experience*, Volume 12, Number 8, August, 1982, pp. 719-754.
- [Andrews 83] Andrews, G. R. and Schneider, F. B., "Concepts and notations for concurrent programming.", *Computing Surveys*, Volume 15, Number 1, March 1983, pp. 3-43.
- [Balzer 71] Balzer, R. M., "PORTS - A method for dynamic interprogram communication and job control.", *Proceedings AFIPS Spring Joint Computer Conference*, Atlantic City, N. J., May 18-20, 1971, Volume 38, AFIPS Press, Arlington, VA, 1971, pp. 485-489.
- [Bernstein 81] Bernstein, P. A. and Goodman, N., "Concurrency Control in Distributed Database Systems.", *Computing Surveys*, Volume 13, Number 2, June 1981, pp. 185-222.
- [Brinch Hansen 77] Brinch Hansen, P., *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1977.
- [Brinch Hansen 78] Brinch Hansen, P., "Distributed processes: A concurrent programming concept.", *Communications of the ACM*, Volume 21, Number 11, November, 1978, pp. 934-941.
- [Burns 87] Burns, A., Lister, A. M. and Welling, A. J., *A Review of ADA Tasking*, Volume 262 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1987.
- [CCITT 80] CCITT Study Group XI, *Introduction to CHILL*, The International Telegraph and Telephone Consultative Committee, Geneva, Switzerland, May 1980.
- [CCITT 85] *Network Service Definition for Open Systems Interconnection (OSI) for CCITT Applications - X.213*, The International Telegraph and Telephone Consultative Committee, Geneva, 1985.
- [Cmelik] Cmelik, R. F., Gehani, N. H. and Roome, W. D., "Experience with distributed versions of concurrent c.", AT&T Bell Laboratories internal memorandum, Murray Hill, NJ.
- [Date 83] Date, C. J., *An Introduction to Database Systems*, Volume II, Addison-Wesley, Reading, MA, 1983.
- [DEC 86a] Digital Equipment Corporation, *VAX/VMS Networking Manual*, Maynard, MA, April 1986, Order Number AA-Y512C-TE.
- [DEC 86b] Digital Equipment Corporation, *Writing a Device Driver for VAX/VMS*, Maynard, MA, April 1986, Order Number AA-Y511B-TE.
- [Filman 84] Filman, R. E. and Friedman, D. P., *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill Book Company, New York, 1984.

- [Gehani 86] Gehani, N. H. and Roome, W. D., "Concurrent C.", *Software - Practice and Experience*, Volume 16, Number 9, September 1986, pp. 821-844.
- [Gelernter 82] Gelernter, D. and Bernstein, A. J., "Distributed communication via global buffer.", *Proceedings of the Symposium on the Principles of Distributed Computing*, Ottawa, Canada, August 18-20, 1982, ACM, New York, 1982, pp. 10-18.
- [Hoare 78] Hoare, C. A. R., "Communicating sequential processes", *Communications of the ACM*, Volume 21, Number 8, August 1978, pp. 666-677.
- [Kenah 88] Kenah, L. J., Goldenberg, R. E. and Bates, S. F., *Version 4.4 VAX/VMS Internals and Data Structures*, Digital Press, Bedford, MA, 1988.
- [Kohler 81] Kohler, W. H., "A survey of techniques for synchronization and recovery in decentralized computer systems.", *ACM Computing Surveys*, Volume 13, Number 2, June 1981, pp. 149-183.
- [Liskov 82] Liskov, B. L. and Scheifler, R., "Guardians and actions: Linguistic support for robust, distributed programs.", *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 25-27, 1982, ACM, New York, 1982, pp. 7-19.
- [Petersen 83] Petersen, J. L. and Silberschatz, A., *Operating System Concepts*, Addison-Wesley, Reading, MA, 1983.
- [Sloman 87] Sloman, M. and Kramer, J., *Distributed Systems and Computer Networks*, Prentice-Hall International, Englewood Cliffs, NJ, 1987.
- [Solomon 79] Solomon, M. H. and Finkel, R. A., "The Roscoe distributed operating system.", *Proceedings of the Seventh Symposium on Operating System Principles*, Pacific Grove, Ca, December 10-12, 1979, ACM, New York, 1979, pp. 108-114.
- [Sunshine 75] Sunshine, C. A., *Interprocess Communication Protocols for Computer Networks*, Technical Report Number 105, Digital Systems Laboratory, Stanford Electronics Laboratory, Stanford University, December 1975.
- [Tanenbaum 81] Tanenbaum, Andrew S., *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
- [USDoD 83] U. S. Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, American Standards Institute, Inc., February 17, 1983.
- [Wecker 80] Wecker, S., "DNA: The Digital network architecture.", *IEEE Transactions on Communications*, Volume COM-28, Number 4, April 1980, pp. 510-526. Volume 11 1981, pp. 257-290.

APPENDIX A

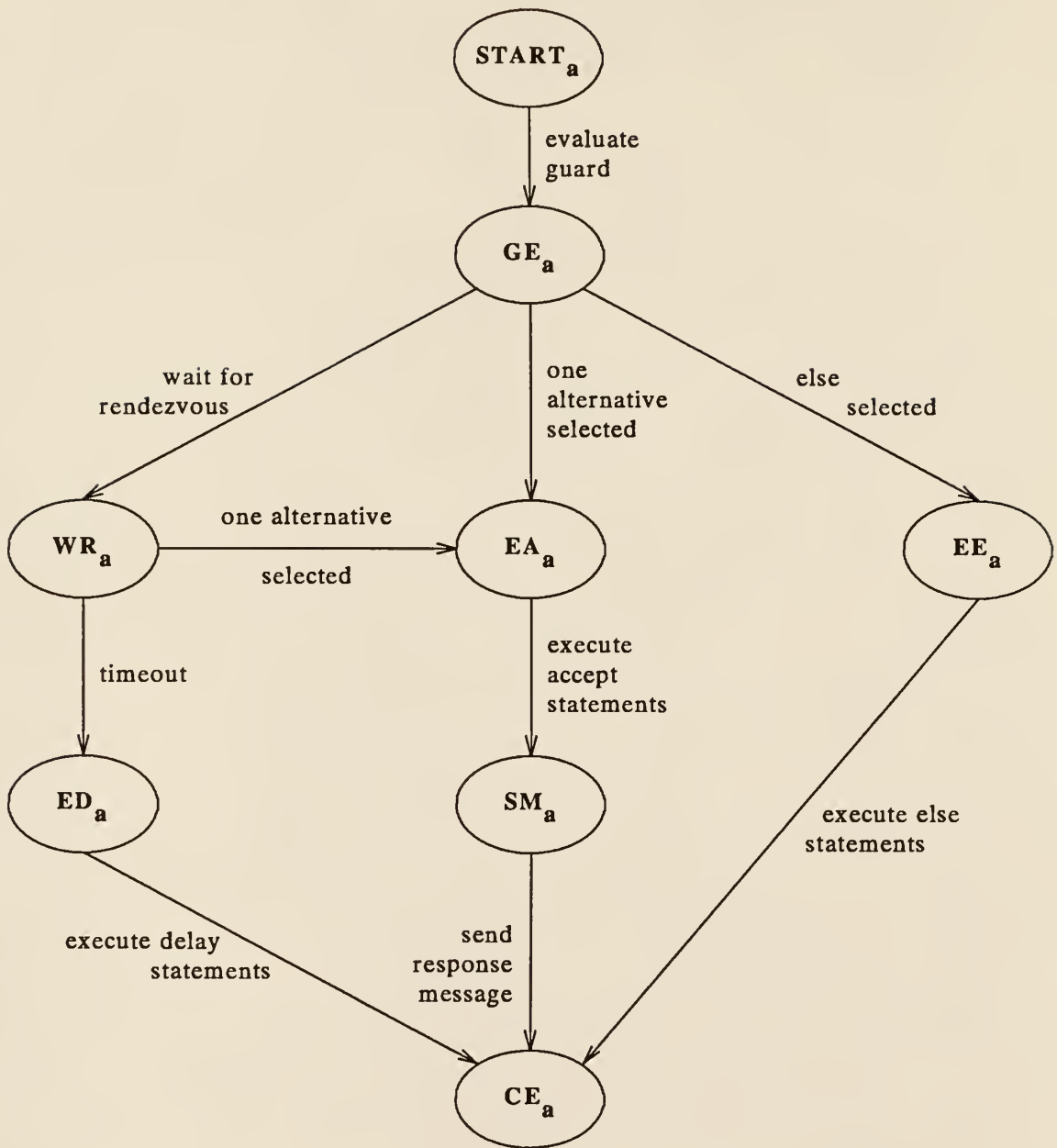


Figure A-1. State Diagram for Ada Accept Select Statement

APPENDIX A

Table A-1. State Transition Table for Ada Accept Select Statement

Current State	Event	Next State	Action
START _a	evaluate guards	GE _a	The when clause is evaluated for each alternative. The Directory Server is called with true alternatives.
GE _a	else selected	EE _a	No processes available for a rendezvous, and an else clause is specified.
	wait	WR _a	The process waits for a rendezvous on one of the true alternatives.
	alternative selected	EA _a	The Directory Server informs the process that another process is ready for a rendezvous on an accept statement.
EE _a	execute else	CE _a	The statements associated with the else clause are executed.
WR _a	alternative selected	EA _a	The Directory Server informed the process that another process is ready for a rendezvous on an accept statement.
	timeout	ED _a	The process specified a delay , and it expired before a rendezvous started.

APPENDIX A

Table A-1. State Transition Table for Ada Accept Select Statement (Continued)

Current State	Event	Next State	Action
EA _a	execute accept statements	SM _a	Execute the statements associated with the accept that was completed.
ED _a	execute delay statements	CE _a	The statements associated with the delay alternative are executed.
SM _a	send message	CE _a	The response message is sent back to the process involved in the rendezvous.
CE _a	---	---	The end of the select statement has been reached. The process continues execution.

APPENDIX A

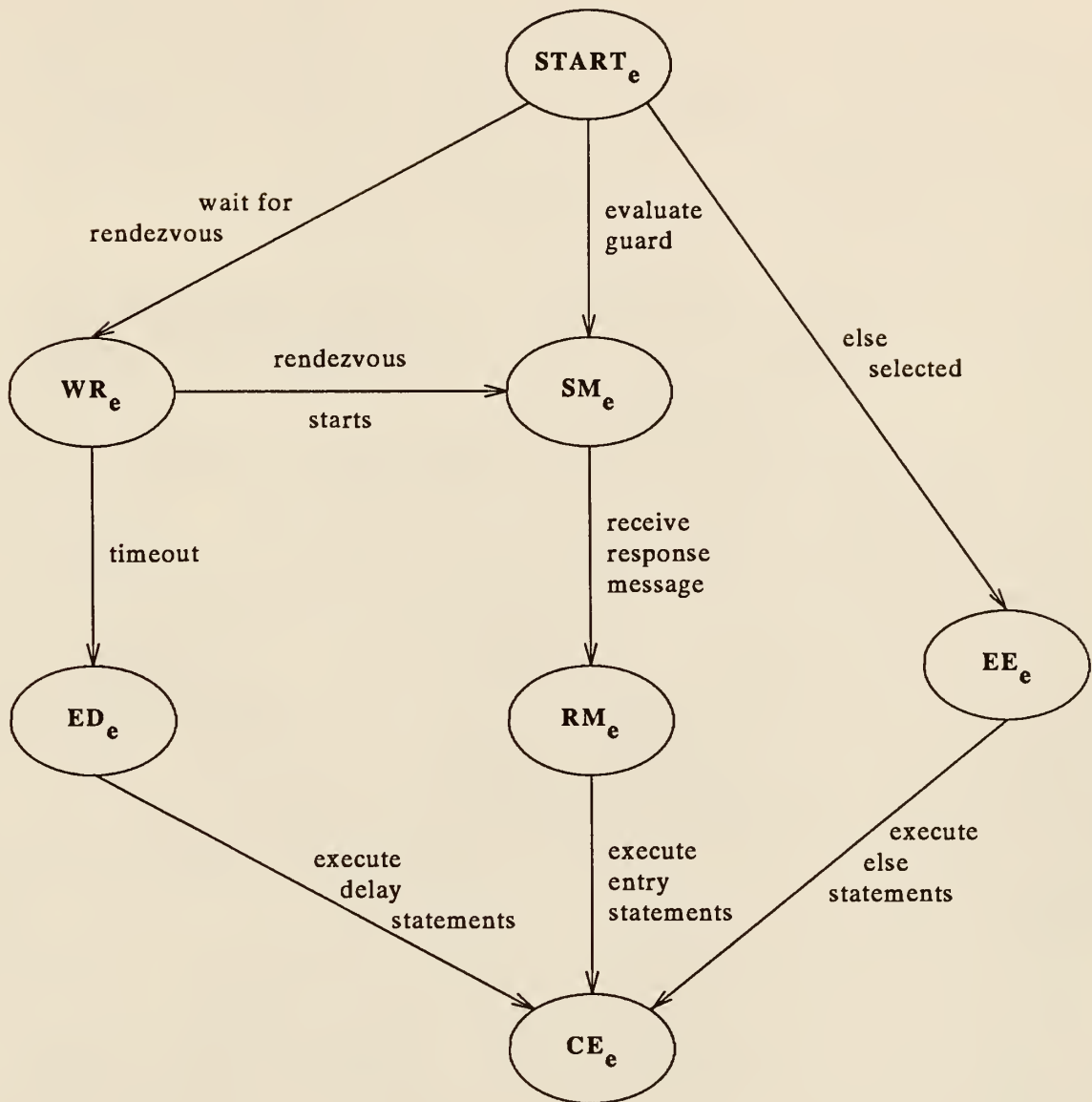


Figure A-2. State Diagram for Ada Entry Call Select Statement

APPENDIX A

Table A-2. State Transition Table for Ada Entry Call Select Statement

Current State	Event	Next State	Action
START _e	execute else statements	EE _e	The process called the Directory Server for available processes for a rendezvous, and no process is available. An else clause is specified.
	wait for a rendezvous	WR _e	Waiting for a rendezvous because none is available, and no else clause is specified.
	rendezvous starts	SM _e	The Directory Server identifies a process available for a rendezvous. This process sends a message to the designated process. It then waits for a response message from the other process.
EE _e	execute else statements	CE _e	The statements associated with the else clause are executed.
WR _e	timeout	ED _e	The process timed out before a rendezvous started.
	rendezvous started	SM _e	The Directory Server process sent a message indicating that a process is ready for a rendezvous. The process sends a message to the other process.

APPENDIX A

Table A-2. State Transition Table for Ada Entry Call Select Statement (Continued)

Current State	Event	Next State	Action
ED_e	execute delay statements	CE_e	The statements associated with the delay alternative of the select statement are executed.
SM_e	receive response	RM_e	The response message has been received from the other process.
RM_e	execute statements	CE_e	The statements following the entry call are executed.
CE_e	---	---	Continue execution of the process with the statement following the end of the select statement.

APPENDIX A

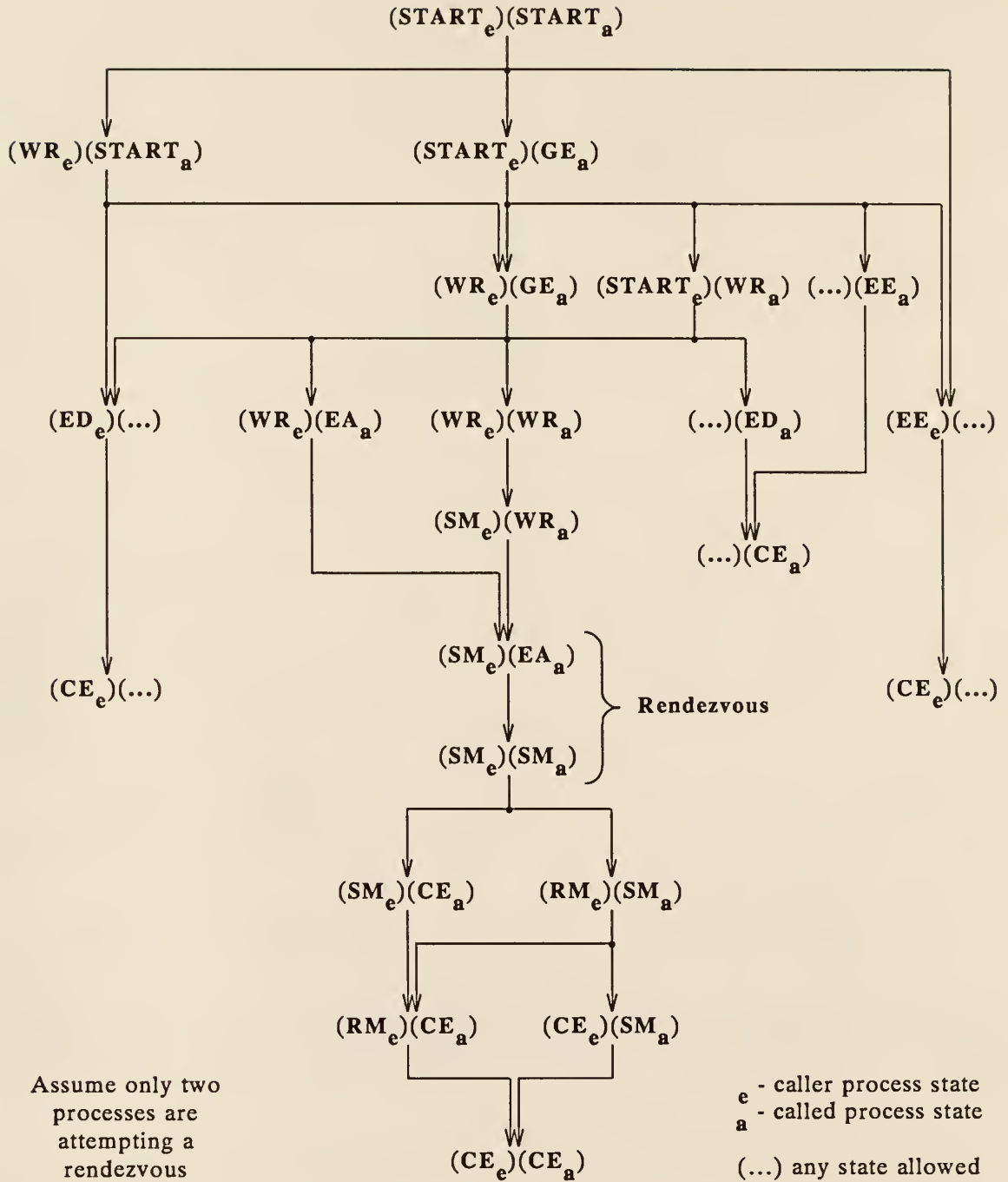


Figure A-3. Composite State Diagram for Ada Rendezvous

APPENDIX B

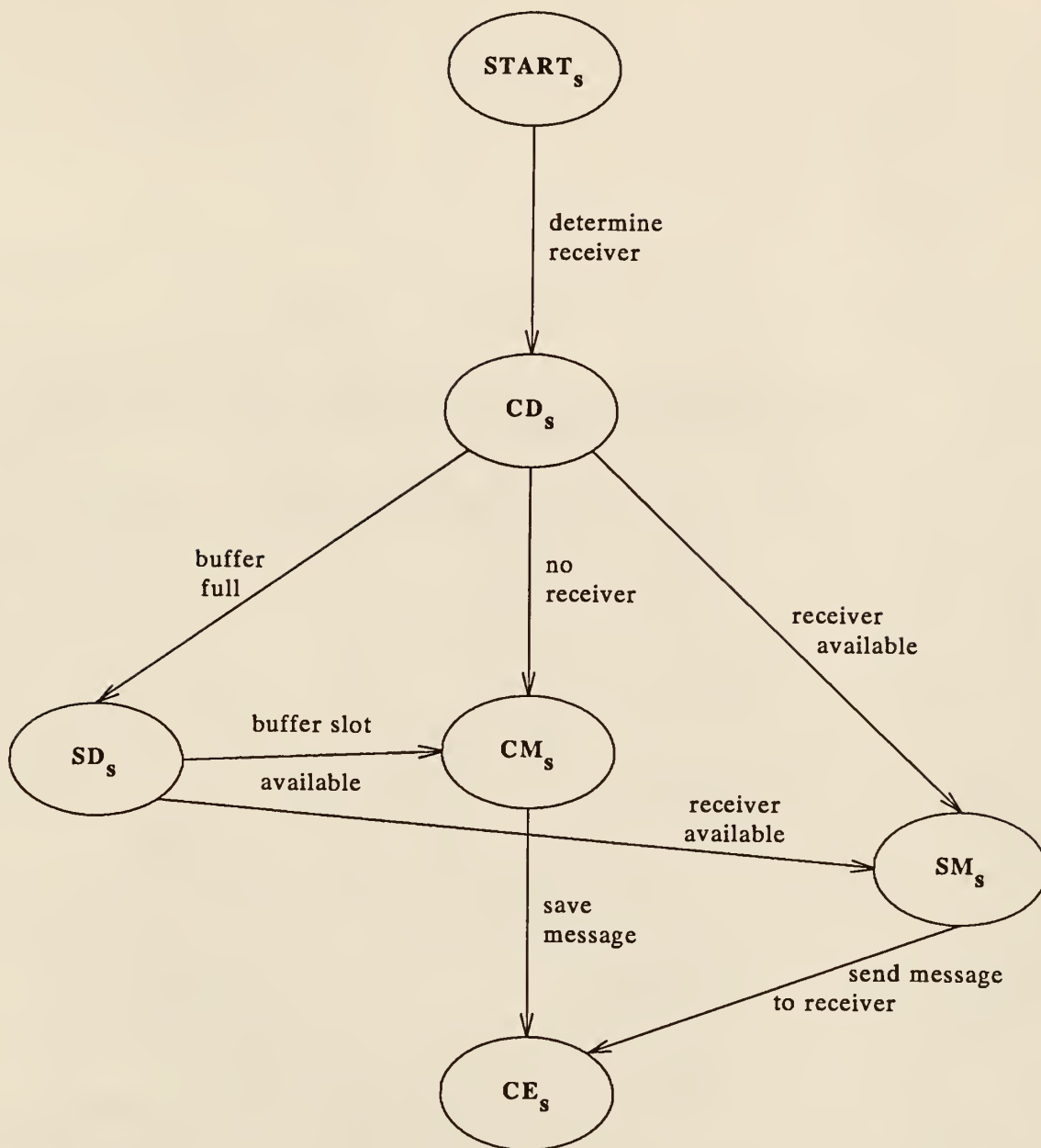


Figure B-1. State Diagram for CHILL Send Statements

APPENDIX B

Table B-1. State Transition Table for CHILL Send Statements

Current State	Event	Next State	Action
START _s	determine receiver	CD _s	The sending process sends a message to the Directory Server process to determine if a process has an outstanding receive on the buffer or signal.
CD _s	receiver available	SM _s	A receiver is available to take the message immediately.
	no receiver available	CM _s	No receiver is available.
	buffer full	SD _s	The buffer is full. The sender is blocked, waiting for a receiver to either remove a message, or take its message.
SM _s	send message	CE _s	The message is sent to the selected process.
CM _s	copy message	CE _s	Copy the message into a storage area to hold it for an ultimate receiver.
SD _s	receiver available	SM _s	A receiver is available, and the priority of the send is higher than any other messages available. The message is sent from sender process to the receiver process.
	buffer available	CM _s	Buffer space is available to hold the message.

APPENDIX B

Table B-1. State Transition Table for CHILL Send Statements (Continued)

Current State	Event	Next State	Action
CE_s	---	---	The sender is allowed to continue execution after the send statement.

APPENDIX B

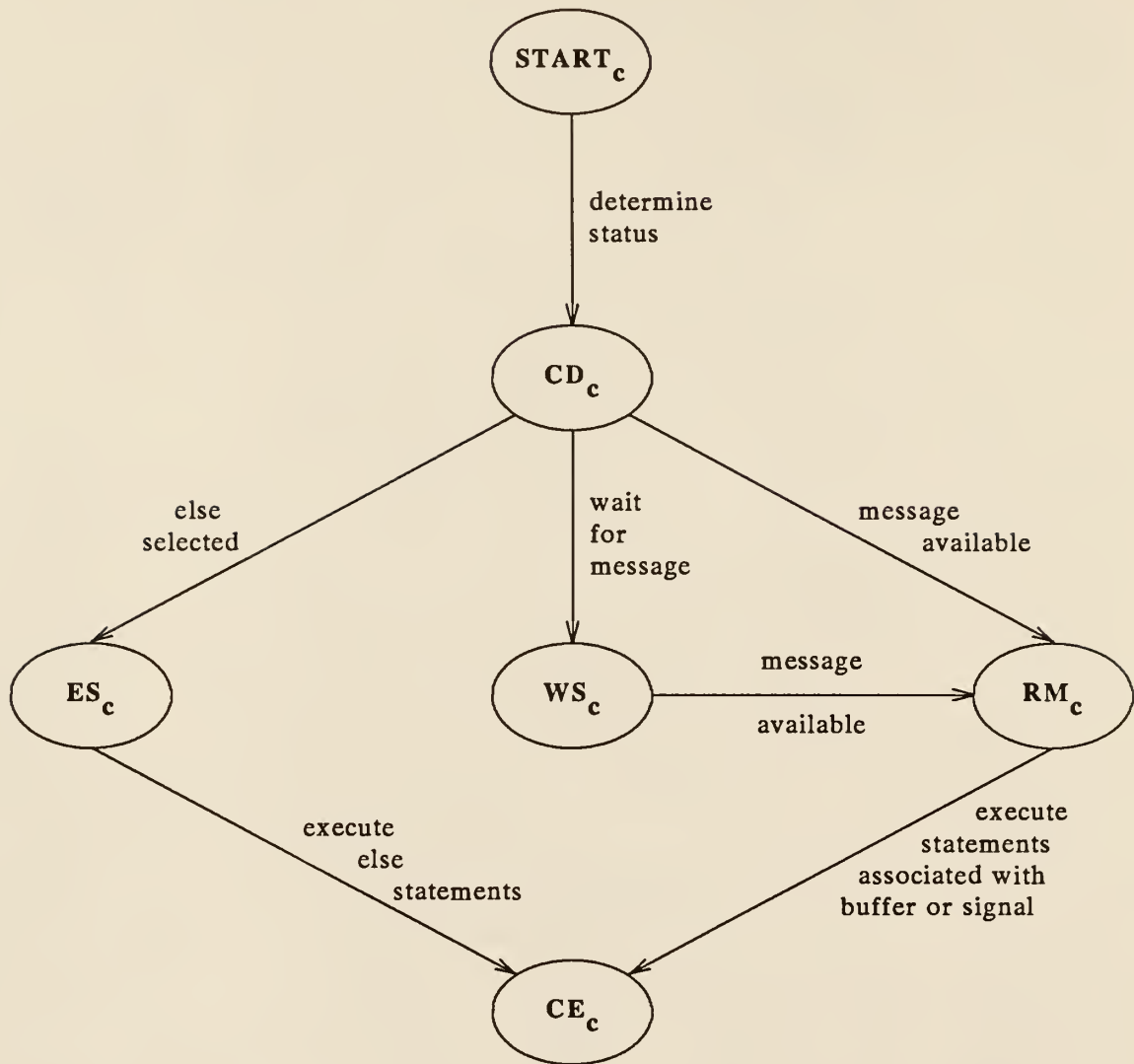


Figure B-2. State Diagram for CHILL Receive Statements

APPENDIX B

Table B-2. State Transition Table for CHILL Receive Statements

Current State	Event	Next State	Action
START _c	determine sender	CD _c	The receiver sends a message to the Directory Server process to determine if a message is available for reception.
CD _c	message available	RM _c	A message is available. Receive it from the indicated buffer or signal.
	wait for message	WS _c	The process waits for a message to be sent to any of the buffers or signals specified. It waits for a message from the Directory Server to determine the source of the message.
	else clause selected	ES _c	No messages are available in any of the buffers or signals specified, and an else alternative is specified.
RM _c	execute receive statements	CE _c	If a signal was specified, and, an IN clause was specified, the values in the message are copied to the variables specified. If a buffer was specified, and the IN clause was specified, copy the message into the variable identified in the IN clause. The statements associated with the buffer or signal that the message was received from are executed.

APPENDIX B

Table B-2. State Transition Table for CHILL Receive Statements (Continued)

Current State	Event	Next State	Action
WS_c	message available	RM_c	A message is available from one of the alternative buffers or signals.
ES_c	execute else statements	CE_c	Execute the statements associated with the else alternative.
CE_c	---	---	Continue execution of the process with the statement following the receive case statement.

APPENDIX B

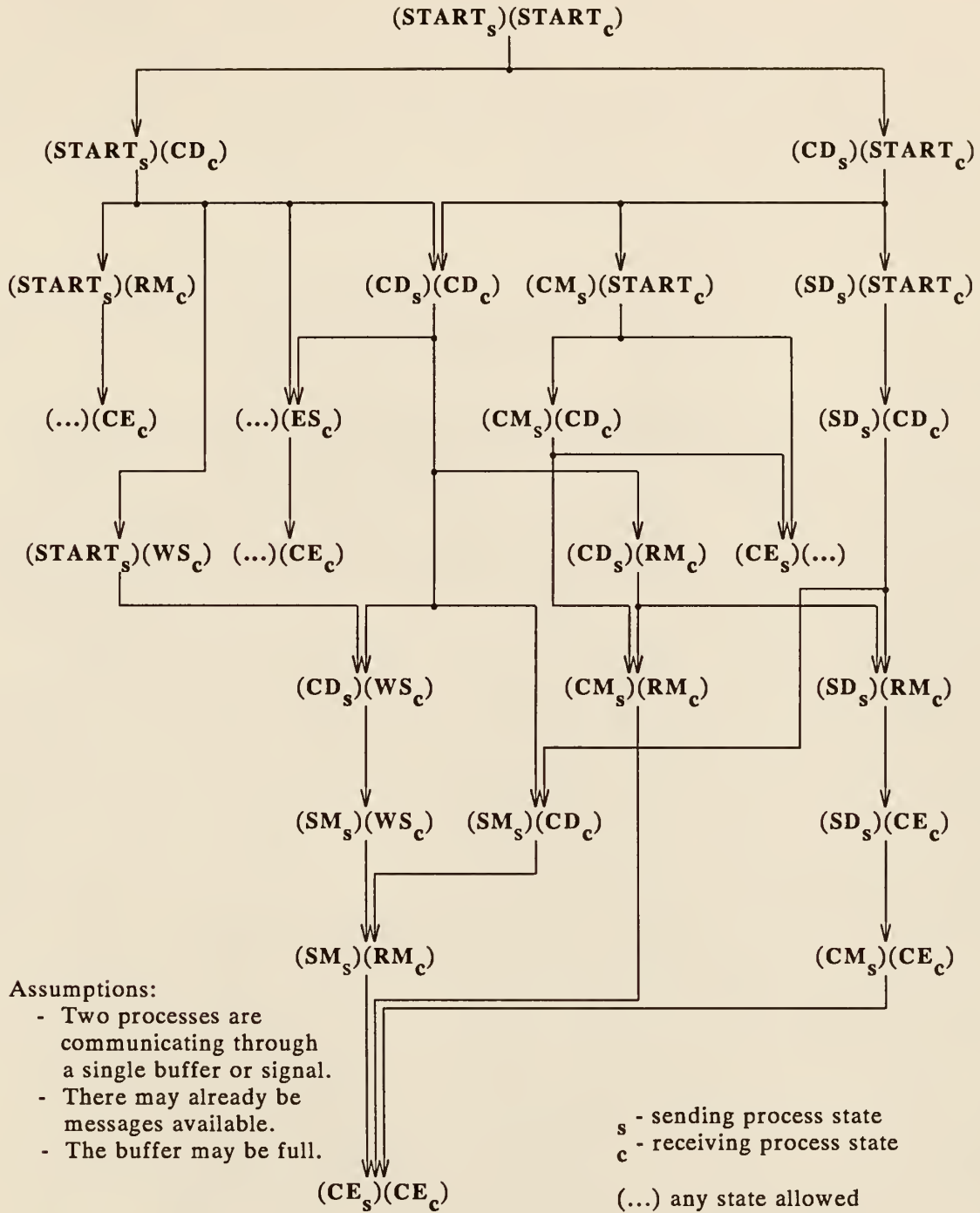


Figure B-3. Composite State Diagram for CHILL Communications

APPENDIX C

ADDITIONAL DETAILS ON THE COMMUNICATION SYSTEM

C.1 Major Software Components

The following subsections describe the Device Driver and Network Server in more detail than the discussion of these components in Section 3.4.

Miscellaneous operations available in the communication system are presented in Section C.2. These operations are presented to show the additional services that are available in the Communication System, which can aid the applications programmers.

C.1.1 Device Driver

A device driver is a system-level process that operates in kernel mode to manipulate the data structures for which it is responsible. Interrupts are received by a device driver when a service is requested from it. Refer to [DEC 86b] and [Kenah 88] for information on the capabilities of device drivers in the VMS operating system.

If the processor is in a VAXcluster, the Device Driver will also attempt to connect to the other members of the cluster when the processor is first booted. Also, if connection requests are delivered from other processors, it will respond to them, and establish the connection. This allows all device drivers in a VAXcluster to be completely interconnected. Each Device Driver maintains a table of all processors to which it is connected, and it is responsible only for those service queues that are resident on its processor.

APPENDIX C

C.1.2 Network Server

The Network Server only connects to processors that cannot be connected through the Device Driver. Processors that can only be connected through DECnet use the Network Server to pass messages. Processors that are in the *cluster group* communicate with processors in the *net group* through the Network Server.

C.1.2.1 Establishing Connections A configuration table is used by the Network Server to determine the processors to which it must attempt to connect. The configuration table contains the maximum configuration of the processors, including processors that only connect to the network occasionally.

If the Network Server cannot connect to a specified processor in a number of connection attempts, it will log an alarm message.³ Even after issuing the alarm message, it will still attempt to connect, but at a slower rate.⁴ This method allows for machines that enter and leave the network at different times. It also handles cases where the physical connection between machines may be down for an extended period of time.

C.1.2.2 Maintaining Connections If the Network Server is connected to another machine, and then subsequently loses the connection, it will attempt to reconnect, using the algorithm mentioned above.

If it cannot reconnect within two attempts, it assumes the other processor has crashed,

3. The alarm message is a warning to the operator. If the processor is known to be down, the operator can ignore the message. If the processor is functioning normally, the operator will need to analyze and correct the problem.

4. The normal rate to attempt connections is once every 2 minutes, while the long-term rate is once every 10 minutes. The normal rate is adjustable, whereas the long-term rate is always 5 times as long as the normal rate.

APPENDIX C

and issues a major alarm to alert the operator. If it can reconnect immediately, it assumes the connection loss was transitory.

C.1.2.3 Special Routing Algorithms The Network Server can receive messages either from processes on the local machine, or from any of its partner Network Servers. When it receives a message from a local process, it needs to determine to which processor(s) to send the message.

The algorithms used to map logical names to physical names are described in Section 3.5.

C.1.2.3.1 Locally Received Messages The Network Server receives messages locally from a service queue, much as any other process in the system. When a message is received, the Network Server is interrupted to process the message.

If the destination processor is the local processor, the Network Server will send the message to the indicated service queue directly.

If the destination processor is a symbolic name, the Network Server determines how to identify the processor(s) that constitute the set for the symbolic name.

If the destination is a processor to which the Network Server is connected, it sends the message to its partner Network Server on the destination processor. The Network Server on the other processor will process the message in the manner described in the next section.

C.1.2.3.2 Remotely Received Messages When a message is received from a remote processor, the Network Server is interrupted to process the message. The server then determines if the ultimate destination is the local machine, or another processor. The message may have been directed to a site other than its final destination for one of the

APPENDIX C

following reasons.

- An alternate route was selected by the sending Network Server, because it was not connected to the specified destination processor.
- The Network Server believed that the coordinator processor is the local processor⁵.

If the ultimate destination is another processor, the Network Server attempts to send the message to the correct processor. If it cannot, it discards the message. Note that in this case the original sender of the message is not notified of this action.

If the message is destined for the local processor, the Network Server sends the message to the service queue specified in the header of the message.

C.2 Miscellaneous Operations

In addition to the operations described in Sections 3.5 and 3.6, a number of operations are available. They had been developed to aid the applications developers in the common functions that they perform.

The two operations, `DRAIN_QUE` and `CANCEL_IO`, are described in the following subsections. These are the only additional operations available at the present time.

C.2.1 Empty Service Queue of Messages

One operation, `DRAIN_QUE`, removes and discards all the messages in a service queue. This operation allows the developer to empty a queue of old messages that are not meaningful to it. They may be messages intended for an old incarnation of the process,

5. The coordinator processor is a special logical designation, and is discussed in Section 3.5.

APPENDIX C

and the new instantiation does not have any state information to correctly interpret the information in the messages. Or, they may be responses to old request messages and the process timed out waiting for the response. In the latter case, the process will normally drain the queue before making a new request.

The format of the `DRAIN_QUE` is shown in Figure C-1.

```
DRAIN_QUE( que );
```

where:

que The name of the service queue to be emptied.

Figure C-1. `DRAIN_QUE` Call Format

C.2.2 Cancel I/O for a Service Queue

This operation is useful in the case where a process issues receive requests on a number of service queues. The requests are all done through calls to `ASYNC_RECV`. The completion of the call will occur inside an AST, as specified in Section 3.6.1. The process is interested in receiving a message on any of the queues, but only needs one message. When a message is received on a queue, the outstanding requests on the other queues are canceled.

Outstanding requests that are canceled return to the caller, through the AST, with a special status. The AST must handle this status correctly, which in most cases means to ignore the receive operation.

APPENDIX C

The format of the operation is given in Figure C-2.

```
CANCEL_IO( que );
```

where:

que The name of the service queue on which the process has outstanding receive requests.

Figure C-2. Format of CANCEL_IO Operation

A PROPOSED IMPLEMENTATION OF COMMUNICATION PRIMITIVES
IN ADA AND CHILL FOR DISTRIBUTED SYSTEMS

by

JOHN WILLIAM UNGER

A. B., Rutgers University, 1978

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

A PROPOSED IMPLEMENTATION OF COMMUNICATION PRIMITIVES
IN ADA AND CHILL FOR DISTRIBUTED SYSTEMS

by John William Unger

AN ABSTRACT OF A MASTER'S THESIS

As computing systems have become more distributed, communications between processes have become more important. Languages have been developed which include communication primitives as part of their syntax, to provide for communicating processes. These primitives are not always suitable for distributed communicating processes.

This paper examines the communication primitives in a number of these languages, namely Ada, Concurrent C and CHILL. It also presents a communication system that had been developed for homogenous computer networks consisting of processors using the Digital Equipment Corporation's VMS operating system and DECnet network protocol.

The paper introduces a sample implementation which maps the communication primitives of Ada and CHILL into the communication system described, to illustrate the issues involved with distributed processing. Possible alternative solutions are discussed in order to show that tradeoffs need to be examined when the opposing requirements for reliability and performance conflict.

