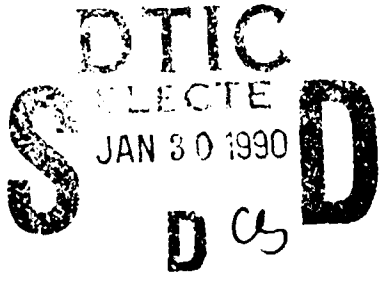
(4)

NUSC Technical Report 6834
11 October 1989

Issues and Approaches in the Design of Distributed Ada Programs

J. W. Brennan Jr.
Combat Control Systems Department



Naval Underwater Systems Center
Newport, Rhode Island / New London, Connecticut

PREFACE

This report was prepared under NUSC Project No. E45146, "Next Generation Computer Resources (NGCR)," principal investigator J. Oblinger (Code 2211). The sponsoring activity is the Space and Naval Warfare Systems Command (SPAWAR 324).

The technical reviewer for this report was B. W. Stevens (Code 2211).

REVIEWED AND APPROVED: 11 OCTOBER 1989

J. R. Short
Head, Combat Control Systems Department

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	11 October 1989	Final

4. TITLE AND SUBTITLE
Issues and Approaches in the Design of Distributed Ada Programs

5. FUNDING NUMBERS
PR E45146

6. AUTHOR(S)
J.W. Brennan Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Underwater Systems Center
Newport Laboratory
Newport, RI 02841-5047

8. PERFORMING ORGANIZATION REPORT NUMBER
TR 6834

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Space and Naval Warfare Systems Command
SPAWAR 324
Washington, DC 20363-5100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The design of Ada programs across multiple processors is not fully supported by current Ada run-time environments. This report examines the relevant issues facing the designers of such programs and discusses the approaches to program design. A distributed Ada interprocessor message-passing kernel was designed and implemented to support the design of distributed programs and to help illustrate these issues. Implementations of the Ada constructs based on the use of the kernel are suggested, and several possible approaches to the partitioning of Ada programs across multiple processors are discussed. This report attempts to resolve these issues and presents several viable approaches to the production of portable, reusable distributed Ada programs, utilizing the full advantages provided by the Ada language and current programming support environments, while requiring only currently available compilers and run-time environments.

14. SUBJECT TERMS
Distributed Computer Programs
Programming Languages
Ada

15. NUMBER OF PAGES
79

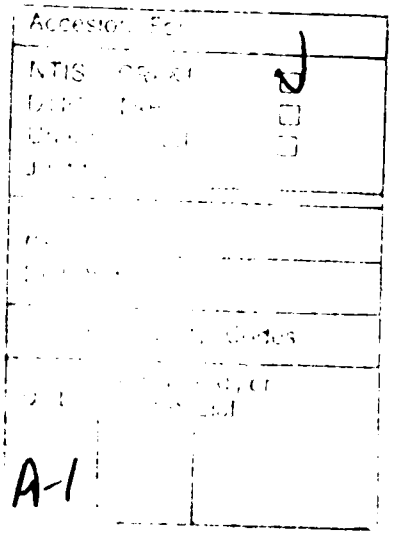
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

LIST OF ILLUSTRATIONS



LIST OF ABBREVIATIONS AND ACRONYMS

ACK	Positive acknowledgment
ANSI	American National Standards Institute
APPL	Ada program partitioning language
APSE	Ada programming support environment
ARQ	Automatic repeat request
CPU	Central processing unit
FDDI	Fiber-optic distributed data interface
ID	Identifier
I/O	Input/output
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Standards Organization
LAN	Local area network
LLC	Logical link control
LME	Layer management entity
LSAP	Logical link service access point
MAC	Medium access control
NAK	Negative acknowledgment
OSI	Open systems interconnection
RPC	Remote procedure call
RTE	Run-time environment
SAFENET	Survivable adaptable fiber-optic embedded network
SAP	Service access point
TSAP	Transport service access point
VLSI	Very large scale integration
VMTP	Versatile message transaction protocol
XTP	Express transfer protocol
WAN	Wide area network

ISSUES AND APPROACHES IN THE DESIGN OF DISTRIBUTED Ada PROGRAMS

1. INTRODUCTION

The design of distributed Ada* programs across multiple processors is not fully supported by current Ada run-time environments. The present tendencies have been to extensively modify the Ada compiler, either directly or by the addition of one or more compiler pragmas, or tailor the vendor's run-time system to support the application. This methodology results in application-specific run-time environments (RTEs) and compiler-dependent or RTE-dependent application programs. Essentially, multiple Ada dialects are being generated, making the Ada programs written in these dialects non-portable and hardly reusable. This report examines the relevant issues and approaches to the production of portable, reusable, distributed Ada programs.

Although the Ada language was designed to implement embedded computer systems, Ada provides little help with loosely coupled distributed environments. Extensions to the Ada language have been proposed to support distributed Ada processing, but it will be several years before a new version of Ada is available. Also, current interprocessor communication facilities do not adequately support the real-time requirements of many Ada applications. Clearly, a kernel designed to meet the needs of the Ada language would be a valuable asset to designers of distributed Ada programs. Such a kernel was designed and implemented in this study to support the design of distributed Ada programs and to help illustrate associated issues and possible approaches.

This report begins with a discussion of some of the problems that face software developers when the Ada language is applied to the distributed environment and how a distributed Ada kernel should support the use of the language (section 2). Next, the design and implementation of the actual kernel are presented, together with some of the major design issues that were encountered (section 3). The report then presents an implementation of each Ada construct across multiple processors based on the use of the kernel (section 4). In light of the implementations of the Ada constructs, several possible approaches to the construction of a distributed program are then presented and discussed (section 5). Finally, several global issues are discussed as they relate to the application of Ada to the distributed environment (section 6).

---

*Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

2. Ada AND THE DISTRIBUTED ENVIRONMENT

2.1 DEFICIENCIES OF Ada WITH DISTRIBUTED SYSTEMS

The Ada language is designed to implement embedded computer systems. This feature of the Ada language is important in the development of modular portable software. The Ada multitasking model provides a means of specifying logical concurrency. Since the real world functions in terms of concurrent activities, this is a natural approach to the solution of real-world problems. However, when applying the Ada multitasking model to a distributed environment,* several difficulties arise due to the fact that the distributed environment is seldom considered to be truly embedded. That is to say, there is no single, centralized, universal operating system or any universally shared global memory. Ada provides no specific solutions to the radically different distributed architectures possible. The weaknesses of Ada in this respect are as follows:

1. Tasks may communicate in ways other than the Ada rendezvous. Many tasks in time-critical systems use shared memory to reduce the communication overhead and global memory may not be accessible to all tasks in the network.

2. The Ada language assumes the existence of a centralized heap. This is seldom the case in a distributed environment, so the passing of access types from a task on one processor may have absolutely no meaning to a task on a different processor. Of further complication are access types that can be embedded (and often are) within more complex data structures.

3. The conditional and timed entry calls across processors cannot be directly implemented. Using a communication link implies a wait that must be considered (reference 1). Any asynchronous communication using conditional or timed entry calls cannot be nonblocking, since the message exchange will not take place without either the entry or accept blocking.

4. The packages STANDARD and SYSTEM are implementation dependent. There may be multiple definitions among heterogeneous processors, which implies an interface between internal representations of data types to be passed between processors.

---

*No specific assumptions are made regarding a "distributed environment," except that there exists a communications link between processors. There may or may not be available shared memory, and the individual processors may or may not be identical. This is usually referred to as a "loosely coupled" environment.

5. Representation specifications may introduce specific assumptions of the target hardware. Their use may be necessary to resolve the implementation-dependent storage differences mentioned earlier. However, representation specifications may be inserted late in the development cycle.

6. There is no Ada construct to represent a node or processor in the network. Thus, distribution of the program cannot be handled from within the language. The use of a user-defined compiler directive (pragma) (references 2, 3, and 4) can bind sections of the program to a specific processor, but this implies a static hardware distribution and results in compiler-dependent code.

The advantages of distribution are substantial. These advantages include performance gains from parallel execution of concurrent processes, cost savings from being able to locate the processing closer to the hardware resources to be monitored or controlled, a high-degree of configuration flexibility with support of dynamic reconfiguration, and an increased fault tolerance with replication of vital system functions. The distribution of an Ada program should not restrict the use of the language, but this restriction seems to be an inherent result of the deficiencies described previously (references 5 and 6). This report examines these issues, discusses their ramifications, and attempts to resolve them.

2.2 NEED FOR PORTABLE SOFTWARE

The need for portability is important in the development of good software for distributed environments. Often the target hardware is unknown or incomplete, and the software must be developed on a different machine, or on some subset of the final configuration. Also, the target hardware may undergo change or upgrade, even after the software is fully functional, or a software change may require a hardware reallocation. For these reasons, the software should be designed, developed, and tested according to the *functions* to be performed, not according to the *hardware* on which it is to be distributed.

The software development should be a two-phase process. The first phase should occur on a single machine, using the full power of the Ada language to develop the program. In the second phase, the fully tested software is then partitioned to the target hardware components. The partitioning process should not affect the functionality of the software.

The present tendencies in the design of distributed Ada software have been to extensively modify the Ada compiler to support distribution of the application. The compiler is either modified directly (references 4, 7, and 8), or one or more compiler pragmas are added to the language (as described in

section 2.1). In other cases, the vendor's run-time system is actually tailored to support the application (references 7 and 8). This methodology results in application-specific RTEs and compiler-dependent or RTE-dependent application programs. Essentially, a new Ada dialect is being generated with every application, making the Ada programs written in these dialects nonportable and hardly reusable.

2.3 REQUIREMENTS OF THE DISTRIBUTED Ada KERNEL

Ideally, an Ada program should be able to perform the following operations:

1. Call procedures and functions declared in libraries on remote processors.

2. Rendezvous with tasks running on remote processors by entry calls (including timed or conditional entry calls).

3. Read from and write to remotely stored data objects, as well as declare and allocate local variables whose types are declared in library packages on remote processors.

4. Elaborate tasks whose types are declared in remote library packages and terminate tasks running on remote processors.

5. Propagate exceptions from the called task or subprogram back into the calling module.

It is the contention of this report that the distributed Ada message-passing kernel can meet the above requirements, using existing Ada compilers and run-time systems without modifications. Both the kernel itself and any programs constructed based on its use are portable and reusable.

The distributed Ada kernel should be compatible with the ISO standards described for the OSI reference model, yet the kernel should be able to meet the real-time requirements of many Ada applications, especially time-critical military applications. With these considerations in mind, the kernel was designed to be implemented with the high-speed fiber-optic distributed data interface (FDDI). Some modifications to the protocol stack proved to be necessary to increase throughput. In light of this, "lightweight" or "express" protocols were examined for possible incorporations into the model (references 9, 10, and 11).

The distributed Ada kernel should also utilize the strong typing of the Ada language. All messages should be verified statically. This facility was incorporated into the model by including declarations within the kernel (and any necessary representation specifications) for each data type of all parameters to be passed. This facility can be global to all applications, but generally will be application-specific by including the kernel specification within the application.

3. DESIGN AND IMPLEMENTATION OF THE KERNEL

3.1 METHODOLOGY

The methodology used to construct the distributed Ada message-passing kernel followed good software engineering practices. ISO standards exist for the OSI reference model, and the kernel was designed according to these standards. Since the ISO/OSI reference model is a layered concept (see figure 1), the kernel was written and tested in a bottom-up fashion, each layer using the services of the lower layers. Each layer was constructed using good software engineering practices: ISO standards were used for requirements definition; the requirements were decomposed in a top-down fashion into small independent modules; the design was implemented in Ada; and the model was tested using a single processor. The entire system was tested on a homogeneous set of computers linked via a single local area network (LAN).

The Ada package specification of the kernel is listed in the appendix.

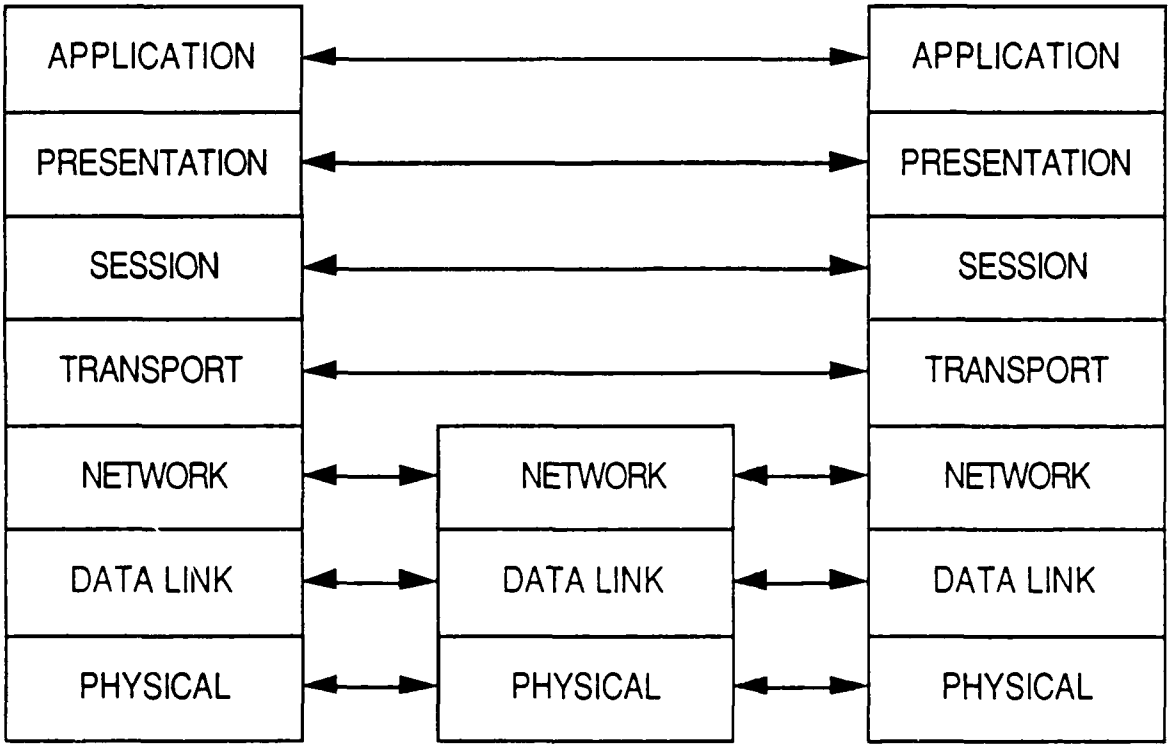


Figure 1. ISO/OSI Reference Model

3.2 ISO LAYERS

Each layer of the ISO/OSI model shown in figure 1 is composed of one or more entities that provide the services of the layer. Each entity communicates, via a layer-specific protocol, with a peer entity located in an analogous layer on another machine. Users of the layer can access the services provided by the entity through a service access point (SAP) provided by the entity.

ISO has specified the use of four primitive types in the provision of layer services. A user may request a service of the layer by issuing a *.request* primitive. The request is transmitted to the peer entity, which notifies the destination user of the request through an *.indication* primitive. The destination user can respond to the request with a *.response* primitive, which is returned to the original user in a *.confirm* primitive. These four ISO primitives are illustrated in figure 2.

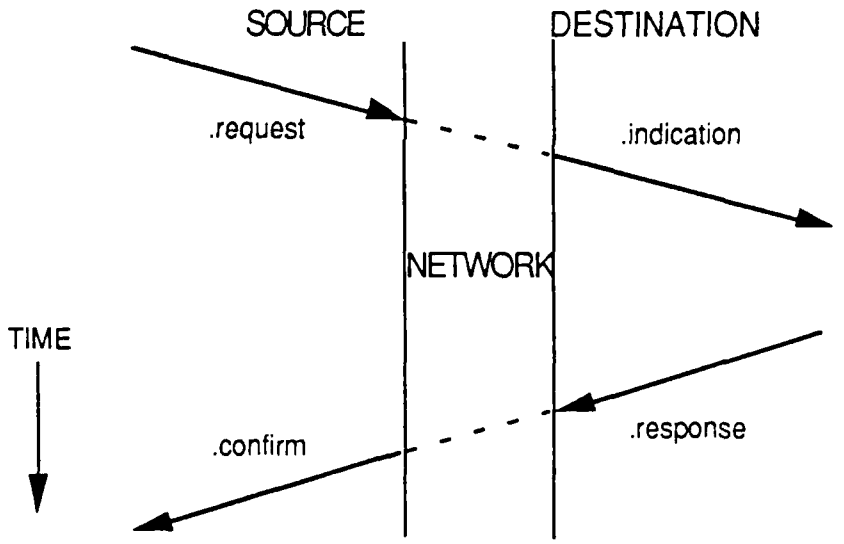


Figure 2. ISO Primitives

3.2.1 Physical Layer

The physical layer is responsible for transporting bit streams from one machine to another across the communications medium. The kernel was designed to be implemented with FDDI, which utilizes a dual counter-rotating, fiber-optic token ring with data transmission rates of 100 Mb/sec.

The FDDI standard currently encompasses both the physical layer and the medium access control (MAC) sublayer of the data link layer (reference 12).

3.2.2 Data Link Layer

The data link layer activates, maintains, and deactivates the physical link between nodes and attempts to provide virtually error-free transmission along the link. The data link layer is composed of two sublayers: the MAC sublayer and the logical link control (LLC).

One function of the MAC sublayer is to provide a checksum for error detection. Frames with correct checksums are delivered intact, while frames that are found to contain errors are ignored. The MAC sublayer is included within the FDDI specification.

The LLC was designed according to the Institute of Electrical and Electronic Engineers (IEEE) 802.2 standard (references 13, 14, and 15) for class IV service, incorporating type 1 (connectionless), type 2 (connection-mode), and type 3 (acknowledged connectionless) services. All primitives specified by the 802.2 standard were implemented in the kernel.

It was elected to buffer frames at both the sending and receiving ends of the communication link. This decision enabled the LLC of the kernel to implement the selective-repeat automatic repeat request (ARQ) data link protocol, in lieu of the go-back-N ARQ protocol specified by the 802.2 standard. Since any correct but out-of-order frames, received after a bad or missing frame, are saved (providing their sequence numbers fall within the specified window), only the bad or missing frame need be retransmitted – not each succeeding frame. This reduces not only the total number of data frames retransmitted but also the number of control frames sent.

The use of acknowledged connectionless service encounters an inefficiency with the early token release of FDDI. Since the token is released as soon as the last bit of the last frame is placed on the medium, the A and C bits are no longer useful for immediate acknowledgment of frames.* The 802.2 standard provides for the use of separate acknowledgment frames with acknowledged connectionless service. The stop-and-wait ARQ protocol is used.

---

*The A and C bits are set by the receiver of the frame, after the checksum has been evaluated. The A bit acknowledges the frame and the C bit shows the result of the checksum evaluation. When the frame returns to the sender, the sender can check the A and C bits to see if the frame arrived correctly. This is done prior to release of the token, so that retransmission may occur immediately, if necessary. However, with early token release, the sender must wait until the next token is received.

Perhaps one of the major shortfalls of the 802.2 standard lies in the area of addressing. The standard provides for eight-bit addressing. However, one bit designates a group or individual address, and one bit is reserved for 802.2 definition. Allowing for additional addresses for LLC management functions, this leaves only about 32 possible logical link service access points (LSAPs). In light of the fact that, with LLC connection-oriented service, each priority level needs to be assigned an individual LSAP to maintain sequencing, the LSAP address space can easily fall short of system requirements. The LSAP addressing scheme in the communications kernel consists of one LSAP for each connection priority level and a single LSAP for connectionless service.

3.2.3 Network Layer

The network layer provides for the transparent transfer of data across the communications facility, possibly utilizing many dissimilar networks. This layer maintains a consistent view of the entire network and is responsible for the routing of messages between the interconnecting subnetworks. Congestion control is also considered to be a major function of the network layer. The network layer is null for the purposes of this model, since a homogeneous set of computers linked by a *single* local area network is used. A network layer may be added at any time without revising the model.

3.2.4 Transport Layer

The transport layer is responsible for the reliable end-to-end transfer of data. This layer maintains connections between end systems and provides end-to-end flow control and error control. The transport layer was designed according to the ISO standards (reference 16), providing all class 3 transport services. Primitives for connectionless, connection-oriented, and acknowledged connectionless services are provided. Some aspects of the express protocols were incorporated into the model to meet the real-time requirements of the distributed Ada applications.

The address of a process using the communications kernel is considered to be its transport service access point (TSAP). Since class 3 service is implemented in the kernel, more than one TSAP may be multiplexed onto a single network connection at any one time. Above the transport layer, is a one-to-one relationship.

With the widespread use of FDDI as a communications medium, it is expected that the overall reliability of corresponding networks would be increased significantly. Error detection with Type C (unreliable) networks was not considered to be an issue at the transport layer. However, the kernel does provide for error recovery for network layer signaled failures (i.e., a reset or a

disconnect), allowing for compatibility with both Type A (reliable) and Type B (reliable delivery, but occasional signaled failures) networks.

It is assumed that the capacity of FDDI, in regard to maximum packet size, would suffice for almost all distributed Ada applications using the kernel. Therefore, the segmenting of messages into separate packets was not provided as a function of the transport layer. However, segmenting may be added to the kernel, if necessary, without severe modifications.

One feature of the lightweight protocols used in the model was the omission of any quality of service negotiation in the establishment of connections. It was assumed that distributed Ada applications would adhere to a single connection format, and quality of service negotiation was considered an unnecessary function involving considerable overhead. However, priorities are considered necessary and are associated with the connections when they are established.

3.2.5 Session Layer

The session layer provides the means to establish a reliable dialogue between processes. The session layer is generally considered to provide value-added services, so it was kept minimal to meet the real-time requirements of the distributed Ada applications. "Graceful" disconnect was implemented to ensure successful delivery of all messages sent prior to the DISCONNECT.request. The kernel also provides for "out-of-band" data transfer, bypassing the flow control mechanisms of the transport layer.

Message control types were included in the kernel to provide several session dialogue services. All messages may be acknowledged through the use of ACK (for positive acknowledgments) and NAK (for negative acknowledgments) message types. Token management was also provided with these message control types. Other services include transaction management (request/reply) and control messages, such as a server busy message and an end of data transmission mark.

3.2.6 Presentation Layer

The presentation layer normally is responsible for maintaining consistent internal representations of all data types to be passed as messages. Since implementation is on a homogeneous system, the presentation layer is null. On heterogeneous systems, the presentation layer performs transformations on Ada data types to a common representation specification. This layer also may be used to resolve implementation differences in the packages STANDARD and SYSTEM and may be added at any time without revising the model.

3.2.7 Application Layer

The application layer generally provides services for specific types of applications. The application layer in the communications kernel contains declarations of all the data types that will be passed as messages, thus utilizing the strong typing of the Ada language. All messages undergo static verification, leading to more reliable programs. Also contained in the application layer are any "stubs" or "agents" to successfully implement remote procedure call (RPC) or remote rendezvous.

The services provided by the application layer of the kernel include both connection-oriented and connectionless services. Connectionless services are implemented via send and receive procedures. A broadcast primitive is also provided. A multicast primitive could be added to the model, using the same general scheme as broadcast. Connection-oriented protocols include connect and disconnect operations, as well as send and receive primitives. To emulate the client-server model of communications, listen primitives are also provided. All procedures can be synchronous or asynchronous for user flexibility.

The application layer generally is considered to interface with the name server. The function of the name server is to provide the address of a remote process, which, in this case, is its transport layer address (i.e., its TSAP). A process must first register with the name server, so that other processes may communicate with it. This is vital to the location of server stub processes to implement the remote procedure call and remote rendezvous.

A two-part name is used to represent a process in the network to an individual application. The two-part name is composed of an application name and a process name. A process is defined to be any Ada program unit (a complete program, an Ada task, or any other Ada unit) confined to run in a single address space. The application name represents the union of one or more cooperating processes to provide a single function. More than one application may run simultaneously on one processor, and more than one process may run under an application.

3.3 REVISED LAYERS

Many time-critical applications have found that implementations of the ISO layer protocols do not sufficiently meet their real-time requirements in interprocess communication. The cost in throughput is not balanced by the increased functionality of a full implementation of the ISO/OSI model. Clearly, some streamlining is required. Following the recommendation of the French military standard (reference 17), the application, presentation, and

session layers were combined to form a user layer. This user layer establishes the data types to be passed between processes, provides any implementation-specific data transformations, and maintains all connections required by the application.

Other recommendations for revision of the ISO/OSI model include the union of the network and transport layers to form a transfer layer (references 10 and 17). Since the network layer is not implemented in this kernel, the union is irrelevant; but the transfer layer makes very good sense, especially since it would not be unreasonable to assume that time-critical systems would generally be distributed along a single local area network (LAN), or a small group of similar LAN segments – not large, complex wide area networks (WANs). This simpler topology would require only a minimal network layer, providing, perhaps, some routing capability and congestion control, and possibly using only MAC level bridges to interconnect the subnets.

3.4 IMPLEMENTATION

The kernel was implemented on a VAX 11/785 computer located in Building 1171/1 of the Naval Underwater Systems Center, Newport, R.I. The kernel was completely tested using this single computer. An implementation of FDDI was unavailable for use, so the kernel was tested with the this computer and another VAX computer with identical internal representations (a VAX 8600), connected by a DECNET (ethernet LAN). The DECNET had no MAC level primitives available to the user, so implementation of internode communication was done with VAXELN services, an operating system available on both machines.

4. IMPLEMENTING THE Ada CONSTRUCTS

4.1 REMOTE DATA OBJECTS

Perhaps the best understood application of a program to the distributed environment is the distribution of data objects; however, the access of data objects on remote machines presents a sizeable potential for problems. The following sections describe how the kernel is to be used to implement the declaration and access of data objects on remote machines.

4.1.1 Declaring Remote Data Objects

The types of all data objects that are to be passed as messages, including Ada private types and exceptions, are declared within the communications kernel. A copy of the kernel exists on every node on which the application is to be distributed. All other data types are replicated, but the original scope is maintained. When a data object is declared by a process, the object is created by the processor on which that process is running and stored in that process's memory space.

There is a deficiency in the inclusion of data types within the kernel. This is the possible loss of some of the information hiding facilities provided by the Ada package. This would occur whenever a package contains its own type definitions, and the package is allocated to one processor and used by another processor via the Ada *with* clause. Some of the type definitions of the package (those that are to be passed as messages) may need to be removed from the package and inserted into the communications kernel.

4.1.2 Reading Remote Data Values

When a process desires to obtain the value of a data object declared by a remote process, the process creates a message of the data type of the requested object. The message control field is set to REQUEST and the message is sent to the remote process. The remote process obtains the value of that object, copies it into the message record, sets the message control field to REPLY, and sends the message back to the original process.

The reading of the value of a remote data object requires the creation of agents, both at the sender and receiver. (Throughout this report, the terms "agent" and "stub" are equivalent.) The following section of code would be included in the agent requesting the value of a remote data object of a given type, ITEM. (The data type (ITEM), the information type enumerated literal (ITEM_TYPE), and the message information record field (ITEM_MESSAGE) -

all are declared in the communications kernel. The sending and receiving agents are assumed to have established a connection, identified by PORT.*)

```
function READ_ITEM return ITEM is
  MESSAGE : MESSAGE_TYPE;
begin
  MESSAGE := new INFORMATION_RECORD(ITEM_TYPE);
  MESSAGE.CONTROL := REQUEST;
  SEND(MESSAGE, PORT);
  RECEIVE(MESSAGE, PORT);
  if MESSAGE.CONTROL = REPLY then
    return MESSAGE.ITEM_MESSAGE;
  else
    <raise exception>
  end if;
end READ_ITEM;
```

The value of the remote data object would be obtained from the agent by the following:

```
declare
  X : ITEM;
begin
  X := READ_ITEM;
  <use X>
end;
```

There would be a single agent at the receiving process to handle accesses to all data objects distributed by the application that reside in that process's memory space. This agent would contain the following code:

```
task body READ_AGENT is
  MESSAGE : MESSAGE_TYPE;
begin
  loop
    RECEIVE(MESSAGE, PORT);
    case MESSAGE.CLASS is
      when ITEM_TYPE =>
         MESSAGE.ITEM_MESSAGE := <local copy>
      when ANOTHER_TYPE =>
         <other types handled similarly>
    end case;
    MESSAGE.CONTROL := REPLY;
    SEND(MESSAGE, PORT);
  end loop;
end READ_AGENT;
```

Under most circumstances, this model will suffice; however, more than one instantiation of a data type on a single machine may cause confusion during a read or write operation. In these cases, an identifier field

---

*In the examples in this section, the use of connection-oriented service is required. For a more comprehensive discussion of connection-oriented service versus connectionless service, refer to section 6.1.

should be added to the corresponding message type to discern the correct data object, and this identifier provided as a parameter to the read function.

Record assignment should be performed on a component-by-component basis, rather than using the Ada whole record assignment mechanism. The reason for this is that if a component of the record happens to be located on a remote machine, code will need to be added to access the remote component. The same rationale holds for arrays.

4.1.3 Writing to Remote Data Objects

When a process needs to update the value of a data object declared by a remote process, it creates a message of the corresponding data type. The message control field is set to DATA and the message is sent to the remote process. The remote process updates the value of the data object and then can return an ACK message to the sender, providing the write was successful. If the write was unsuccessful (i.e., some datum in the message was invalid, the variable does not exist, lack of memory, etc.), the remote process can return a NAK message.

Writing to a remote data object also requires the creation of agents, both at the sender and receiver. The following section of code would be included in the agent attempting to update the value of the object of a given data type, ITEM. (The conditions listed in section 4.1.2 are assumed for this example also.)

```
procedure WRITE_ITEM (MY_ITEM: in ITEM) is
  MESSAGE : MESSAGE_TYPE;
begin
  MESSAGE := new INFORMATION_RECORD(ITEM_TYPE);
  MESSAGE.CONTROL := DATA;
  SEND(MESSAGE, PORT);
  RECEIVE(MESSAGE, PORT);
  if MESSAGE.CONTROL = ACK then
    <message received correctly>
  else
    <raise exception>
  end if;
end WRITE_ITEM;
```

A remote data object could be updated by the process with the following:

```
declare
  X : ITEM;
begin
  <calculate new value of X>
  WRITE_ITEM(X);
end;
```

There would be a single agent at the receiving process to handle all updates to data objects distributed by the application that reside in that process's memory space. This agent would contain the following code:

```
task body WRITE_AGENT is
  MESSAGE : MESSAGE_TYPE;
begin
  loop
    RECEIVE(MESSAGE, PORT);
    case MESSAGE.CLASS is
      when ITEM_TYPE =>
        <local copy> := MESSAGE.ITEM_MESSAGE;
      when ANOTHER_TYPE =>
        <other types handled similarly>
    end case;
    if <write was successful> then
      MESSAGE.CONTROL := ACK;
    else
      MESSAGE.CONTROL := NAK;
    end if;
    SEND(MESSAGE, PORT);
  end loop;
end WRITE_AGENT;
```

The use of separate read and write agents can handle concurrent requests; however, the model doesn't preclude the possibility of failure. Consider two tasks, implemented on separate remote machines, that successively read a variable, increment it, and write the result back. Normally, if each task executes independently, the variable will be incremented twice. However, if these tasks make their requests simultaneously, then both may read the same value, each may increment their own copy of the variable once, and each may rewrite the same (but incorrect) result. Some method of mutual exclusion is needed for these kinds of operations. The following controller task can be added to the model.

```
task body ITEM_CONTROLLER is

begin
  loop
    select
      accept OK;
    or
      accept LOCK;
      accept UNLOCK;
    end select;
  end loop;
end ITEM_CONTROLLER;
```

All read agents that do not require mutual exclusion would call the entry, ITEM_CONTROLLER.OK. The implementation of mutual exclusion would include a call to ITEM_CONTROLLER.LOCK in the READ_AGENT

(before the read is executed) and ITEM_CONTROLLER.UNLOCK in the WRITE_AGENT (after the write is executed).

4.1.4 Use of Attributes with Remote Data Objects

Any of the predefined type attributes (P'FIRST, P'SUCC, P'LARGE, P'LENGTH, P'POS, A'RANGE, etc.) could be applied successfully on any machine, since all types are defined within the communications kernel. However, use of the predefined object attributes (A'CONSTRAINED, P'SIZE, etc.) would require the existence of a function, which would send a message to a remote agent, which, in turn, would evaluate the attribute and return a reply to the sender. The appropriate message type would need to be added to the kernel.

4.2 REMOTE PROCEDURE CALL (RPC)

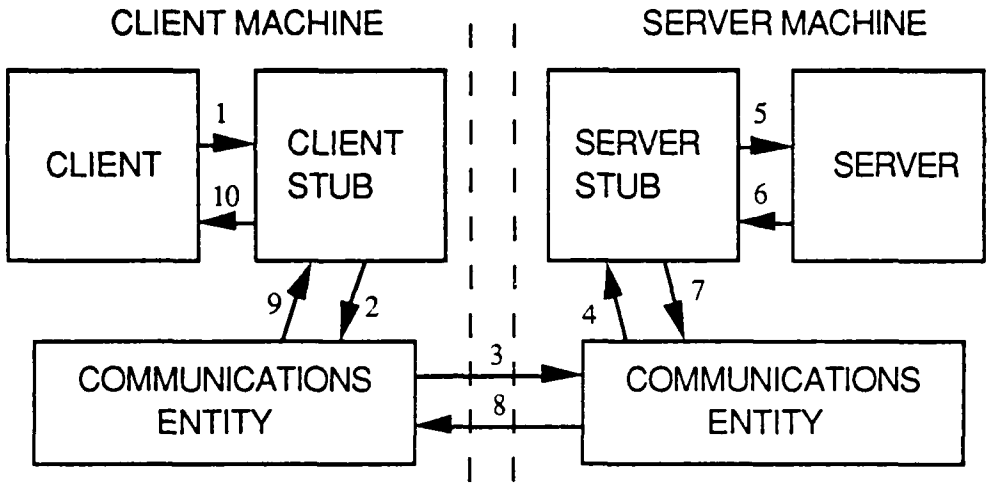
4.2.1 Implementation of RPC

Procedure* calls within the Ada language follow the client-server model. That is, the client (caller) makes a request (calls the procedure) to the server (the procedure) and the server returns a reply (updates any *out* parameters). The model is asymmetric, in that the client knows the server, but the server need not know the client until the service is actually requested.

This model has been extended to the distributed environment (reference 9). The model requires that stub processes, or agents, be created on behalf of both clients and servers. Server stubs "listen" for clients at an address known to the name server. When a client wishes to call the remote procedure, what is called is actually the client stub, which, in turn, packages the *in* parameters into a message (of type REQUEST) and sends the message to the server stub. The server stub receives the message, unpacks the parameters, and then calls the actual procedure in the normal way. If any results are returned, the server stub puts the results into a message (of type REPLY) and sends the message to the client stub, who returns the results to the client, by updating any *out* parameters. (Any *in out* parameters would appear in the parameter lists of both the REQUEST and REPLY messages.) This process is illustrated in figure 3.

---

*Procedure refers to both kinds of Ada subprograms: procedures and functions.



1. CLIENT PROGRAM CALLS STUB PROCEDURE
2. CLIENT STUB PUTS PARAMETERS IN MESSAGE
3. COMMUNICATIONS ENTITY SENDS MESSAGE
4. COMMUNICATIONS ENTITY DELIVERS MESSAGE
5. SERVER STUB CALLS SERVER PROCEDURE
6. SERVER RETURNS RESULT TO SERVER STUB
7. SERVER STUB PUTS RESULT IN RETURN MESSAGE
8. COMMUNICATIONS ENTITY SENDS MESSAGE
9. COMMUNICATIONS ENTITY DELIVERS MESSAGE
10. CLIENT STUB RETURNS RESULT TO CLIENT

Figure 3. Remote Procedure Call

The client stub would contain the following code. (The client stub is assumed to have previously established a connection (identified by PORT) to the server stub. The definitions of ITEM, ITEM_TYPE, and ITEM_MESSAGE for INPUT and OUTPUT are assumed to be contained within the kernel specification.)

```
procedure SERVICE(INPUT  : in INPUT_ITEM;
                  OUTPUT : out OUTPUT_ITEM) is
  IN_MESSAGE, OUT_MESSAGE : MESSAGE_TYPE;
begin
  IN_MESSAGE := new INFORMATION_RECORD(INPUT_ITEM_TYPE);
  IN_MESSAGE.CONTROL := REQUEST;
  IN_MESSAGE.INPUT_ITEM_MESSAGE := INPUT;
  SEND(IN_MESSAGE, PORT);
  RECEIVE(OUT_MESSAGE, PORT);
  if OUT_MESSAGE.CONTROL = REPLY then
    OUTPUT := OUT_MESSAGE.OUTPUT_ITEM_MESSAGE;
  else
    <raise exception>
  end if;
end SERVICE;
```

To obtain the service, the client process simply provides the correct parameters and calls the procedure in the stub.

```
declare
  A : INPUT_ITEM;
  B : OUTPUT_ITEM;
begin
  <obtain input values>
  SERVICE(A, B);
end;
```

The server stub would contain the following code. (The server stub is dedicated to a single client in the examples, so a connection, identified by PORT, is assumed to exist between the client stub and the server stub, and all ITEM definitions occur in the kernel.)

```
task body SERVER is
  INPUT  : INPUT_ITEM;
  OUTPUT : OUTPUT_ITEM;
  IN_MESSAGE, OUT_MESSAGE : MESSAGE_TYPE;
begin
  loop
    RECEIVE(IN_MESSAGE, PORT);
    case IN_MESSAGE.CLASS is
      when INPUT_ITEM_TYPE =>
        INPUT := IN_MESSAGE.INPUT_ITEM_MESSAGE;
        SERVICE(INPUT, OUTPUT);
        OUT_MESSAGE :=
                    new INFORMATION_RECORD(OUTPUT_ITEM_TYPE);
        OUT_MESSAGE.CONTROL := REPLY;
        OUT_MESSAGE.OUTPUT_ITEM_MESSAGE := OUTPUT;
        SEND(OUT_MESSAGE, PORT);
      when ANOTHER_TYPE =>
        <other types handled similarly>
    end case;
  end loop;
end SERVER;
```

4.2.2 Error Handling and Semantics of RPC

If RPC performed its function normally, the correct result is obtained and control is returned to the client program. However, RPC is subject to exceptions that are raised during the execution of the procedure, lost messages, errors in either the server or client stubs, and failure of either the server or client.

Exceptions raised during the execution of the actual procedure can be communicated to the client stub and then raised in the client program. Section 4.6 discusses the mechanism by which exceptions are communicated by the kernel.

If connection-oriented communication service is used,* a lost message will be automatically retransmitted, or a communications failure will be reported with the exception, CONNECTION_LOST. However, messages could be delivered correctly by the communications facility, but the procedure call is not executed. In these cases, timers could be used for retransmission of the request. The exception, COMMUNICATION_ERROR, is provided by the kernel for those cases where requests or replies are hopelessly lost (i.e., a maximum number of retries have been attempted).

An implementation of timers is a built-in feature of the distributed Ada kernel. Normally, the client stub invokes the RECEIVE procedure and waits until a reply is received. If the conditional RECEIVE procedure is invoked instead, the procedure will time out and return, if a reply is not received within the specified delay. The SEND and RECEIVE calls in the earlier example of the procedure SERVICE would be replaced by the following code:

```
declare
  WAIT_TIME : constant DURATION := <maximum delay>
  STATUS : RECEIVE_STATUS;
begin
  SEND(IN_MESSAGE, PORT);
  RECEIVE(OUT_MESSAGE, PORT, WAIT_TIME, STATUS);
end;
```

This implementation may be expanded further to include a number of retries. If, after the number of retries, a reply has not been received, a failure in the communications link or the server processor should be assumed, and the agent should raise the exception, COMMUNICATION_ERROR. The revised code is as follows:

```
declare
  WAIT_TIME   : constant DURATION := <maximum delay>
  STATUS      : RECEIVE_STATUS;
  MAX_RETRIES : constant INTEGER := <number of retries>
  RETRY_COUNT : INTEGER := 1;
begin
  loop
    SEND(IN_MESSAGE, PORT);
    RECEIVE(OUT_MESSAGE, PORT, WAIT_TIME, STATUS);
    exit when STATUS = SUCCESS
         and OUT_MESSAGE.CONTROL = REPLY;
    if RETRY_COUNT = MAX_RETRIES then
      raise COMMUNICATION_ERROR;
    else
      RETRY_COUNT := RETRY_COUNT + 1;
    end if;
  end loop;
end;
```

*If connectionless service is used, an ACK message should be used to acknowledge receipt of the REQUEST message.

This type of model should be used with connectionless services. Since delivery of messages is not guaranteed, communication can then be reattempted, until a reply is received or until the client wishes to give up.

In the face of server crashes or server stub errors, the client may have no way of knowing whether or not the procedure was executed successfully. "At least once" semantics (the request is retransmitted until a proper reply is received) will always apply to idempotent operations. Idempotent operations are operations that may be repeated without causing any ill effects (i.e., a read operation). Many real-time data sources (i.e., sensor results, etc.) are idempotent, and requests may be repeated until a successful response is obtained. On the other hand, if the operation was nonidempotent (i.e., a bank withdrawal), then retransmissions of the request may cause reexecutions of the procedure that actually may be harmful.

Ideally, RPC would like to have "exactly once" semantics, where the procedure is always executed once and only once. This is impossible to achieve. Since Ada supports exception handling, an appropriate handler could be invoked and the error communicated to the client stub, which can invoke its own exception handler or raise an exception in the client. The use of transaction IDs, generated for each request by the client stub, could be added to the model, which would be stored by the server stub, together with all appropriate state information and any returned results of the procedure call. Retransmissions by the client stub would bear the same transaction ID, so the server stub could determine if the procedure had been executed and make the correct response (i.e., reexecute the procedure, or return the result of the previous execution).

If the client crashes or there is an error in the client stub before the result is returned, then the server process, in some cases, may continue to run without a client. (This is an example of an "orphan"; see section 6.7.) If the server has claimed computer resources, these resources need to be released.

One difficulty with the RPC model is the fact that procedures are reentrant. Simultaneous calls to the procedure are expected to run concurrently; however, if a server stub accepts RPC requests one at a time, then this will not be possible. In fact, this is never strictly achievable, since the communications link only delivers messages one at a time. The only possible solution is replication of the server stub, either dynamically on receipt of a request, or by maintaining a separate connection to every client stub. The latter is the approach taken in these examples. The dynamic server stub creation approach is illustrated in figure 4.

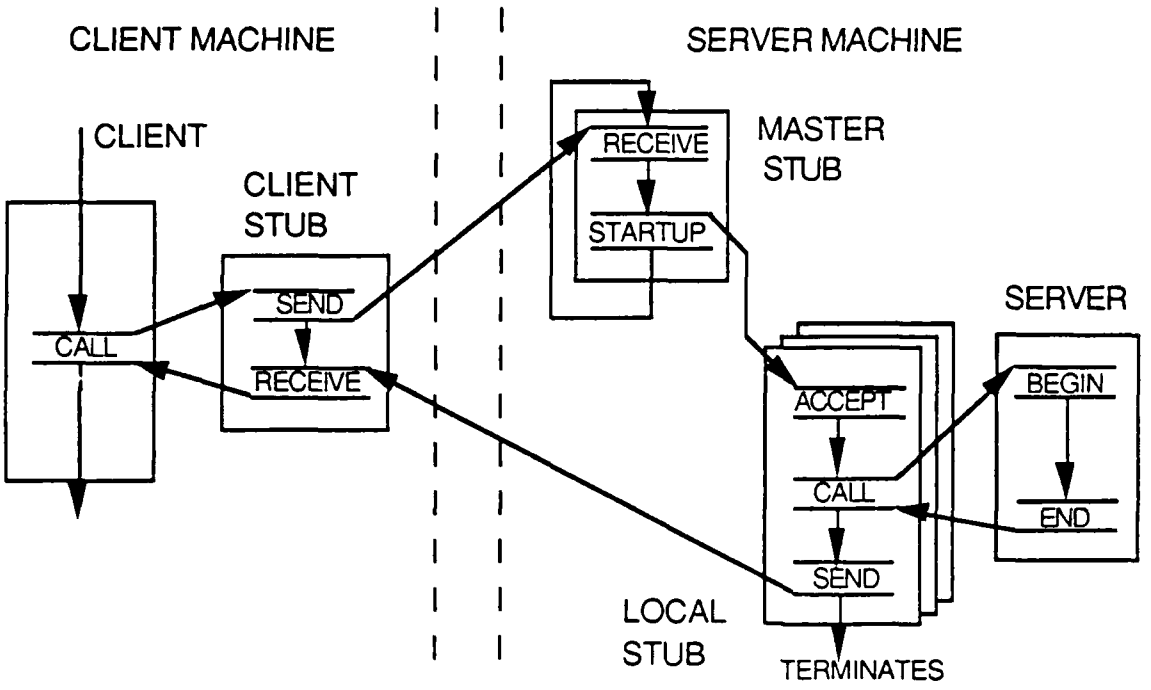


Figure 4. Dynamic Server Stub Creation

Dynamic creation of server stubs, via the model illustrated in figure 4, would be the method of choice with the use of connectionless communications services. There would exist one master server stub per server to receive all requests of that server (from any client stub). Upon receipt of a request, the master stub instantiates a local server stub and passes the address of the client stub that made the request, along with any input parameters. The master stub, after creating the local stub, is free to accept another incoming request. The local stub then proceeds to call the server and any results are returned directly to the client stub. After returning the results to the client stub, the local server stub terminates. The following code is an example of a master server stub, with dynamic server stub creation, using connectionless service.

```
task body MASTER_SERVER is

task type SERVER_TYPE is
  entry STARTUP(APPLICATION : APPLICATION_UNIT_NAME;
                PROCESS     : PROCESS_NAME
                MESSAGE     : MESSAGE_TYPE);
end SERVER_TYPE;

task body SERVER_TYPE is
  CALLER_APPLICATION : APPLICATION_UNIT_NAME;
  CALLER_PROCESS     : PROCESS_NAME;
  IN_MESSAGE, OUT_MESSAGE : MESSAGE_TYPE;
```

```
  PRIORITY     : PRIORITY_CLASS := <priority of return>;
begin
  accept STARTUP(APPLICATION : APPLICATION_UNIT_NAME;
                 PROCESS     : PROCESS_NAME
                 MESSAGE     : MESSAGE_TYPE) do
    RETURN_APPLICATION := APPLICATION;
    RETURN_PROCESS := PROCESS;
    IN_MESSAGE := MESSAGE;
  end STARTUP;
  case IN_MESSAGE.CLASS is
    when INPUT_ITEM_TYPE =>
      INPUT := IN_MESSAGE.INPUT_ITEM_MESSAGE;
      SERVICE(INPUT, OUTPUT);
      OUT_MESSAGE :=
                   new INFORMATION_RECORD(OUTPUT_ITEM_TYPE);
      OUT_MESSAGE.CONTROL = REPLY;
      OUT_MESSAGE.OUTPUT_ITEM_MESSAGE := OUTPUT;
      SEND(OUT_MESSAGE, SERVER_APPLICATION, SERVER_PROCESS,
           RETURN_APPLICATION, RETURN_PROCESS, PRIORITY);
    when ANOTHER_TYPE =>
      <other types handled similarly>
  end case;
end SERVER_TYPE;

type SERVER is access SERVER_TYPE;

SERVER_TASK : SERVER;
APPLICATION : APPLICATION_UNIT_NAME;
PROCESS     : PROCESS_NAME;

begin
  loop
    RECEIVE(MESSAGE, APPLICATION, PROCESS, PORT);
    SERVER_TASK := new SERVER_TYPE;
    SERVER_TASK.STARTUP(APPLICATION, PROCESS, MESSAGE);
  end loop;
end MASTER_SERVER;
```

Recursion is another aspect of procedure call that requires handling. Recursion in a server should occur normally; however, if a remote procedure calls the procedure that initiated the RPC, and the original procedure recursively issues another RPC, then deadlock is possible. To illustrate, assume that a procedure A remotely calls a procedure B, which then calls A again. If A then tries to call B a second time, B's server stub can never accept the call, since B's server stub will be blocked awaiting completion of the original call to procedure B, which is also blocked awaiting completion of the call to A. The difficulty with this type of recursion is resolved by the use of dynamic creation of server stubs.

Ada allows the redefinition of operators as a form of subprogram. Since redefinition of operators would usually occur within a package associated with the presentation of an abstract data type, these operators may be replicated along with the type definitions. An alternative is to rename the operators as procedures, but this distorts the syntax of the operations.

4.3 REMOTE RENDEZVOUS

4.3.1 Implementation of Remote Rendezvous

As with the procedure call, the Ada rendezvous exhibits the asymmetry of the client-server model. The RPC mechanism also works very well to implement the remote rendezvous. This similarity is expected, since the Ada language permits task entries to be renamed as procedures.

The client stub would contain the following code. (As in the RPC example, the client stub is assumed to have previously established a connection (identified by PORT) to the server stub. The definitions of ITEM, ITEM_TYPE, and ITEM_MESSAGE for INPUT and OUTPUT are assumed to be contained within the kernel specification.)

```
task body SERVICE_TASK is
  IN_MESSAGE, OUT_MESSAGE : MESSAGE_TYPE;
begin
  loop
    select
      accept SERVICE(INPUT  : in INPUT_ITEM;
                     OUTPUT : out OUTPUT_ITEM) do
        IN_MESSAGE :=
                     new INFORMATION_RECORD(INPUT_ITEM_TYPE);
        IN_MESSAGE.CONTROL := REQUEST;
        IN_MESSAGE.INPUT_ITEM_MESSAGE := INPUT;
        SEND(IN_MESSAGE, PORT);
        RECEIVE(OUT_MESSAGE, PORT);
        if OUT_MESSAGE.CONTROL = REPLY then
          OUTPUT := OUT_MESSAGE.OUTPUT_ITEM_MESSAGE;
        else
          <raise exception>
        end if;
      end SERVICE;
    or
      <other task entries are similar>
    or
      terminate;
    end select;
  end loop;
end SERVICE_TASK;
```

To obtain the service, the client process simply provides the correct parameters and calls the entry in the stub.

```
declare
  A : INPUT_ITEM;
  B : OUTPUT_ITEM;
begin
  <obtain input values>
  SERVICE_TASK.SERVICE(A, B);
end;
```

The server stub would contain the following code. (Again, a connection, identified by PORT, is assumed to exist between the client stub and the server stub, and all ITEM definitions occur in the kernel.)

```
task body SERVER is
  INPUT  : INPUT_ITEM;
  OUTPUT : OUTPUT_ITEM;
  IN_MESSAGE, OUT_MESSAGE : MESSAGE_TYPE;
begin
  loop
    RECEIVE(IN_MESSAGE, PORT);
    case IN_MESSAGE.CLASS is
      when INPUT_ITEM_TYPE =>
        INPUT := IN_MESSAGE.INPUT_ITEM_MESSAGE;
        SERVICE_TASK.SERVICE(INPUT, OUTPUT);
        OUT_MESSAGE :=
                  new INFORMATION_RECORD(OUTPUT_ITEM_TYPE);
        OUT_MESSAGE.CONTROL := REPLY;
        OUT_MESSAGE.OUTPUT_ITEM_MESSAGE := OUTPUT;
        SEND(OUT_MESSAGE, PORT);
      when ANOTHER_TYPE =>
        <other types handled similarly>
    end case;
  end loop;
end SERVER;
```

Families of entries would be reproduced in the client stubs. The server stubs would either be replicated, one stub for each entry of the family, or a single server stub would be used, passing the index of the individual entry of the family along with the parameters.

The Ada *select* statement is implemented normally. Since the selection of the entry to accept is of interest only to the server, then the selection process may be restricted to the server, even if all the entry calls are by the remote rendezvous mechanism.

4.3.2 Conditional and Timed Entry Calls

Ada provides a mechanism whereby a process may opt out of a rendezvous, if the rendezvous cannot be performed immediately or within a specified length of time. This mechanism fails when a client attempts a remote rendezvous. The client may be able to rendezvous with a client stub; however, when the request arrives at the server stub, the server may be engaged in a rendezvous with some process other than the client and would not be able to complete the rendezvous with the server stub (on behalf of the client). In this case, the original rendezvous with the client and client stub should not have taken place, and the client should have executed the alternate section of code.

The remote rendezvous mechanism needs to be modified, somewhat, to accommodate the communications delays. Two parameters are added to the client stub entry: a STATUS parameter, which is a Boolean *out* parameter that informs the client whether or not the remote rendezvous was performed; and a WITHIN parameter, which is an *in* parameter of the Ada predefined type DURATION that specifies the delay after which the rendezvous is abandoned (a delay of 0.0 specifies a conditional entry call). A field in the input message type of the communications kernel must be reserved to transmit this delay to the server stub. The revised client stub would be as follows:

```
task body SERVICE_TASK is
  IN_MESSAGE, OUT_MESSAGE : MESSAGE_TYPE;
begin
  loop
    select
      accept SERVICE(INPUT  : in INPUT_ITEM;
                     WITHIN : in DURATION;
                     OUTPUT : out OUTPUT_ITEM;
                     STATUS : out BOOLEAN) do
        IN_MESSAGE :=
                    new INFORMATION_RECORD(INPUT_ITEM_TYPE);
        IN_MESSAGE.CONTROL := REQUEST;
        IN_MESSAGE.INPUT_ITEM_MESSAGE := INPUT;
        IN_MESSAGE.WITHIN := WITHIN;
        SEND(IN_MESSAGE, PORT);
        RECEIVE(OUT_MESSAGE, PORT);
        if OUT_MESSAGE.CONTROL = REPLY then
          OUTPUT := OUT_MESSAGE.OUTPUT_ITEM_MESSAGE;
          STATUS := TRUE;
        elsif OUT_MESSAGE.CONTROL = NAK then
          STATUS := FALSE;
        else
          <raise exception>
        end if;
      end SERVICE;
    or
      <other task entries are similar>
    or
      terminate;
    end select;
  end loop;
end SERVICE_TASK;
```

The conditional or timed entry call in the client would be replaced by the following code:

```
declare
  A : INPUT_ITEM;
  B : OUTPUT_ITEM;
  STATUS : BOOLEAN;
  WITHIN : DURATION;
begin
  <obtain input values>
```

```
    WITHIN := <0.0 or delay in timed entry call>
    SERVICE_TASK.SERVICE(A, WITHIN, B, STATUS);
    if not STATUS then
      <alternate section of code>
    end if;
  end;
```

The server stub would execute the conditional or timed entry call as a timed entry call using the delay transmitted by the client stub. If the delay is zero, then the timed entry call behaves as a conditional entry call. The server stub would contain the following code:

```
when INPUT_ITEM_TYPE =>
  INPUT := IN_MESSAGE.INPUT_ITEM_MESSAGE;
  OUT_MESSAGE := new INFORMATION_RECORD(OUTPUT_ITEM_TYPE);
  select
    SERVICE_TASK.SERVICE(INPUT, OUTPUT);
    OUT_MESSAGE.CONTROL := REPLY;
    OUT_MESSAGE.OUTPUT_ITEM_MESSAGE := OUTPUT;
  or
    delay IN_MESSAGE.WITHIN;
    OUT_MESSAGE.CONTROL := NAK;
  end select;
  SEND(OUT_MESSAGE, PORT);
```

Assuming that a timed entry call with zero or negative delay is a conditional entry call, then the question of which machine, the client's or the server's, is to evaluate the delay to determine whether to abandon the rendezvous becomes critical. The semantics of the remote conditional entry call are fairly straightforward. The rendezvous should be completed, only if the remote server task is able to accept the rendezvous immediately. Otherwise, the existence of a communications delay means that all conditional entry calls should fail (references 18 and 19).

With the remote timed entry call, the semantics are not so straightforward. The client machine could easily time out if a reply is not received within the specified delay (using the conditional RECEIVE); however, this is not sufficient in the face of communications delays or lost replies. There is a need to make the distinction between communications failure and the expiration of a delay. The server task could complete the rendezvous, and yet, the client stub could still time out and abandon the rendezvous. The client cannot know for sure if the rendezvous was performed. Timed entry calls were not designed for network failure; rather, exceptions should be used instead.

The timed entry call is the only Ada construct that places an upper bound on the time to make the call. This would seem to imply that the client should control the delay. If the client machine maintains the sense of time for the delay, then the client machine would need to know exactly when the server task becomes ready to accept the rendezvous. This would require a

message from the server, indicating the server's ready state. The client stub would then respond with a message, indicating that it is OK to proceed with the rendezvous, or that the rendezvous should be cancelled. This four-packet protocol is illustrated in figure 5.

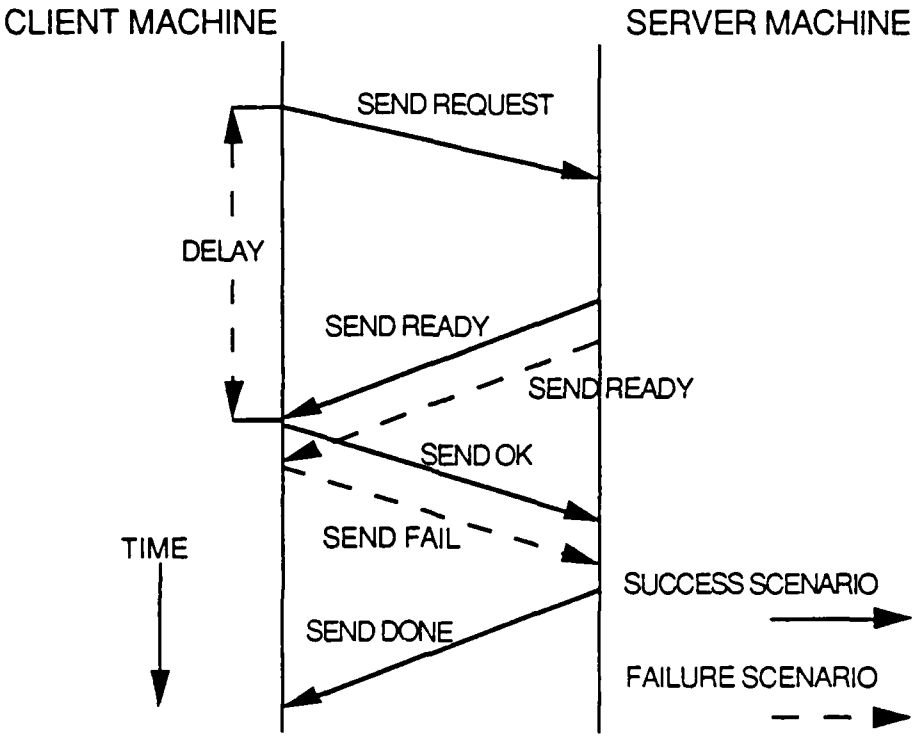


Figure 5. Timed Entry Call – Client Controlled

The timed entry call illustrated in figure 5 shows an anomaly for small delays. Since timed entry calls with a zero or negative delay are considered conditional entry calls, then these calls should succeed, providing the server is ready to accept the rendezvous immediately. However, for any delay that is less than the round trip network transmission time, the call will fail, since the client will receive notification that the server is ready, only after the delay has expired. This model clearly is not sufficient.

The approach taken in this report is that the server machine maintains the sense of time for the delay. This approach is illustrated in figure 6. Not only is this two-packet protocol conceptually simpler and easier to implement, but the actual semantics of the Ada timed entry call are retained. It is up to the client to allow for possible communications delays in the calculation of the time-out values. The disadvantage of the server maintaining the sense of

time is that the client cannot place an upper bound on the time of the client's call. However, the Ada language specifies that the wait time is an upper bound on the time to *accept* the rendezvous, not *complete* it, so the server sense of time is consistent with the Ada language definition. The user should understand that communications delays conceptually become a part of the execution time of the remote task.

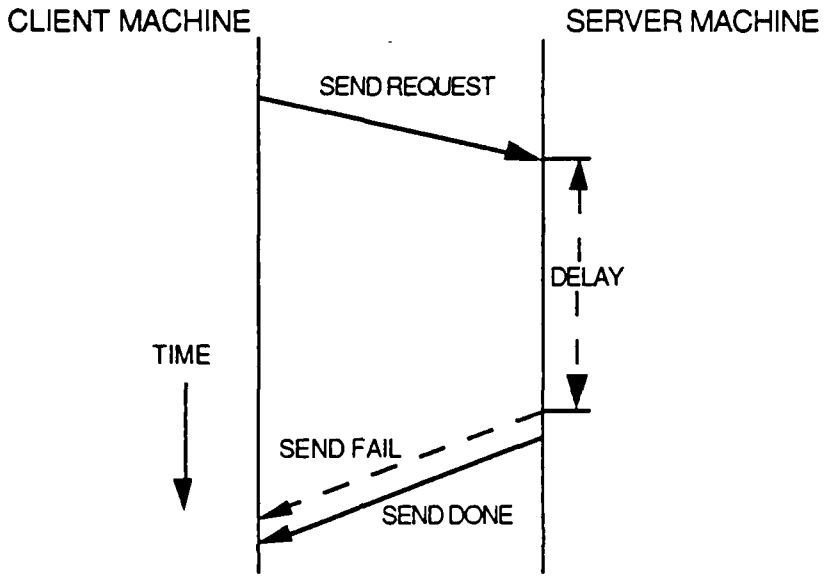


Figure 6. Timed Entry Call – Server Controlled

It would seem that the major problem of the timed entry call is the lack of a consistent sense of time across the network, leading one to believe that the solution is to utilize a network time server. However, this does not really solve the anomaly mentioned earlier. In addition, the delays associated with accessing the network time server would prove to be excessive.

4.4 BEHAVIOR OF REMOTE TASKS

4.4.1 Elaboration of Remote Tasks

As with variable types, task types are declared within the communications kernel, and replicated among the processors running the distributed application. A process can elaborate a task of this type, which is run on the same processor. All dynamically created child tasks are expected to share the same address space as the parent task. This approach maintains the scope of

the original source program, so the run-time environment can properly terminate tasks, according to present algorithms.

A dynamically created task will initially register with the name server. Through registration, other tasks can learn of the completion of the elaboration process by receipt of its address from the name server. If it is desired to control the initiation of the execution of a remote task, then a START entry may be provided to implement the necessary synchronization.

4.4.2 Termination of Remote Tasks

Termination of a task running on a remote processor is accomplished by the sending of an ABORT_TASK message. The receiving process then terminates the task either by calling a shutdown entry or abnormally terminating it by using the Ada *abort* statement. The following example shows the graceful termination of a server and server stub, through the use of a shutdown entry in the server.

```
task body SERVER is
  MESSAGE : MESSAGE_TYPE;
begin
  loop
    RECEIVE(MESSAGE, PORT);
    if MESSAGE.CONTROL = ABORT_TASK then
      SERVICE_TASK.SHUTDOWN;
      exit;
    end if;
  end loop;
end SERVER;
```

The removal of an aborted client task from the entry queue of a remote server is not supported by the model. Normally, the run-time system would perform this function, but, since it is the server stub that is actually awaiting the rendezvous, the rendezvous will eventually be performed, despite the lack of a client. It is possible, though, that ABORT_TASK messages could be used to abort the server stub, providing that the server stub was dynamically created.

4.4.3 Use of Task Attributes with Remote Tasks

Any of the predefined task attributes (T'TERMINATED, T'CALLABLE, E'COUNT, etc.) could be applied successfully to the local stubs. However, due to communications delays or abnormal termination of the remote task, the state of these stubs may not be consistent with the state of the actual remote

task.* The task attributes could be replaced by calls to an appropriate function, which would send a message to the agent for the remote task, which, in turn, would identify the actual state of the task, and return the state to the sender via a message. The appropriate message type would need to be added to the communications kernel.

4.5 GENERIC INSTANTIATIONS

Generic units are passive and are handled by the kernel in the same way as task types – all generic units are replicated on all of the distributed processors. When a process instantiates a generic unit, the instantiated unit belongs to the process containing the instantiation and runs on the same processor. Since all data types are replicated, they are visible to the generic instantiation. Generic subprogram parameters are also available; however, the actual subprogram parameter may be the agent code, which would execute a remote procedure call to access the actual subprogram.

4.6 EXCEPTION PROPAGATION

Every message that is sent contains a field denoting the exception condition. Normally, it is set to NO_EXCEPTION. Upon generation of an exception by a remote process, the remote processor returns a message to the original process with the exception field set to corresponding exception. The original process is responsible for the propagation of this exception into the application.

All server stubs or server agents must be responsible for communication of the exception condition to the client. This is supported by exception handlers in the agent as follows:

```
exception
  when NUMERIC_ERROR =>
    MESSAGE.ERROR := NUMERIC;
  when CONSTRAINT_ERROR =>
    MESSAGE.ERROR := CONSTRAINT;
  when PROGRAM_ERROR =>
    MESSAGE.ERROR := PROGRAM;
  when STORAGE_ERROR =>
    MESSAGE.ERROR := STORAGE;
```

*It is interesting to note that even on a uniprocessor the value returned by these attributes may not correctly give the current exact state of the task (which may have changed since the attribute call was invoked). However, distribution greatly magnifies this problem.

```
    when TASKING_ERROR =>
      MESSAGE.ERROR := TASKING;
    when others =>
      MESSAGE.ERROR := USER_DEFINED;
  end;
```

When this return message arrives at the client stub or client agent, the exception is propagated to the client with the following code:

```
case MESSAGE.ERROR is
  when NUMERIC =>
    raise NUMERIC_ERROR;
  when CONSTRAINT =>
    raise CONSTRAINT_ERROR;
  when PROGRAM =>
    raise PROGRAM_ERROR;
  when STORAGE =>
    raise STORAGE_ERROR;
  when TASKING =>
    raise TASKING_ERROR;
  when COMMUNICATION =>
    raise COMMUNICATION_ERROR;
  when USER_DEFINED =>
    raise;
end case;
```

4.7 ACCESS TYPES

The use of access types in Ada represents a serious stumbling block in the distribution of Ada programs. In the absence of shared memory, an address on one machine would have no significance to another machine. Access types can be embedded within more complex data structures. Furthermore, access types may point to task objects.

Most of the work to date with distributed Ada has forbidden the use of Ada access types across machine boundaries, seriously reducing the scope of the Ada language. Other work has modified the run-time system to resolve remote machine accesses (references 7, 8, and 20).

A possible solution to the use of remote accesses is to define a remote address as a combination of a physical node identifier and the physical address within that node (reference 19). The definition of SYSTEM.ADDRESS is stated to be implementation dependent, so nothing forbids redefining SYSTEM.ADDRESS as a record (although redefinition of the package SYSTEM becomes an implementation dependency). A definition of a remote address, without any implementation dependency, is given in the following code. (ACCESS_CLASS_TYPE is an enumeration type that is defined by the kernel, consisting of literals for all access types that are passed across machine boundaries.)

```
type ITEM is access <some user object type>;
type REMOTE_ADDRESS(CLASS : ACCESS_CLASS_TYPE) is record
  MACHINE : NODE_ID;
  case CLASS is
    when ITEM =>
      LOCATION : ITEM;
    when others =>
      <other access types handled similarly>
  end case;
end record;
```

All remote access types would be declared as follows:

```
declare
  MY_ITEM_POINTER : REMOTE_ADDRESS(MY_ITEM_ACCESS_TYPE);
begin
  MY_ITEM_POINTER := (NODE, new ITEM'(<user object>));
  <use MY_ITEM_POINTER>
end;
```

To access a remote object, two agents would be created. Both agents are assumed to have established a connection (identified by PORT), and REMOTE_ADDRESS_TYPE and REMOTE_ADDRESS_MESSAGE are assumed to be defined within the kernel. The client agent would be as follows:

```
type ITEM is <some user object type>;
function GET_REMOTE(WHERE : REMOTE_ADDRESS) return ITEM is
  MESSAGE, RETURN_MESSAGE : MESSAGE_TYPE;
begin
  MESSAGE := new INFORMATION_RECORD(REMOTE_ADDRESS_TYPE);
  MESSAGE.CONTROL := REQUEST;
  MESSAGE.REMOTE_ADDRESS_MESSAGE := WHERE;
  SEND(MESSAGE, PORT);
  RECEIVE(RETURN_MESSAGE, PORT);
  if RETURN_MESSAGE.CONTROL = REPLY then
    return RETURN_MESSAGE.ITEM_MESSAGE;
  else
    <raise exception>
  end if;
end GET_REMOTE;
```

The server agent is as follows:

```
task body REMOTE_AGENT is
  MESSAGE, RETURN_MESSAGE : MESSAGE_TYPE;
begin
  loop
    RECEIVE(MESSAGE, PORT);
    if MESSAGE.CLASS = REMOTE_ADDRESS_TYPE then
      case MESSAGE.REMOTE_ADDRESS_MESSAGE.CLASS is
        when ITEM_TYPE =>
          RETURN_MESSAGE :=
                          new INFORMATION_RECORD(ITEM_TYPE);
```

```
            RETURN_MESSAGE.ITEM_MESSAGE :=
                                MESSAGE.ITEM_MESSAGE.LOCATION.all;
          when ANOTHER_TYPE =>
            <other types handled similarly>
        end case;
      end if;
      MESSAGE.CONTROL := REPLY;
      SEND(MESSAGE, PORT);
    end loop;
  end REMOTE_AGENT;
```

The Ada language makes task objects appear very similar to data objects (constants), so it is not surprising that Ada permits the use of access types to task objects. Access types to task objects present a very effective mechanism in the implementation of capability addressing, where a task manager controls access to a given server task. If a client wishes to call the server, the client must first get the address of the server from the task manager, which would check the validity of the client to access the server before dispensing the capability (the address of the task) to the client. In this way, servers may remain anonymous, and clients cannot access the server without first acquiring the capability.

The dynamic creation and termination of task objects through access types present several difficulties. Since the declarations of task types are replicated, use of the allocator to elaborate a task object would occur in the usual way, but the object is to exist only on the machine that executed the *new* operation. However, task objects cannot migrate from one processor to another, so references to task objects using the *.all* construct should be restricted.

Entries to task objects via access types cannot be called in the usual way (remote rendezvous). Since the name of the task is actually a remote address, the run-time environment cannot resolve the entry call. If access types to task objects are passed as parameters with RPC or remote rendezvous, the server cannot always be expected to know the name of the actual task object in advance, so a remote rendezvous with that object may be impossible. However, providing that server and client stubs are generated for each task object created via the allocator (or, perhaps, general server and client stubs for the parent task type, passing the actual task object's remote address as a parameter with the entry call), the model may be extended to include these stub task names with the remote address. It is clear that great care is necessary with the use of access types in general, but especially access types to task objects.

5. PARTITIONING THE PROGRAM

5.1 UNITS OF DISTRIBUTION

The first problem encountered in the partitioning of an Ada program across several machines is which Ada constructs are allowed to be distributed. These range from distributing any viable Ada construct to distributing only complete Ada programs. The following sections describe a variety of possible units of distribution, including the separate communicating programs approach, partitioning on task boundaries, partitioning on package boundaries, the concept known as the virtual node approach, and partitioning on any Ada construct.

5.1.1 Separate Communicating Programs

The most straightforward (and presently the most common) approach has been to prepartition the application functionality among the processors and then design a separate program per processor. The programs use the standard I/O features provided with the language. The advantage to this approach is that it is based on available technology and requires no special supporting tools or communications system. Moreover, implementations of this approach are usually efficient, in that they take into account resource and performance constraints, and the typical *ad hoc* interfaces can utilize all the primitive facilities provided by the network to implement an optimal communications mechanism.

The separate programs approach has two important disadvantages. The first disadvantage is that the advantages of the strong typing of Ada are lost, since interprocessor messages are no longer under the supervision of the Ada compiler. The second disadvantage is that partitioning decisions are made early in the software development cycle. Since the software is designed in a "hardware first" fashion, portability and flexibility are lost. Modifications are expensive, since the design is intimately related to the structure of the underlying software. A change in the functionality of the software or a change in the hardware may require a complete redesign of the component programs.

5.1.2 Tasks

Since the Ada tasking mechanism naturally implements concurrently executing modules, it would seem that there are many benefits to assigning individual tasks to separate processors. In fact, there is a good synergy between the Ada task and a processor (references 4 and 21).

Partitioning the program on task boundaries also has several important disadvantages. First, the Ada task is not a library unit and must be enclosed inside a package. Secondly, it is difficult to ensure that tasks do not share data. Thirdly, concurrency is not the only criteria for partitioning, so partitioning on task boundaries may be unnatural. Many programs cannot be neatly and evenly partitioned to tasks; some processors may be assigned heavy workloads, while others will hardly be used at all. The software designer must possess a comprehensive view of the hardware topology, and prepartition the software to tasks, prior to design. Finally, it becomes difficult to take advantage of task types and dynamic task creation in a straightforward way.

5.1.3 Packages

Packages are the primary units of functional encapsulation of the Ada language; they are library units and can be naturally partitioned among processors. Packages provide the nice data abstraction and information hiding features of Ada and support the design of modular portable software. Packages also have a visible part, making control of the remote interface manageable. However, packages are static and do not possess their own "thread of control", so they need to be encapsulated by a task. Also, packages are not required to exhibit concurrency, so, in some cases, the advantages of distribution may be minimal. However, packages do represent a viable unit of distribution for Ada (references 3, 22, 23, and 24).

One disadvantage to the use of packages as the units of distribution is that packages are not dynamically instantiatable. This makes reconfigurability more difficult. Generic packages are of limited use, since generic packages are instantiated at compile time and their names are not exportable. Ada gives the declaration of generic packages a dynamic allocation flavor with the use of the *new* keyword, but this is not really the case. The visible package components can only be accessed locally, since the package possesses a static, local name. The only solution to this is to embed the instantiation of the package within a task type; however, this defeats the intended purpose of the package – to provide a useful data abstraction mechanism – by supplying an additional arbitrary layer of abstraction over the package.

5.1.4 Virtual Nodes

Several projects have supported the use of virtual nodes (references 2, 18, 25, 26, and 27). In this approach, an Ada program is split into indivisible

units of abstraction called virtual nodes.* A virtual node will implement some particular function of the application. This is similar to the Ada package concept, except that communication between virtual nodes is more restrictive (usually limited to RPC or remote rendezvous). Within a virtual node, Ada is used with complete freedom. A virtual node is assigned to a single processor and can never span the interprocessor link. However, virtual nodes may be distributed freely among processors, and more than one may coexist on a single processor. Virtual nodes are designed such that they can be remapped, if necessary, to support reconfigurability, portability, and fault tolerance.

The virtual node concept represents an effective compromise approach, attempting to minimize the partitioning overhead, while maximizing the hardware utilization. The major disadvantage is that the virtual node is not defined within the Ada language; therefore, a certain amount of prepartitioning is required to delineate a virtual node. If the rationale for a particular virtual node definition is subject to change, system redesign may be necessary. However, some prepartitioning may be advisable, to make more efficient use of the hardware topology and the communications facility. In fact, many programs lend themselves quite easily to this degree of prepartitioning, applying knowledge of the general distributed nature of the hardware, without committing the program to a specific hardware allocation.

5.1.5 Full Ada

Most of the previously mentioned units of distribution involve some restrictions in the way Ada is used across multiple processors. The application of these restrictions is some basic distributable atomic unit, which has been termed a "strong module" (reference 23). The most commonly observed limitation is the use of shared variables, especially access types. Since many projects involve heterogeneous systems without the use of shared memory, these projects will generally prohibit the distribution of data objects. The alternative is data replication; however, provisions must be made to maintain consistent copies of the replicated data.

The obvious goal of the distribution of Ada would be to provide unrestricted use of the Ada language in the design of the program and to provide efficient hardware allocations, completely transparent to the programmer. Software designed in this way would be highly portable, but, at present, this goal can be accomplished only with a full-scale clever compiler and considerable modifications to the run-time environment thus, eliminating the portability advantages.

---

*Virtual nodes have also been referred to as "bubbles" (reference 28), or as Ada virtual machines (reference 22).

The use of a clever compiler to produce multiple object modules, which are then distributed to the individual processors for execution, is referred to as *target code allocation*. The view of the program as a single unit is maintained throughout the program's life cycle. Target code allocation requires considerable run-time support, not present in current run-time environments, for remote operations. Also, the distributed version of the program is not available, making debugging difficult.

The best example of the clever compiler approach is the Honeywell Distributed Ada Project (references 7, 8, 20, 29, 30, and 31). Any Ada entity may be distributed, including packages, subprograms, tasks, generic instantiations, data objects, access objects, renamed objects, deferred constants, derived subprograms, and even array slices. Remote procedure call, remote read, and remote write mechanisms are provided by the run-time system to successfully implement remote access to any of these entities.

The specification of the partitioning of a program is provided in a separate language called the Ada program partitioning language (APPL). The compiler was modified to recognize both the Ada compilation unit and its APPL partitioning specification. This partitioning is independent of the functional design, and occurs after the design of the program is complete (see section 5.2.1 for a discussion on prepartitioning versus postpartitioning). APPL provides for the specification of program *fragments*, which are mapped to individual *stations* or processors. Fragments may be replicated among stations for fault tolerance. The intermediate representation of the program code contains a fragment attribute, and whenever a remote access is required, the compiler inserts a call to the run-time system.

Even with compiler and RTE modifications, extremely high overheads may be incurred to support all language features. Since these overheads are hidden from the programmer, they are not controlled easily. For example, consider a process that accesses a record that is allocated to another processor. Embedded within this record are access types to objects on other processors. Before the correct value of the record can be returned to the calling process, all of the accesses to the individual remote objects via access types must be satisfied. The communications delays can be compounded and system response times are unpredictable.

The approach advocated in this report is *source code allocation*, where the tested software is partitioned into separate source modules, which are then distributed to the individual processors where they must be individually compiled and linked. This approach is less transparent, in that the distributed modules may be individually modified, but it is supported by currently available compilers and run-time environments.

It is the view of this report that the full facilities of the Ada language should be provided to the programmer, even at the risk of substantial performance costs associated with the distributed program. The programmer is expected to build modules that are well structured, with clean, simple interfaces, using proper software engineering techniques.

5.2 PARTITIONING ALGORITHM

5.2.1 Prepartitioning Versus Postpartitioning

The stage of the software development life cycle, at which the partitioning process is to occur, is cause for considerable debate. The design of the program with the hardware topology in mind is prepartitioning. Suggested changes to Ada have included the addition of a construct to bind sections of code to a processor. At present, compiler pragmas are used to bind sections of code to processors. (As an example, the pragma SITE(n) has been defined (reference 3) to associate all code following the pragma with a specific processor, indicated by the parameter, n.)* Also, virtual nodes are used for prepartitioning as well as the separate communicating programs approach.

The advantages of prepartitioning are that, not only do the source listings reflect the actual dispersion of the software, but the programmer's awareness of the hardware topology generally leads to a high level of optimization. The visibility of the hardware helps deal with issues such as resiliency and robustness. However, modification becomes awkward for anyone but the original programmer, and a substantial change in the hardware may invalidate large sections of the program (if the functionality is not lost, at the very least the optimization is lost). Also, prepartitioning significantly affects program portability.

Postpartitioning is advocated in this report for several reasons. First, Ada has no facilities for configuration management, so it would seem inappropriate to include such information inside the program. Secondly, there is a need to validate the functional design of the program. The partitioning process occurs after the design is complete and does not impinge on the software development process, so the full advantages of the Ada language and current Ada programming support environments (APSE) are available to the programmer.

Thirdly, and perhaps, most significantly, the developed software is portable and can be mapped onto different hardware configurations. The

---

*The pragma ALLOCATE(task, processor) has also been used (reference 4) to support the binding of tasks to processors.

underlying hardware can be changed or the mapping of the program to the hardware can be changed without changing the software. There may be unacceptable hidden costs associated with a partition, but an individual partition can be tested and the partitioning process reiterated, fine-tuning the partition, until an acceptable distribution is found.

Postpartitioning requires the definition of a language to specify the partitioning and any repartitioning would require someone familiar with this language. However, it is feasible that programming tools (discussed in section 5.2.2) can be built to aid the user with the partitioning.

5.2.2 Use of an Adaptor

The implementations of the Ada constructs (as described in section 4) could all be provided by a code generation tool. This adaptor would take the compiled program, developed and tested on a single processor, and the specification of the partitioning, as inputs and produce individual partitioned units for each processor (references 3 and 32). These units are compiled and linked on their respective hardware. The complete process is referred to as *source translation*. An illustration of the use of the adaptor in the partitioning process is shown in figure 7.

Some level of intelligence could be programmed into the adaptor to assist in the partitioning process. The adaptor could be used to identify potential overheads. For example, the adaptor could warn the programmer to the injudicious use of access types across machine boundaries. Partitioning algorithms (discussed in section 5.2.3) could be introduced into the adaptor, as well as hardware capabilities and code analysis.

5.2.3 Partitioning Algorithm

The algorithm used to partition the program could follow several paths. There are many factors contributing to the efficiency of the distribution. A good partitioning will maximize the number of local data references to minimize interprocessor data transfers, as well as consider other resources, such as processor capacity, memory, system load balancing, real-time response, and application-dependent resources.

Perhaps the most obvious parameter is communications costs. The individual costs of object access, RPC, etc., can all be estimated, and these costs can be used to compare one distribution with another. Graph theory can be applied naturally, with iterative assignment-improvement techniques, to produce an acceptable algorithm of this nature (reference 33).

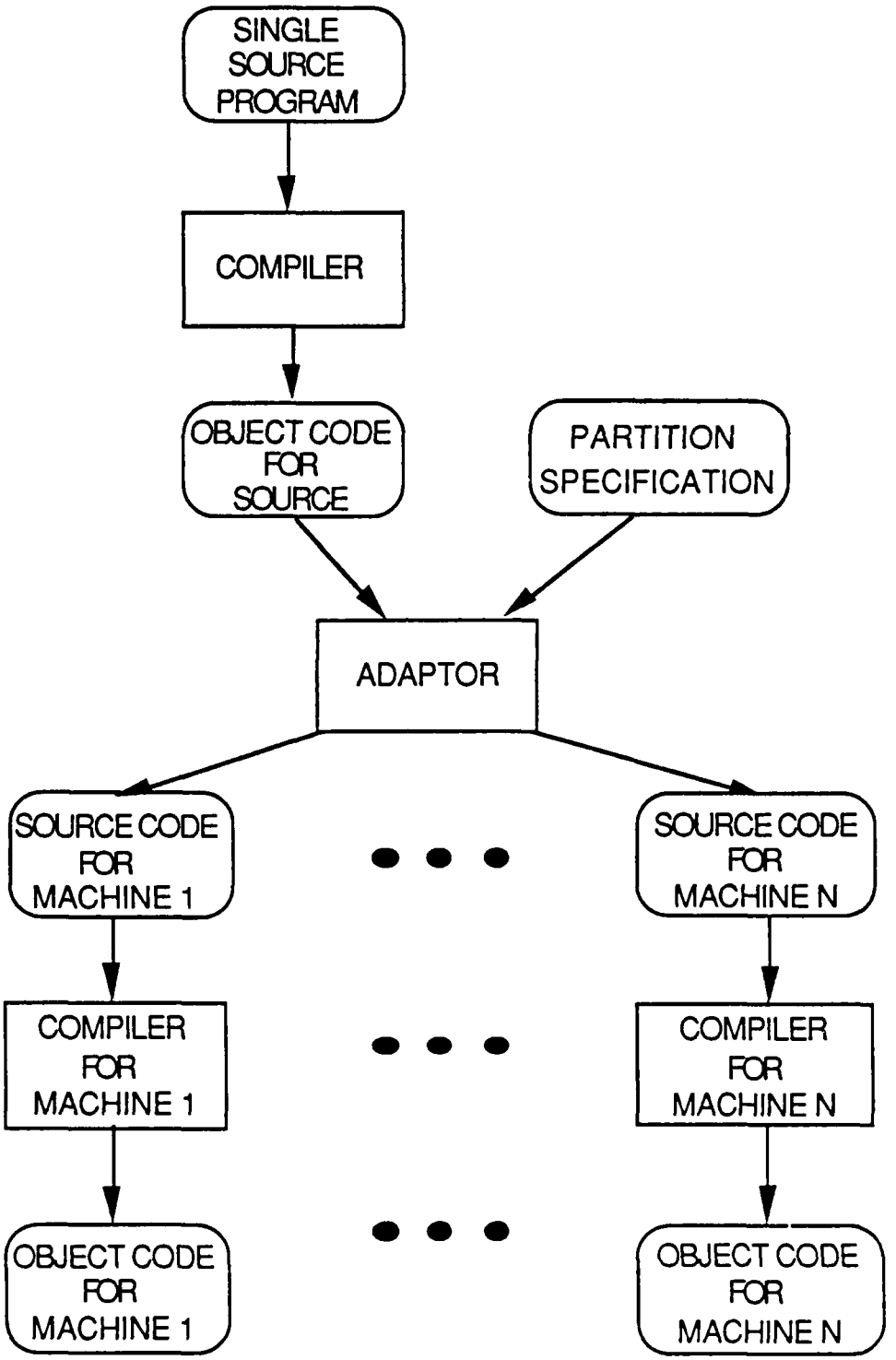


Figure 7. Partitioning a Program Using an Adaptor

Associated with communications costs is the decision to use connection-oriented or connectionless services (see section 6.1 for a more detailed discussion of the pros and cons of both types of services). This decision normally is based on required system response time versus reliability requirements. The decisions could be based individually on each transfer across the network, or a global decision could be made.

Another important parameter is the potential for efficient use of the hardware. There are wide variations in processor speed, so the more heavily loaded partitions may be allocated to the fastest processors. Also, certain processor types support certain types of applications. For example, a complex matrix multiplication would be assigned most naturally to a vector or array processor.

5.3 PARTITIONING THE PROGRAM WITH THE KERNEL

The approach taken in this report to partitioning the program is to allow full use of the Ada language. However, communications costs must be factored into the partitioning process. This will inevitably result in some restrictions in the way Ada is used. A notable restriction will be the limitation of the use of access types, since embedded access types can result in significant hidden communications costs. The partitioning algorithm should provide feedback to the design process to iteratively improve the efficiency of the design when applied to the distributed environment.

The partitioning of the original program is based on the division of the program into several executable objects. These objects may be packages, subprograms, tasks, data objects, or any combination, and each object is assigned to a particular processor. One object is designated as the main program, and its processor is designated as the main processor. As described in section 4, all type declarations, including data types, task types, and generic units are replicated on each processor.

The global strategy for handling references by a local executable object to remote objects is based on the inclusion of local agents. Each agent serves one particular remote executable object and contains the code necessary to communicate a request to the object (or, more correctly, the agent for remote object) and receive the reply from the object. The agents are nested exactly as in the master program, thus preserving the original scope. The local agent behaves exactly as if the remote object were located on that processor, returning any results to the process making the original request. The remote object, too, has an agent, which receives remote requests on behalf of the object, passes the request to the object, receives the results (if any), and returns the results in a reply to the requestor.

The main thread of control will exist on one processor only. The other processors will contain a null program body in the main unit, but will possess all of the specified dependents, as in the source program. Since the agents maintain the scope of the source program, the order of elaboration is maintained.

An example of a program distribution using this approach is illustrated in figure 8. The program is divided into a main unit and two subunits (S1 and S2) allocated to three machines (M1, M2, and M3). The main unit calls an object in S1, which, in turn, calls an object in S2. On machine M1, the subunits are replaced by their respective local agents, and calls to either subunit are handled by the agents. On machine M2, an agent is created to handle requests from machines M1 or M3, addressed to subunit S1. Subunit S2, is replaced by a local agent as on machine M1. On M3, S2 will have an agent to receive requests from M1 and M2, and S1 is replaced by its local agent. If the main unit on M1 calls an object in S1, the request is given to the agent for S1 on M1, which sends the request to the agent for S1 on M2. This agent will then call the corresponding object in S1 and send the reply back to the agent for S1 on M1, who returns the result to the main unit (this is essentially the RPC mechanism, described in section 4.2). If S1 requires a call to an object in S2, S1 calls the agent for S2 on M2 (which sends the request to the agent for S2 on M3 by the same mechanism), prior to returning the result to M1.

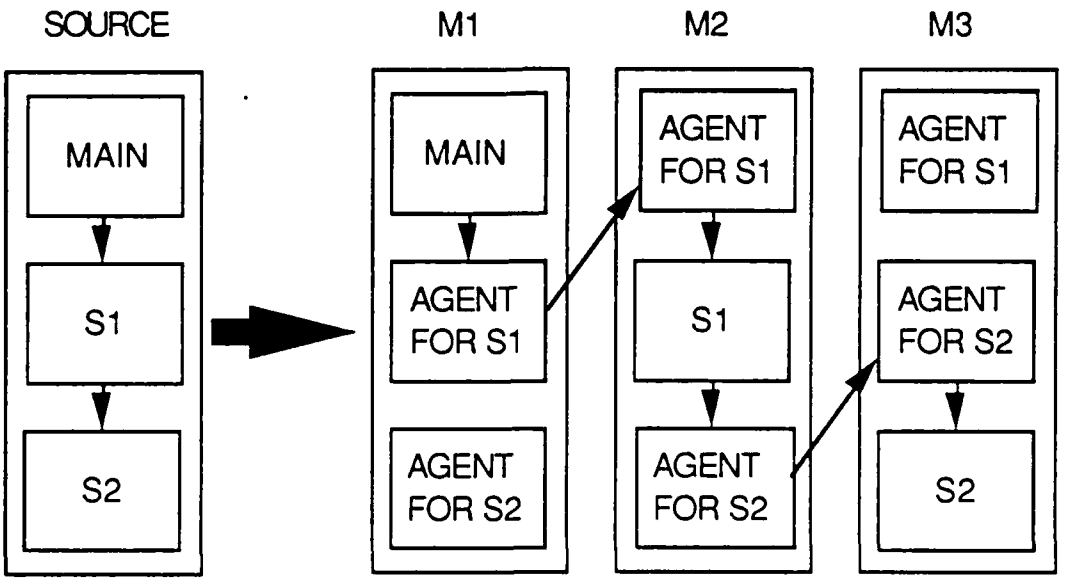


Figure 8. Example Distribution of a Program

6. ISSUES

6.1 DATAGRAM VERSUS VIRTUAL CIRCUIT SERVICE

The question of whether to use connection-oriented (i.e., virtual circuit) service or connectionless (i.e., datagram) service is an issue at practically every level of communications. Connectionless service offers speed, but is susceptible to out of order messages, corrupted messages, or even missing messages. Virtual circuits, on the other hand, provide reliable service, delivering messages in the order they were sent, but incur the high overhead of connection setup, error detection and recovery, and buffer allocation. This issue will be examined at the application layer (including all layers above the transport layer), the transport layer, the network layer, and the data link layer.

6.1.1 Application Layer

The choice of underlying service at the application layer is more a matter of the requirements of the application environment at this level than at any other level. If the application is time-critical, but can survive an occasional need to retransmit a request, then the communications system should use datagrams as the underlying service. Much of the work to date in the distribution of Ada has employed datagrams as the underlying communications service (references 9 and 34). However, virtual circuits provide point-to-point service and messages are guaranteed to arrive intact and in the order they were sent. This is the assumption used for the examples in section 4.

The use of connection-oriented service at the application level implies an overhead that must be considered carefully. If connections are expected to be long-lived, then dedicated server-client connections (between agents) are worth the overhead. Connection setup may require a considerable amount of control messages before any actual messages may be sent, especially if a transport, network, or data link connection must be set up also. However, the rationale of this report is that connection setup may begin at process creation time (i.e., connection setup is included in the initialization code of a package or as a function called to initialize a local task variable during its declaration). That is, connections are initialized before the actual working part of the application begins to execute. Connections, used in this way, can also serve to synchronize the completion of the elaboration of the distributed program. If this approach is possible, then the actual transmission of messages will be both reliable and fast, with speeds approaching (and possibly even exceeding) that of connectionless service.

Connectionless service cannot guarantee the reliable delivery of messages. Thus, applications requiring reliable delivery must buffer the

message at the source and utilize a protocol that uniquely acknowledges the message. This is accomplished by the kernel through the use of ACK (for positive acknowledgments) and NAK (for negative acknowledgments) message types. The receipt of any message requires the immediate return of an ACK or NAK message. Only when an ACK message is received can the buffer be released. Applications using transaction processing follow this model of communications. Also, the asymmetry of the client-server model seems to imply the use of datagrams, since the server can accept requests from any client to which it is visible.

Acknowledged connectionless service also provides for acknowledgment of messages, but ISO has been slower to develop standards for acknowledged connectionless service. At present, the standards specify that receipt is acknowledged only at the data link level. That is, messages are guaranteed to be delivered to the doorstep, but not guaranteed to be picked up. It is possible that remote processes may be synchronized more closely using acknowledged connectionless service, since the communication delay is bounded more tightly, but the synchronization is not exact.

Connection-oriented services may not provide sufficient end-to-end error recovery. Applications may require more extensive guarantees than are provided by virtual circuits. That is, an application may need to know that not only was the message correctly received, but also that it was correctly interpreted, or that the proper response was taken. This requires the use of application layer protocols. The communications kernel provides for these protocols via message types. These message types can be individually specified by the application to implement the protocols, depending on the requirements. If error recovery is to be handled by the application layer, then it makes sense to use datagram service as the underlying communications service. In any case, the kernel provides all the tools and allows the application the flexibility to implement the desired protocols.

6.1.2 Transport Layer

It is at the transport level that much of the overhead for connection-oriented service is incurred. The transport layer is responsible for the reliable end-to-end transfer of data, providing such functions as quality of service negotiation, error detection, error recovery, flow control, and multiplexing. Implementations of the ISO class 4 transport protocol (all transport layer functions) specifications may be quite complex. However, if the application requires reliable delivery of messages, then the transport layer had better provide connection-oriented service. In general, the transport layer should provide the functionality necessary to use both connection-oriented and connectionless services.

6.1.3 Network Layer

It is at the network layer that the issue of connection-oriented service versus connectionless service becomes very clouded indeed. Transport layers may offer connection-oriented service and utilize a network layer providing only connectionless service. Conversely (although not common), transport level connectionless service may be built on network level virtual circuits.

It is in the area of routing that connection-oriented and connectionless services exhibit the greatest differences. A virtual circuit has a static route associated with it when the circuit is established, and all packets follow this route. This route may be predetermined to be an optimal route. However, virtual circuits are vulnerable to failures; if an intermediate node fails, then the circuit is shut down. Since the route is predetermined, packets are not rerouted, and many packets may be delayed or even lost. However, an important advantage to the use of virtual circuits within the network layer is that network layer virtual circuits relieve some of the complexity of the transport layer.

Connectionless network service provides for the dynamic routing of packets, each packet being routed independently of all other packets. Datagram networks offer more robust service, greater flexibility, and better fault tolerance than virtual circuits. Network layers can be programmed to learn and, depending on the routing algorithm, may produce routes that are near optimal. Routes can be adapted to bypass failed nodes, and even possible points of congestion (providing queue information is passed between nodes). However, the disadvantage of datagram networks is that the complexity must be in the transport layer, which must then implement class 4 service. The trend in time-critical networks has been toward connectionless network service (references 35 and 36).

6.1.4 Data Link Layer

As with the network layer, the type of service at the data link layer need not match the service at the network layer. Connectionless service at the network layer may be built on top of connection-mode service at the link layer, and network layer virtual circuits may use connectionless data link protocols. Wide area networks, whether connection-oriented or connectionless, usually require virtual circuit service at the data link layer for reliability. Conversely, with the use of ring topologies, there is no routing functionality necessary at the link level. Connectionless link level service has been used to directly support distributed program communications (reference 37).

The use of FDDI carries a restriction that may impact the decision to use connection-oriented service. The specification for FDDI requires that

frames be a minimum length of 128 bytes. Therefore, the control frames of the connection-mode protocols will require padding, thus increasing the total volume of traffic on the subnet. This is a disadvantage to connection-mode service at the link layer. Also, overhead caused by the use of full destination addresses, required by network layer connectionless service, becomes less of a disadvantage, since short frames would be padded anyway.

6.2 LIGHTWEIGHT PROTOCOLS

There has been a movement in the networking community to provide faster transport layer service yet, maintain the reliability of connection-oriented service. The use of FDDI has moved the network bottleneck out of the physical medium and into the processing protocols. Much of the functionality of the ISO specifications for the class 4 transport protocol represents too great an overhead for real-time networks. The French military standard (reference 17) combines the transport and network layers to form a transfer layer. The SAFENET standard of the U.S. Navy specifies the use of both a class 4 transport protocol stack for ISO compatibility and a lightweight protocol stack (also encompassing the network layer) for real-time applications (references 35 and 36).

The express transfer protocol (XTP) (references 10, 38, and 39) is also a combination of the transport and network layers. Data are transferred with greater efficiency by means of a "context", rather than a connection. The context communicates control information as well as data, thereby reducing the number of packets required to maintain a connection. The protocol is designed to be implemented via very large scale integration (VLSI), providing faster service.

One of the main underlying assumptions of XTP is that the system performance depends heavily on the performance at the receiving end of the context. To alleviate receiver complexity, XTP implements all timers with sending operations only. Perhaps, more importantly, XTP utilizes an addressing scheme in which the sequence numbers of the packet represent byte offsets, which can be mapped directly into an upper-layer buffer space. This allows the address lookup to be performed while incoming data are being buffered, so that the lookup will complete in time to process control information at the end of the incoming packet. Since FDDI has a minimum frame size, there will always be a guaranteed time in which to complete this lookup.

The versatile message transaction protocol (VMTP) (references 11 and 40) is a transport level express protocol designed to interface with a connectionless network layer, and so, does not provide network layer functions. The protocol utilizes the transaction (request/reply) method of message transfer, where a client initiates a transaction by sending a request

and the server terminates this transaction by sending a reply. RPC exemplifies this type of communication, so VMTP appears to be capable of some session functionality, as well. VMTP closely resembles acknowledged connectionless service; since, normally, VMTP allows the client to have only one transaction outstanding (stop-and-wait ARQ). However, VMTP also has facilities for multiple message transactions.

VMTP associates transactions with endpoint identifiers (called entities), rather than with network addresses. Entity identifiers are guaranteed to be stable for a specified time period. This allows for process migration between processors. Further flexibility is provided by VMTP in request processing, which can be message handling (the normal way), or procedure handling (invocation of a procedure, as in RPC). Also, idempotent operations may be specified by the protocol, improving the overall efficiency.

Both XTP and VMTP offer selective retransmission and rate-based flow control. Both protocols assume that, with the low error rate of FDDI, errors in packets will occur primarily as a result of buffer overruns at the receiving end. Rated-based flow control provides receivers with a means to control the *rate* at which data are sent to them, not the *volume* of the data. Both protocols omit the implementation of the transport level quality of service negotiation, feeling that this overhead is unnecessary in distributed applications. Also, both XTP and VMTP offer efficient multicast facilities, which are quite useful in the implementation of fault-tolerant systems (see section 6.8 for a discussion of fault tolerance).

The development of lightweight protocols is one example of the enormous impact that advancing technology is bound to have on distributed systems. The high data rates of fiber-optic technology have moved the bottleneck in network communications from the physical transmission of uninterpreted bits across the comunications medium to the processing of these messages within the protocol layers of the ISO/OSI model. Microchip technology (VLSI) is allowing the movement of the lower layers of the model (at present, the data link and physical layers) to silicon, providing much faster execution. Lightweight protocols are expected to follow suit in the near future. In addition, memory limitations are becoming less and less restrictive, as larger and faster memories are being produced at a lower cost.

6.3 ERROR CONTROL

Another issue that will impact the services required by the kernel at each layer is where to place error control. Is the error control at each layer too good or too weak? Will the next higher layer perform the same error control?

Generally, end-to-end error control is always present at, but is not restricted to, the transport layer. However, if the application layer performs all the necessary error checking, the transport layer need not repeat this function. Conversely, if the transport layer contains sufficient error control, then the application layer need not duplicate the same function.

If the transport layer uses the services of network layer virtual circuits, then considerable error control is performed by the network layer and the transport layer is relieved of this burden. On the other hand, connectionless network layers require that complete error checking exist at some higher layer.

Most MAC sublayers provide error checking by means of a checksum on all frames. Error recovery is left to a higher layer. Connection-mode LLC service provides this recovery. This method is chosen by the communications kernel. The rationale is that the error recovery should occur as soon as possible after the error checking. Any errors for which recovery is impossible at the link layer are communicated to the transport layer for recovery at that layer.

It is believed that the overhead incurred via connection-mode data link service is more than balanced by the savings of error recovery at the higher levels. As an example, consider the transmission of a packet requiring several hops to get to its destination. Suppose that an error occurs in the first hop. If recovery occurs at the link level, then the transmission time of the packet will be increased only by the time it takes to repeat the first hop. However, if error recovery is only at the transport layer (or even the application layer), then the error will not be discovered until the packet reaches the final destination, or until the source times out, and the entire route must be repeated.

6.4 NETWORK MANAGEMENT

6.4.1 Network Management Entities

Network management represents an area where the ISO standards are still in the formative stages. The current standards provide for a layer management entity (LME) to be associated with every layer in the ISO/OSI model. The LME provides an interface between the layer and network management for operations such as adjustment of layer parameters, reconfiguration, fault tolerance, and sending and receiving of management control frames or packets. The specifications of the entity that handles this network management have not been provided by the ISO standards.

In networks likely to be used to implement the distribution of Ada, there seems to be a clear division of network management into three areas:

1. Station management, which maintains the state of the station, or node

2. Ring management, which maintains a consistent view of the ring subnet (i.e., reconfiguration, which stations are up, which bridges are active, etc.)*

3. Network management, which maintains a global view of the interconnection of the subnets, encompassing such functions as directory services, routing between subnets, etc.

These management subfunctions could be provided in separate network management entities. Each entity would interface with only certain layers of the ISO/OSI model. This theory is logical, since, for example, a ring management entity would need to interface only with the network and data link layers, at most. Each management entity would communicate with the entities immediately above and below it, as well as its peer management entities on other machines. This should result in a simpler model for network management. Figure 9 illustrates this kind of partitioning.

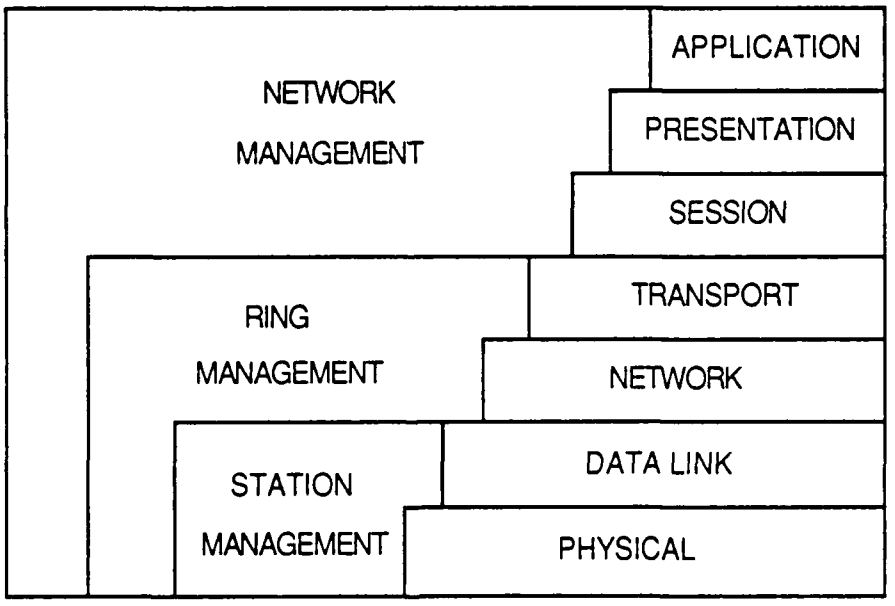


Figure 9. Network Management Model

*Ring management would also apply to any type of single LAN segment.

6.4.2 Directory Services

Directory services are an important function of network management. Most advocate the use of a distributed name server (references 9, 41, and 42) over a centralized server. Although a centralized server can provide a global view of the system and guarantee unique process identifiers, the communications costs of accessing a centralized server can be prohibitive and such a server is vulnerable to failures.

A distributed server would permit more local accesses. Each machine can maintain a local copy of the process name table. Every time a new process is elaborated, the process registers with the name server, perhaps through the initialization code of the package which contains the process or through a call to an initialization function in the declaration of a local task variable. Periodically, each machine could send updates to its neighbors, which would update their tables and regenerate updates to its own neighbors.

6.5 COMMUNICATIONS PRIMITIVES

Although the ISO standards specify the functionality of the communications primitives, there are several factors that need to be considered in an applications-level protocol to support interprocessor communication. Communication may be synchronous or asynchronous. With specific regard to the send operation, the types of communications primitives may be classified as follows:

1. No-wait send, where the sender will pass the message to the underlying layers of the communications kernel and continue with its own execution;

2. Synchronization send, where the sender will wait for receipt of its message before resuming execution;

3. Remote-invocation send, where the sender waits for a reply that the command was carried out, before resuming execution. The reply may contain some result parameters. This is essentially the semantics of the procedure call mechanism.

A disadvantage to the synchronous send is the limitation imposed on the realizable concurrency. Under the normal uniprocessor concurrency model, the sending procedure will be able to continue execution when the receiving process receives the message. However, when applied to the distributed environment, the sending process must wait for an explicit acknowledgment. Semantically, the synchronization send in the distributed

environment more closely resembles the remote-invocation send (without return parameters).

An advantage to the no-wait send is that the user is allowed to perform other operations during the communications delays that are bound to exist while waiting for an acknowledgment of a reply. Forcing the process to wait reduces the potential parallelism. In fact, the possibility exists with the remote-invocation send that the reply may be infinitely delayed due to a server failure.

The ISO standards specify the use of the no-wait send with both connectionless and connection-oriented protocols. The message transfer is initiated and control is immediately returned to the invoker of the primitive. With connection-oriented service, buffering is provided by the transport layer for retransmission of lost or corrupted messages. A synchronous send can always be constructed from a no-wait send followed by a blocking receive (on the acknowledgment) (reference 43). However, synchronous communication is provided by the standards with acknowledged connectionless service, where the communication primitive will wait until the message is acknowledged before returning control to the process that invoked it.

The communications kernel maintains compatibility with the ISO standards by providing asynchronous sends with connectionless and connection-oriented services. When synchronous sends and remote invocation sends are used in the examples, they are constructed from no-wait sends and blocking receives. The synchronous send is implemented with acknowledged connectionless service. The receive operation can be synchronous (i.e., blocks until a message arrives) or asynchronous (i.e., returns if there is no message available, possibly within a specified interval).

6.6 TRANSPARENCY

A strong attempt has been made in this report to provide transparency to the user in the distribution of an Ada program; i.e., the user need not know whether the object being accessed is located in the local program space or on a remote processor. RPC has been designed to provide this transparency. However, complete transparency to the distribution of the program is not possible. There are several reasons for this apparent contradiction.

Communications costs play such an important role in the efficient distribution of an Ada program that they must be considered by the user. The user needs to understand these costs from the start and the user *must* design the program accordingly. Packages, if they are to be distributed, must be designed to minimize their interfaces, thereby, minimizing their communications costs. Access types, when distributed, can generate excessive communications costs, and their use must be restricted.

The remote procedure call mechanism is transparent when all parameters are passed by value. However, the Ada language allows parameters to be passed by reference. This strategy fails for a remote call, since the reference parameter may have no meaning when passed to another machine.

One solution is to replace the call-by-reference mechanism with call-by-copy/restore. The client stub locates the item being referenced by the parameter and passes a copy of the item to the server stub. The server stub then stores the item and creates a pointer to it, and the procedure call is continued in the usual way. If the item is modified and returned, the client stub overwrites the original location referred to by the reference parameter. However, this mechanism can fail, also. Consider the following procedure:

```
procedure TEST(X : in out INTEGER) is
  type REF_INT is access INTEGER;
  REF_X : REF_INT := new INTEGER'(X);
  procedure DOUBLEINCR(P, Q : in out REF_INT) is
  begin
    P.all := P.all + 1;
    Q.all := Q.all + 1;
  end DOUBLEINCR;

begin
  DOUBLEINCR(REF_X, REF_X);
  X := REF_X.all;
end TEST;
```

Normal calls to the procedure TEST return the value of X + 2. However, if DOUBLEINCR is a remote procedure, and call-by-copy/restore is used, then the pointers, P and Q, will refer to separate locations on the remote processor and each will be incremented by one. When these values are restored, the incorrect value of X + 1 will be returned to the procedure TEST.

If the mechanism described in section 4.7 is used, the parameters are passed as remote addresses. P.all and Q.all are replaced by calls to the corresponding remote access stubs. This results in four remote accesses (two remote read operations and two remote write operations) implemented by at least eight messages. Although the correct result is returned, this is hardly an efficient mechanism.

There is also a strong case to be made for nontransparency. The user can take advantage of his knowledge of the hardware topology. This would be appropriate for many one-of-a-kind applications. The user can utilize potential parallelism by bypassing the strict Ada semantics used by the examples in this model. For example, a process may be able to perform useful work after sending a message, instead of waiting for a reply. The process can

check for a reply at a later time and, at that time, can perform any necessary recovery.

The kernel provides flexibility in the communications primitives. Both blocking and nonblocking send and receive primitives are provided. Destinations may be specified by either the application and process names or by a PORT_ID. The CONNECT and LISTEN primitives can be blocking or nonblocking (a polling mechanism is provided to check for the status of nonblocking connection attempts). Messages may be sent point-to-point or broadcast or sent as a high priority alarm.

6.7 ORPHANS

When programs are distributed, many agent tasks are created to support the distribution. Several agent tasks may exist on separate processors, all supporting one server. If the server dies, these agents may continue to run. Also, if server stubs are dynamically generated upon request from a client and the client dies, then these server stubs may continue to run.* These are examples of orphans.

Orphans can cause various problems. Orphans can consume CPU resources and can lock files and other resources, denying any access from other processes. Therefore, it is considered prudent to require termination of these orphans.

The Ada exception handling mechanism allows for abnormal termination of a process engaged in a procedure call or rendezvous to be communicated to the process that invoked the call. If the terminated process is a server, then the server stub will receive an exception. The exception may be propagated to the client via the mechanism described in section 4.6. The client stub can then be terminated. However, all other client stubs that utilize that server must also be informed that the server died, so that they may take the appropriate action.

One method of terminating these orphans is to maintain a log of all tasks that are connected to a given server stub. If the server process goes down, then, when the process comes back up, it checks the log and sends an ABORT_TASK message to any stubs that require termination. The same method is applied to terminate all server stubs with which a particular client is communicating, providing that these stubs were dynamically generated.

---

*This is not a problem with the remote rendezvous, since Ada specifies that, if the caller of the entry is aborted during the rendezvous, then the rendezvous is allowed to proceed.

Orphans may not always require termination. If client stubs are considered to exist for the life of the program, as would be expected using connection-oriented procedures, then they can continue to exist, despite the lack of a server. If one of the stubs attempts to send a request, the kernel will raise the exception, CONNECTION_LOST. The stub will know that the server died, and may attempt to reconnect when the server is back up.

A related problem to the handling of orphans is encountered with the use of timeouts to provide an upper bound on communications. If a client times out and wishes to cancel the service, the server may have already completed (or in the process of completing) the service. The server may need to roll back the effect of the service to maintain a consistent state. To signal the rollback, the client stub would send a predefined exception message (i.e., COMMUNICATION_ERROR) to the server stub, which could invoke an exception handler to perform the actual rollback.

6.8 FAULT TOLERANCE

One of the advantages of distribution is that a hardware failure need not cripple the entire system. Loss of a node in the network should result only in degraded service, not complete loss of service. The execution environment should be able to detect failures and protect the application program from them by either repairing the problem, dynamically reconfiguring the system to recover lost functionality, or explicitly notifying the application of the failure.

Perhaps the most important aspect of fault tolerance is the detection of a failure in the system. Fault detection may be passive or active. Passive detection is provided by the kernel through use of the exceptions, COMMUNICATION_ERROR and CONNECTION_LOST. Ada provides for the use of exception handlers, which may be included in the distribution of the application to provide the necessary degree of fault tolerance.

The disadvantage to passive detection is that a failed processor or a communications break will not be detected until communication is attempted. This may be long after the failure occurred, and damage to the system may have resulted in the interim. For active fault detection, some kind of interprocessor activity is required periodically, and if it ceases, then failure is assumed. The activity associated with active detection is usually in the form of "heartbeat" messages. A heartbeat monitor would exist to invoke the necessary recovery procedures, if a missing heartbeat is detected.

The individual distributable components of the partition are portable, so the application can provide for replication of functionality. Backup processes can be included with the utilization of one-to-many mappings in

the partitioning specification. If these backups are provided by the application design, then a process, on discovery of a failure, can obtain the address of the backup from the name server (the backup is allowed to use the same names in this model, so it can register with the name server, and all subsequent requests for that particular service will be redirected to the backup), connect to the backup, and continue operation.

The replication of data objects in the application needs to be handled with great care. Data inconsistencies can easily result if all copies of the data are not maintained adequately. A two-phase commit protocol has been established for this purpose, and this protocol is well understood in the implementation of distributed data bases.

Fault recovery involves the activation of a recovery procedure. This could be accomplished by the inclusion of a reconfiguration task on each processor, which is suspended on its entry call (reference 44). When a failure is detected via the heartbeat mechanism, the heartbeat monitor will call this entry. The reconfiguration task is as follows:

```
task body RECONFIGURE is
begin
  loop
    accept FAILURE(NODE : in NODE_ID) do
      case NODE is
        when NODE_A =>
          <code to handle failure of processor A>
        when others =>
          <other processors handled similarly>
      end case;
    end FAILURE;
  end loop;
end RECONFIGURE;
```

Once this reconfiguration task is activated, then several possible operations may take place. There may be damaged tasks on the failed processor, and these tasks may need to be aborted. The appropriate backups should then be activated. Also, checking for the consistency of replicated data may be necessary. If the network is partitioned (i.e., divided into two or more separate networks), then one or more processors may be assumed to have failed, and possible reconfiguration information may need to be passed to the network management entities on the remaining processors.

An alternate technique for achieving fault tolerance is the use of the "replicated procedure call" (reference 45). In this method, a server module is replicated among several processors. When a client process (which also may be replicated) requests a service from this module, the service is performed by all copies of the server module. Responses may be analyzed by a "collator", which determines the correct response (perhaps by majority voting) and returns it to the client (or clients). Therefore, failure of one of the servers

does not affect the functionality of the service. Multicast can be used effectively to efficiently implement this type of methodology.

A derivative of the replicated procedure call scheme designates a primary server and a linearly ordered series of secondary servers (reference 46). A client (which may also exhibit this ordered replication) will call the primary server, which then calls the first secondary, and so on, until all servers are called. Under normal operation, the primary server simply returns the results to the primary client, and the secondary servers are informed of the success. However, if the primary fails, the first secondary inherits the responsibilities of the primary and returns the results directly to the client. The advantage to this scheme is that it eliminates both the overhead of checkpointing and recovery in the primary-standby approach and the high message overhead of the replicated procedure call approach (assuming that an efficient multicast facility is unavailable).

6.9 SUGGESTED CHANGES TO Ada

6.9.1 Semaphore Construct

The Ada language was used to implement all layers of the kernel. In the construction of the layers, some difficulties with the use of Ada were encountered. The first of these was Ada's lack of a semaphore construct. Ada does not provide for asynchronous communication to avoid an arbitrary amount of buffer space at run-time and to avoid deadlocks caused by exhaustion of buffer space. Consequently, the rendezvous is the only form of concurrency provided by Ada and many feel that the synchronous nature of the Ada rendezvous is a severe restriction.

There is a strong case to be made for the inclusion of asynchronous communication in the Ada concurrency model. Quite often, real-time systems utilize shared memory to synchronize processes, when only the occurrence of an event, not data, is communicated, and semaphores implement this quite adequately. The semaphore is asynchronous, so a process can communicate the event (i.e., set the semaphore) and continue executing, without the need to wait for a rendezvous. The Ada rendezvous mechanism (discussed next) represents a significant overhead and is not sufficient to implement this type of synchronization.

The overhead associated with the use of the Ada rendezvous can become very significant in real-time systems. Recent benchmarks have shown that the time taken for a rendezvous is more than 20 times that of a subprogram call (reference 47), and the dynamic creation of a task was several times greater. Rendezvous times can even approach network transmission time. It was observed during the construction of the kernel that as many as 55

machine-level instructions may be executed during a single task context switch. Since all the ISO *.indication* primitives were implemented using the Ada rendezvous mechanism, this resulted in a significant loss in the efficiency of the kernel. However, since Ada implements interrupts as task entries, the use of interrupts to implement these primitives would not be expected to greatly improve performance.

6.9.2 Node Construct

The Ada language provides no construct for the definition of a node. This would be of considerable use to the virtual node partitioning method, described in section 5.1.4. Sections of code that would likely be resident on the same processor could be bound together from within the language. For the virtual node construct to be effective, it should be supported by the following:

1. Separate compilation and library units (Both the specification and the body should be library units, so that they may be distributed.)

2. Exception handling facilities to cope with communication failures (Upon a hardware detection of a communications fault, a predefined exception could be immediately propagated into all virtual nodes using the failed communications link.)

3. Dynamic instantiation of virtual nodes without reinitialization of the system. (With the detection of a fault, the backup could be instantiated and service resumed. There is a need for replication of nodes for fault tolerance, and this construct would allow language specification of the replication.)

This "virtual node" or "super package" would be an extension of the package concept, with, perhaps, some restrictions on the communication between different virtual nodes. The *guardian* construct in ARGUS (references 48 and 49) and the *module* construct in CONIC (reference 50) are two examples of implementation of the virtual node concept. Both constructs are quite similar, in that guardians and modules both control processes and data objects within a single address space, each can be dynamically created, and interguardian or intermodule communication is via message passing only. Both guardians and modules are context independent, while Ada packages are not. Guardians are separate compilation units and support compile-time type checking, including compile-time access rights checking associated with capability addressing (described in section 4.7). Both CONIC and ARGUS require a separate configuration language.

Ada package names are not exportable and, thus, do not easily support reconfiguration. To combat this deficiency, the use of a "package type" has been proposed as an extension to the Ada language to support the dynamic

instantiation of a package (references 51 and 52). Package types would behave like task types, in that they may be passed as *in* parameters and may have access types (thus supporting capability addressing of packages, as described for task types in section 4.7). However, package types may be library units, may have subtypes, and may be generic. The notion of a package subtype would be useful to control access to the visible part of the package. Some instantiations of the package type can be allowed to have only limited access to the procedures and objects declared in the package. Package subtypes would be very effective in support of compile-time access rights checking.

Other proposals have included the addition of a "remote" keyword to delineate between objects within the program space and objects outside of the program space.

At present, the only method of defining a node in Ada is to use a compiler pragma to bind parts of the program to a processor. These pragmas have been used successfully to distribute Ada programs (references 2 and 3), but the resultant code is dependent on the existence of a compiler that contains the definition of the pragma.

6.9.3 Predefined Communications Exception

A predefined exception to show a break in the communications link would be a valuable addition to the Ada language. As described above, the run-time system should be able to immediately propagate the exception to exception handlers for all tasks communicating on the failed link, which could reconfigure the system before communications is attempted. The exception, COMMUNICATION_ERROR, included in the kernel, provides this functionality to the application, but it is propagated only on an attempt to communicate on the failed link.

7. CONCLUSIONS

Ada is here to stay. Much investment in time, resources, and personnel has been made to ensure the acceptance of the language. Ada provides excellent software engineering facilities, including separate compilation, modular design, information hiding, extensive compile time error checking, and exception handling. Although the real-time response of implementations of the Ada tasking model still have a way to go, Ada represents a suitable language for real-time applications and distributed environments.

The distribution of Ada programs offers a wealth of opportunities for parallel execution, fault tolerance, reconfigurability, and cost savings; however, distribution also involves a number of difficult issues, including communications delays, consistency of shared data, the use of access types across processors, and task migration. This report has examined these and other issues, as well as the relevant approaches in the design of distributed Ada programs. It should be clear from this report that these issues are a vital part in the development of a useful distributed Ada environment.

The evolution of Ada will probably be toward distributed compilers, but the recommended interim approach should utilize the full advantages of the Ada language and the current APSE tool set, while maintaining program portability. The author feels that the approaches discussed in this report utilizing the distributed Ada message-passing communications kernel presented here are able to supply the user with these advantages and the flexibility to design programs to suit a wide range of application requirements.

8. REFERENCES AND BIBLIOGRAPHY

8.1 REFERENCES

1. Richard A. Volz and Trevor N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs," *IEEE Transactions on Computers*, vol C-36, no. 4, April 1987, pp 449–459.

2. Judy M. Bishop, Stephen R. Adams, and David J. Pritchard, "Distributing Concurrent Ada Programs by Source Translation," *Software–Practice & Experience*, vol 17, no. 12, December 1987, pp 859–884.

3. R. A. Volz, P. Krishnan, and R. Theriault, "An Approach to Distributed Execution of Ada Programs," JPL Workshop on Ada Programs, January 1987.

4. David A. Fisher and Richard M. Weatherly, "Issues in the Design of a Distributed Operating System for Ada," *Computer*, vol 19, no. 5, May 1986, pp 38–47.

5. C. Atkinson, T. Moreton, and A. Natali, *Ada for Distributed Systems*, Cambridge University Press, Cambridge, U. K., 1988.

6. A. Burns, A. M. Lister, and A. J. Wellings, "Ada Tasking Implementation Issues," *Ada User (UK)*, vol 8, no. 2, 1987, pp 30–39.

7. Dennis Cornhill, "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, 15–18 October 1984, pp 153–162.

8. Rakesh Jha, Gregory Eisenhauer, J. Michael Kamrad, and Dennis Cornhill, "An Implementation Supporting Distributed Execution of Partitioned Ada Programs," *Ada Letters*, vol 9, no. 1, January/February 1989, pp 147–160.

9. Andrew D. Birrell and Bruce J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol 2, no. 1, February 1984, pp 39–59.

10. Greg Chesson, "The Protocol Engine Project," *Unix* Review*, September 1987, pp 70–77.

---

*UNIX is a trademark of AT&T.

11. David R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of Communication Systems," *Proceedings of SIGCOM'86*, ACM, 5–7 August 1986, pp 406–415.

12. "FDDI Token Ring, Media Access Control (MAC)," Draft Proposed American National Standard, Accredited Standards Committee X3T9.5, 28 February 1986.

13. *Logical Link Control,* American National Standard ANSI/IEEE Std. 802.2, Institute of Electrical and Electronics Engineers, 1985.

14. "Draft IEEE Standard 802.1 (Part A)–Overview & Architecture," Institute of Electrical and Electronics Engineers, October 1985.

15. "Draft Proposed Addendum to IEEE 802.2 Logical Link Control–Acknowledged Connectionless Service," Institute of Electrical and Electronics Engineers, December 1985.

16. "Information Processing Systems–Open System Interconnection–Transport Protocol Specification; ISO/TC97/SC16/WG6," *Computer Communication Review,* vol 12, no. 3-4, International Standards Organization, July/October 1982, pp 24–67.

17. *Military Real Time Local Area Network–A Reference Model (Transfer Layer)–GAM-T-103,* Ministere de la Defense Republique Francaise, 9 February 1987.

18. Mike Tedd, Stefano Crespi-Reghizzi, and Antonio Natali, *Ada for Multi-Microprocessors,* Cambridge University Press, Cambridge, England, 1984.

19. Justin Jackson, "Distributed Ada Tasks," *WESCON/86 Conference Record 5/4,* November 1986.

20. Dennis Cornhill, "Partitioning Ada Programs for Execution on Distributed Systems," *IEEE Proceedings of the International Conference on Data Engineering,* April 1984, pp 364–370.

21. Richard M. Weatherly, "A Message-Based Kernel to Support Ada Tasking," *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments,* 15–18 October 1984, pp 136–144.

22. S. A. Schuman, E. M. Clarke, Jr., and C. V. Nikolaou, "Programming Distributed Applications in Ada: A First Approach," *Proceedings of the 1981 International Conference on Parallel Processing,* August 1981, pp 38–49.

23. Derek S. Morris and Thomas Wheeler, "Distributed Program Design in Ada: An Example," *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, April 1986, pp 21–29.

24. A. D. Hutcheon and A. J. Wellings, "Ada for Distributed Systems," *Computer Standards & Interfaces*, vol 6, 1987, pp 71–81.

25. A. Dapra et al., "Using Ada and APSE to Support Distributed Multiprocessor Targets," *Ada Letters*, vol 3, no 6, May/June 1984, pp 57–65.

26. R. A. Stammers, "Ada on Distributed Hardware," *Concurrent Languages in Distributed Systems–Hardware Supported Implementation, Proceedings of the IFIP WG 10.3 Workshop*, 1985, pp 35–40.

27. C. Atkinson and S. J. Goldsack, "Ada for Distributed Systems–A Compiler Independent Approach," *Proceedings of the 7th IFAC Workshop on Distributed Computer Control Systems*, October 1988, pp 25–36.

28. P. Inverardi, F. Mazzanti, and C. Montangero, "The Use of Ada in the Design of Distributed Systems," *Ada In Use, Proceedings of the Ada International Conference, Paris*, 14–16 May 1985, pp 85–96.

29. Dennis Cornhill, "A Survivable Distributed Computing System for Embedded Application Programs Written in Ada," *Ada Letters*, vol 3, no. 3, 1983, pp 79–87.

30. Gregory Eisenhauer, Rakesh Jha, and J. Michael Kamrad, "Distributed Ada–Methodology, Notation, and Implementation," *AIAA Computers in Aerospace Conference*, October 1987, Technical Papers, pp. 123–128.

31. Rakesh Jha, J. Michael Kamrad, and Dennis Cornhill, "Ada Program Partitioning Language: A Notation for Distributing Ada Programs," *IEEE Transactions on Software Engineering*, vol SE-15, no. 3, 1989, pp 271–280.

32. Gregory McFarland, Peter Brennan, John D. Litke, and Michael S. Restivo, "A Tool Set for Distributed Ada Programming," *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, May 1988, pp 71–79.

33. A. Chow and M. Feridun, "A Methodology for Partitioning Real-Time Ada Software for Distributed Targets," GTE Laboratories Final Report, September 1986.

34. Judy Bamberger and Roger Van Scoy, "Distributed Ada Real-Time Kernel," *Proceedings of the IEEE 1988 National Aerospace & Electronics Conference: NAECON 88*, vol 4, 1988, pp 1510–1516.

35. "SAFENET I Specification," Draft SAFENET Standard, Revision 2, 8 June 1988 (UNCLASSIFIED).

36. "SAFENET II Specification," Draft SAFENET Standard, Revision 0, 12 January 1989 (UNCLASSIFIED).

37. N. D. Gammage, R. F. Kamel, and L. M. Casey, "Remote Rendezvous," *Software–Practice and Experience*, vol 17, no. 10, October 1987, pp 741–755.

38. Greg Chesson, et al., "Express Transfer Protocol Definition," Revision 3.2, Protocol Engines, Inc., June 1988.

39. Marc Cohn, "A High-Performance Transfer Protocol for Real-Time Local Area Networks," MILCOM'88, 25 October 1988.

40. David R. Cheriton, "VMTP: Versatile Message Transaction Protocol–Protocol Specification," Preliminary Version 0.6, 10 January 1988.

41. Sape J. Mullender and Paul M. B. Vitanyi, "Distributed Match-Making for Processes in Computer Networks," *ACM Operating Systems Review*, vol 20, no. 2, April 1986, pp 54–64.

42. A. Fantechi, P. Inverardi, and N. Lijtmaer, "Using High Level Languages for Local Computer Network Communication: A Case Study in Ada," *Software–Practice And Experience*, vol 16, no. 8, August 1986, pp 701–717.

43. Barbara Liskov, "Primitives for Distributed Computing," *Proceedings of the Seventh Symposium on Operating System Principles*, SIGOPS/ACM, December 1979, pp 33–42.

44. John C. Knight and John I. A. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems," *Ada Letters*, vol 4, no. 3, November/December 1984, pp 53–64.

45. Eric C. Cooper, "Replicated Procedure Call", *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, August 1984, pp 220–232.

46. K. S. Yap, P. Jalote, and S. Tripathi, "Fault Tolerant Remote Procedure Call," *The 8th International Conference on Distributed Computing Systems*, IEEE, June 1988, pp 48–54.

47. R. M. Clapp, et al., "Toward Real-Time Performance Benchmarks for Ada," *Communications of the ACM*, vol 29, no. 8, August 1986, pp 760–778.

48. Barbara Liskov, "On Linguistic Support for Distributed Programs," *IEEE Transactions on Software Engineering*, vol SE-8, no. 3, May 1982, pp 203–210.

49. Barbara Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, vol 5, no. 3, July 1983, pp 381–404.

50. Jeff Kramer and Jeff Magee, "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering*, vol SE-11, no. 4, April 1985, pp 424–436.

51. Warren H. Jessop, "Ada Packages and Distributed Systems," *SIGPLAN Notices*, vol 17, no. 2, February 1982, pp 28–36.

52. Gregory D. Buzzard and Trevor N. Mudge, "Object-Based Computing and the Ada Programming Language," *Computer*, vol 18, no. 3, March 1985, pp 11–19.

8.2 BIBLIOGRAPHY

Armitage, James W. and James V. Chelini, "Ada Software on Distributed Targets: A Survey of Approaches," *Ada Letters*, vol 4, no. 4, 1985, pp 32–37.

Bishop, Judy M., "Practical Issues of Concurrency in Ada," *Major Advances in Parallel Processing*, 1987, pp 236–246.

Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, vol 21, no. 11, 1978, pp 934–941.

Day, John D. and Hubert Zimmerman, "The OSI Reference Model," *Proceedings of the IEEE*, vol 71, no. 12, 1983, pp 1334–1340.

Department of Defense, United States Government, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, 17 February 1983.

Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, vol 21, no. 8, 1978, pp 666–677.

Hosch, Frederick A., "Message Passing and Administrators in Ada," *Ada Letters*, vol 9, no. 2, 1989, pp 106–117.

Ichbiah, J. D., J. G. P. Barnes, R. J. Firth, and M. Woodger, "Rationale for the Design of the Ada Programming Language," Honeywell, Minneapolis, MN and Alsys, La Celle Saint Cloud, France, 1986.

Lampson, B. W. et al., *Distributed Systems–Architecture & Implementation–An Advanced Course*, Springer-Verlag, Berlin, Heidelberg, Germany, 1981.

Paulk, Mark C., "Problems with Distributed Ada Programs," *International Phoenix Conference on Computers & Communications: PCCC'86, 1986 Conference Proceedings*, pp 396–400.

Pratt, S. J., "The Alchemy Model: A Model for Homogeneous and Heterogeneous Computing System," *ACM Operating Systems Review*, vol 20, no. 2, 1986, pp 25–37.

Shatz, Sol M., "Communication Mechanisms for Programming Distributed Systems," *Computer*, vol 17, no. 6, 1984, pp 21–28.

Stallings, William, *Handbook of Computer Communications Standards, Volume 1–The Open Systems Interconnection (OSI) Model and OSI-Related Standards; Volume 2–Local Network Standards*, MacMillan, New York, 1987.

Stankovic, John A., "Software Communication Mechanisms: Procedure Calls Versus Messages," *Computer*, vol 15, no. 4, 1982, pp 19–25.

Tanenbaum, Andrew S., *Computer Networks*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1988.

Volz, Richard A., Trevor N. Mudge, Arch W. Naylor, and John H. Mayer, "Some Problems in Distributing Real-Time Ada Programs Across Machines," *Ada In Use–Proceedings of the Ada International Conference, Paris*, 14–16 May 1985, pp 72–84.

Volz, Richard A., Trevor N. Mudge, Gregory D. Buzzard, and Padmanabhan Krishnan, "Translation and Execution of Ada Programs : Is It Still Ada?" *Space Station Automation II, Society of Photo-Optical Instrumentation Engineers (SPIE)*, vol 729, October 1986, pp 114–125.

Wellings, A. J., G. M. Tomlinson, D. Keefe, and I. C. Wand, "Communication Between Ada Programs," *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, 15–18 October 1984, pp 145–152.

APPENDIX
COMMUNICATIONS PACKAGE SPECIFICATION

The following is the Ada package specification of the distributed Ada message passing kernel. The data types, procedures, and exceptions listed are visible to an application using this kernel via an Ada *with* clause.

```
with TYPES; use TYPES;
package COMMUNICATIONS is

------------------------------------------------------------------------
--  ABSTRACT
--    This package contains the primitives and constructs necessary for
--    an Ada process to communicate with another Ada process.  All of
--    the data types to be passed from process to process are declared
--    within this package specification.  Communication may be
--    connectionless (i.e. datagram), connection-oriented (i.e. virtual
--    circuit) or acknowledged connectionless (i.e. acknowledged
--    datagram).  A process is defined as an operating system process,
--    constrained to fit within a single address space.  There is a
--    natural one-to-one mapping between Ada tasks (or other Ada program
--    units) in an application to schedulable operating system
--    processes.
--
--    Each process will initially register with the system.  The process
--    receives a "system port" through which it can send and receive
--    messages (connectionless messages only).  This system port is used
--    to identify the process to the name server, which, in turn,
--    supplies the port as the address of the process to other processes
--    wishing to communicate with it.  A process can establish a
--    dedicated connection to another process, either by connecting
--    directly or by listening for a client process attempting to
--    connect to it.  Since the Ada rendezvous and procedure call follow
--    the client-server model, the communication model also supports
--    this.
--
------------------------------------------------------------------------

    type PORT_ID is private;


--    An Ada process is defined to the system as a two-part name.  The
--    application can be an Ada program or group of Ada programs,
--    running on a single host machine (although not necessarily on a
--    single processor).  Many applications may run together on a single
--    host.  The process is an Ada task (or other program unit) running
--    within the application.

    subtype APPLICATION_UNIT_NAME is STRING;

    subtype PROCESS_NAME is STRING;


--    Some control messages may be required by the application to
--    implement the transaction model (request/reply), half-duplex
```

```
--    conversations, graceful disconnect and negative acknowledgments.
--    Also, the server can return an exception to the client, if needed,
--    to denote the failure of the remote procedure call or remote
--    rendezvous.  Note the addition of COMMUNICATION_ERROR as a
--    possible exception.

    type CONTROL_TYPE is (DATA, ACK, NAK, REQUEST, REPLY, SERVER_BUSY,
                          ALARM, NO_MORE_DATA, ABORT_TASK, GIVE_TOKEN,
                          TOKEN_PLEASE);

    type EXCEPTION_TYPE is (NUMERIC, CONSTRAINT, PROGRAM, STORAGE,
                            TASKING, COMMUNICATION, USER_DEFINED,
                            NO_EXCEPTION);


--    The following data types will be project dependent.  All data
--    types to be passed within messages must be declared, as well as
--    the necessary control messages.  The data type, CONNECT_DATA, is
--    provided in order for the process initiating the connection to
--    supply the receiving process with its initial system port, which
--    is then supplied to the name server, which returns the identity of
--    the source process.

    type INFORMATION_TYPES is (NAVIGATION_DATA, FIRE_CONTROL_DATA,
                               CONTROL_MESSAGE, CONNECT_DATA, NO_DATA);

    type NAVIGATION_MESSAGE_TYPE is record
        LONGITUDE : FLOAT;
        LATITUDE  : FLOAT;
    end record;

    type FIRE_CONTROL_MESSAGE_TYPE is record
        SOME_INFORMATION : INTEGER;
        SOME_MORE        : INTEGER;
    end record;

    type INFORMATION_RECORD(CLASS : INFORMATION_TYPES := NO_DATA)
    is record
        case CLASS is
            when NO_DATA =>
                null;
            when CONNECT_DATA =>
                SOURCE_PORT : DOUBLE_BYTE;
            when others =>
                CONTROL : CONTROL_TYPE := DATA;
                ERROR   : EXCEPTION_TYPE := NO_EXCEPTION;
                case CLASS is
                    when NAVIGATION_DATA =>
                        NAVIGATION_MESSAGE : NAVIGATION_MESSAGE_TYPE;
                    when FIRE_CONTROL_DATA =>
                        FIRE_CONTROL_MESSAGE :
                                        FIRE_CONTROL_MESSAGE_TYPE;
                    when others =>
                        null;
                end case;
        end case;
    end record;
```

```
    type MESSAGE_TYPE is access INFORMATION_RECORD;

    type CONNECTION_STATUS is (SUCCESS, PROCESS_NAME_NOT_FOUND,
                               APPLICATION_NAME_NOT_FOUND,
                               PENDING, FAILURE);

    type RECEIVE_STATUS is (SUCCESS, NO_MESSAGE, PORT_FAILURE);

    type ACK_NACK_STATUS is (ACKNOWLEDGED, NOT_ACKNOWLEDGED,
                             NO_DESTINATION);

    CONNECTION_LOST, COMMUNICATION_ERROR : exception;
         -- CONNECTION_LOST is raised when a process attempts to
         -- communicate on a port that is no longer connected.
         -- COMMUNICATION_ERROR is raised to report the failure of a
         -- message transmission.


    procedure REGISTER(MY_APPLICATION : in APPLICATION_UNIT_NAME;
                       MY_PROCESS     : in PROCESS_NAME;
                       PORT           : out PORT_ID);
         -- In order to make its identity visible to the system, a
         -- process can register.  A message queue is created to hold
         -- incoming messages.  Registration is necessary in order to
         -- receive broadcast messages, datagrams and acknowledged
         -- datagrams.  However, connections may be initiated and
         -- connectionless messages may be sent without issuing a
         -- register primitive.  In these cases the process is
         -- automatically registered.


--       The following connect primitives are supplied.  The CONNECT
--       procedures will initiate a connection to the specified process, on
--       behalf of the calling process, while the LISTEN procedures will
--       wait for a process to attempt to connect to the calling process.
--       In the client-server model, the server issues a LISTEN, while the
--       client issues a CONNECT.  The server does not know the identity of
--       the caller until the connection is established.
--
--       Those procedures that do not return a STATUS parameter will block
--       until a connection is established.  Connections initiated with the
--       non-blocking procedures may be polled with CONFIRM_CONNECTION
--       until the connection is established.
--
--       With the CONNECT primitives, if the specified destination process
--       has previously sent a request for connection to the calling
--       process at the specified priority, the connection is immediately
--       established.  If the priority parameters differ, then two
--       connections are opened, effectively opening two "one-way pipes",
--       each to send messages at the respectively requested priority.
--       A single queue at each end is shared by each "pipe".  With the
--       LISTEN primitives, if any pending requests for connection to the
--       calling process exist, then the one with the highest priority is
--       selected and a connection is immediately established.

    procedure CONNECT(MY_APPLICATION : in APPLICATION_UNIT_NAME;
                      MY_PROCESS     : in PROCESS_NAME;
                      TO_APPLICATION : in APPLICATION_UNIT_NAME;
```

```
                    TO_PROCESS      : in PROCESS_NAME;
                    PRIORITY        : in PRIORITY_CLASS;
                    PORT            : out PORT_ID;
                    STATUS          : out CONNECTION_STATUS);

    procedure CONNECT(MY_APPLICATION : in APPLICATION_UNIT_NAME;
                    MY_PROCESS      : in PROCESS_NAME;
                    TO_APPLICATION  : in APPLICATION_UNIT_NAME;
                    TO_PROCESS      : in PROCESS_NAME;
                    PRIORITY        : in PRIORITY_CLASS;
                    PORT            : out PORT_ID);

    procedure LISTEN(MY_APPLICATION : in APPLICATION_UNIT_NAME;
                   MY_PROCESS      : in PROCESS_NAME;
                   PORT            : out PORT_ID;
                   STATUS          : out CONNECTION_STATUS);

    procedure LISTEN(MY_APPLICATION : in APPLICATION_UNIT_NAME;
                   MY_PROCESS      : in PROCESS_NAME;
                   PORT            : out PORT_ID;
                   TO_APPLICATION  : out APPLICATION_UNIT_NAME;
                   TO_PROCESS      : out PROCESS_NAME);

    procedure CONFIRM_CONNECTION(PORT   : in PORT_ID;
                                 STATUS : out CONNECTION_STATUS);

    procedure DISCONNECT(PORT : in PORT_ID);
         -- This procedure breaks the connection established with the
         -- CONNECT primitive.  CONNECT and DISCONNECT are symmetric
         -- and must occur in pairs.  An exception, CONNECTION_LOST,
         -- will be raised if an attempt is made to use the connection
         -- after is has been disconnected.  When the disconnect
         -- request is received at the destination entity, the entity
         -- will wait until the message queue at the destination is
         -- empty before closing the connection.  This protocol,
         -- coupled with the transport disconnect protocol, effectively
         -- implements a "graceful disconnect".  That is, when a
         -- DISCONNECT is issued, all outstanding messages are
         -- delivered and acknowledged before the connection is closed.


--    The following send primitives are used to send messages to other
--    processes.

    procedure BROADCAST(MESSAGE        : in MESSAGE_TYPE;
                        MY_APPLICATION : in APPLICATION_UNIT_NAME;
                        MY_PROCESS     : in PROCESS_NAME;
                        PRIORITY       : in PRIORITY_CLASS);
         -- This procedure will broadcast the indicated message to all
         -- registered processes.  Processes must be registered with
         -- the system to receive broadcast messages.


    procedure SEND(MESSAGE        : in MESSAGE_TYPE;
                   MY_APPLICATION : in APPLICATION_UNIT_NAME;
                   MY_PROCESS     : in PROCESS_NAME;
                   TO_APPLICATION : in APPLICATION_UNIT_NAME;
                   TO_PROCESS     : in PROCESS_NAME;
```

```
                PRIORITY        : in PRIORITY_CLASS);
        -- This procedure sends a datagram to the indicated
        -- destination process.  The destination process must be
        -- registered with the system.  The exception,
        -- COMMUNICATION_ERROR, is raised if the destination process
        -- cannot be found.  The message is not acknowledged.


    procedure SEND(MESSAGE   : in MESSAGE_TYPE;
                   PORT      : in PORT_ID);
        -- Once a connection is established, this procedure may be
        -- called to send a message through the source's unique port
        -- to the destination process connected to it.  The
        -- association is established by use of a CONNECT command.


    procedure SEND_ALARM(MESSAGE : in MESSAGE_TYPE;
                         PORT    : in PORT_ID);
        -- This procedure is used to send a high priority message to
        -- the associated destination process.  A prior connection
        -- must have been established.  If the destination cannot be
        -- found or the connection is closed, the exception,
        -- COMMUNICATION_ERROR, is raised.


    procedure SEND(MESSAGE         : in MESSAGE_TYPE;
                   MY_APPLICATION  : in APPLICATION_UNIT_NAME;
                   MY_PROCESS      : in PROCESS_NAME;
                   TO_APPLICATION  : in APPLICATION_UNIT_NAME;
                   TO_PROCESS      : in PROCESS_NAME;
                   PRIORITY        : in PRIORITY_CLASS;
                   STATUS          : out ACK_NACK_STATUS);
        -- This procedure is used to send an acknowledged datagram to
        -- another process.  The recipient process must have
        -- registered.  This procedure will block until the message is
        -- acknowledged.  The appropriate status of the message is
        -- returned.


--    The following receive primitives are used to retrieve the next
--    available message on the queue associated with the indicated port.
--    The procedures which do not return a STATUS parameter will block
--    until a message can be retrieved.  Otherwise, the process will
--    wait for a specified period (note the default of zero wait) before
--    returning.

    procedure RECEIVE(MESSAGE : out MESSAGE_TYPE;
                      PORT    : in PORT_ID;
                      WITHIN  : in DURATION := 0.0;
                      STATUS  : out RECEIVE_STATUS);
        -- This procedure will retrieve a message on the indicated
        -- port within a specified duration and return the appropriate
        -- status of the message.
```

```
    procedure RECEIVE(MESSAGE : out MESSAGE_TYPE;
                      PORT    : in PORT_ID);
        --  This procedure will block until a message can be retrieved
        --  on the indicated port.


    procedure RECEIVE(MESSAGE          : out MESSAGE_TYPE;
                      FROM_APPLICATION : out APPLICATION_UNIT_NAME;
                      FROM_PROCESS     : out PROCESS_NAME;
                      PORT             : in PORT_ID;
                      WITHIN           : in DURATION := 0.0;
                      STATUS           : out RECEIVE_STATUS);
        --  This procedure will retrieve a message on the indicated
        --  port within a specified duration and return the appropriate
        --  status of the message.  The name of the process that sent
        --  the message is also returned.  This procedure is used
        --  primarily with the system port to receive broadcast
        --  messages and datagrams, when the identity of the source is
        --  desired.


    procedure RECEIVE(MESSAGE          : out MESSAGE_TYPE;
                      FROM_APPLICATION : out APPLICATION_UNIT_NAME;
                      FROM_PROCESS     : out PROCESS_NAME;
                      PORT             : in PORT_ID);
        --  This procedure will block until a message can be retrieved
        --  on the indicated port.  The name of the process that sent
        --  the message is also returned.  This procedure is used
        --  primarily with the system port to receive broadcast
        --  messages and datagrams, when the identity of the source is
        --  desired.

private

    type PORT_ID is new INTEGER range 0..255;

end COMMUNICATIONS;
```

INITIAL DISTRIBUTION LIST

Addressee	No. of Copies
SPAWAR (Code 324 (CDR Barbour))	3
DTIC	12
CNA	1