NUSC Technical Document 7000
20 March 1991

Distributed Ada Programs on Heterogeneous Systems

B. W. Stevens
Combat Control Systems Department

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

PREFACE

This report was prepared under NUSC Project No. F45146, "Next Generation Computer Resources (NGCR)," principal investigator B. W. Stevens (Code 2221). The sponsoring activity is the Space and Naval Warfare Systems Command (SPAWAR 231).

The technical reviewer for this report was J. W. Brennan Jr.

Appreciation and gratitude is gratefully extended to Dr. Charles Arnold of the University of Rhode Island, for his instruction and for serving as thesis advisor. His presentations, comments, and wisdom were invaluable during the preparation of this report. I would also like to extend thanks to my coworkers, T. P. Conrad, for his encouragement and constructive criticism, and Mr. Brennan, for his careful attention in detecting not-so-obvious errors and problems.

REVIEWED AND APPROVED: 20 MARCH 1991

P. A. La Brecque
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
	20 March 1991	Final

4. TITLE AND SUBTITLE

Distributed Ada Programs on Heterogeneous Systems

5. FUNDING NUMBERS

6. AUTHOR(S)

B. W. Stevens

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Underwater Systems Center
Newport Laboratory
Newport, RI 02841-5047

8. PERFORMING ORGANIZATION REPORT NUMBER

TD 7000

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Space and Naval Warfare Systems Command
SPAWAR 324
Arlington, VA 22202

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)*

This study investigated distributed applications written in the Ada programming language, in particular, the applications that are implemented on systems dissimilar in underlying hardware architecture and operating systems that require the exchange of data and control. The approach presented in this report rejects the suggestion of modifying both the Ada runtime environment and the Ada language itself to achieve distribution of real-time applications. Distribution can be achieved in the spirit of Ada without use of Ada's tasking features, through use of well-defined standard network or backplane interfaces through other Ada features such as packages and subprograms.

The Ada programming language is mandated as the single high-order language used in implementing systems currently being delivered to the Department of Defense (DoD). Many of these systems are presently under development and are distributed in nature. The Ada programming language contains an abstract feature known as a task, which could lend itself to distribution. The Ada Programming Language Reference Manual, ANSI/MIL-STD-1815A, mentions in a note that "parallel tasks...may be implemented on multi-

14. SUBJECT TERMS

Programming Languages
Distributed Computer Systems
DoD Software

15. NUMBER OF PAGES

61

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

13. ABSTRACT (Cont'd)

computers, multiprocessors, or with interleaved execution on a single physical processor." Currently, many implementations of Ada runtime environments are relatively young when compared with the runtimes of other high order languages. The immaturity of these runtimes is indicated by the fact that many implementors have chosen to interleave execution of tasks on a single physical processor. With many DoD programs bounded by project deadlines and limited budgets, solutions to the Ada distribution problem must be found now. The developer's time should be spent solving application issues, not experimenting with new runtime functionality when developing distributed applications.

It is the author's intent to determine if rigorous definition of an interface to currently existing network protocols such as DECnet, TCP/IP, SAFENET I, and SAFENET II can answer the near term requirements for heterogeneously distributed applications written in Ada.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1 INTRODUCTION

It has long been recognized by the Department of Defense's (DoD) software initiative that the future requirements for defense systems in the 1990's would exhibit the following characteristics among others. The systems will use multiprocessor, networked, and parallel architectures. The cost for developing, evolving, and maintaining defense software will have grown to become a principal factor in the determination of U.S. capabilities [LIE86]. The four technical solutions to controlling this growing software complexity proposed by DoD's software initiative are:

1) Greater use of automation,
2) Higher levels of abstraction,
3) Reusability, and
4) Rapid prototyping.

The three major components of the software initiative currently being used by DoD to research and develop these solutions are:

1) The Ada program,
2) The STARS program, and
3) The Software Engineering Institute.

It has also been accepted in the communications industry that standards are required to govern the physical, electrical, and procedural characteristics of communication equipment [STA872]. Computer vendors, on the other hand, have traditionally attempted to monopolize their customers. DoD has recognized that systems produced by different vendors must be able to communicate with each other. Heterogeneous communication can be assured through the development of communication standards that are adhered to by all vendors producing systems for DoD.

This report addresses the Ada programming language in relation to its use on systems that are parallel and distributed in nature, in particular, those which are not alike in their underlying architecture. This study examines what can be done to reduce the amount of time and cost of building real-time distributed heterogeneous systems using Ada.

2 TYPES OF DISTRIBUTED SYSTEMS

This section will discuss the similarities, variations, and characteristics of distributed systems.

2.1 Conformance Types

Table I delineates a list of distributed system hardware architectures, referred to as conformance types, which will be considered. Conformance type 0 (kernel system) and conformance type 1 (uniprocessor system) essentially have no distribution, but are included for completeness.

Table I Conformance Types

Conformance Type	Description
0	Kernel system
1	Uniprocessor system
2	Multiprocessor system connected by a memory bus (A) or backplane (B) with global memory
3	Multiple computers connected by a backplane bus or a single segment LAN with no global memory or store-and-forward
4	Loosely coupled systems such as multi-segment bridged LANs to full WANs with store-and-forward

2.2 A Taxonomy of DoD System Architectures

There are basically three ways in which multiple processors are distributed. The first model considers placing multiple processors on the same memory bus or backplane with access to global memory and will be referred to hereinafter as multiprocessor systems (conformance type 2). A memory bus is a low protocol media that allows multiple devices to access a common address space. In contrast, a backplane is a high protocol bus that also allows multiple devices to access a common address space. The backplane protocol allows more sophisticated ways for devices to communicate and interact with each other.

The second model is somewhat similar to the multiprocessor system model with one major difference, connection is provided via the more sophisticated

backplane or single segment local area network (LAN) and there is an absence of global memory (conformance type 3). This system architecture will be referred to hereinafter as the moderately coupled distributed system.

The third model considers placing multiple processors or groups of multiple processors on a common network with store-and-forward and will be referred to hereinafter as a loosely coupled system (conformance type 4).

2.2.1 Multiprocessor Systems

In a multiprocessor system it is possible for the processors to share the same set of system resources such as memory, network interfaces, peripherals, etc., over a common memory bus or backplane. It is feasible for a global clock to be shared by all processors. Negligible execution overhead is exhibited when the clock resides on the same backplane as the processors. Having a global clock allows for a more consistent global state throughout the multiprocessor system. It is possible for the hardware to provide interprocessor interrupt generating facilities which may be used to synchronize and signal concurrent actions. It is also a trivial matter to communicate data by sharing globally addressable memory located on the same memory bus or backplane for a set of homogeneous processors [CHO89]. The architecture for a simple multiprocessor system with global memory is depicted in Figure 1.

Where the processors are heterogeneous (i.e., unlike in instruction set architecture and basic data type representations), the possibility could arise where analogous data are represented differently at the machine level. This would make the sharing of data a non-trivial concern. It is also possible for each of the processors to have access to its own private memory or to share memory with a subset of all processors connected to a common backplane. This complex multiprocessor architecture is depicted in Figure 2.

2.2.1.1 Futurebus+

The Navy's Next Generation Computer Resources Program (NGCR) has assembled a backplane working group with the task of developing a set of requirements for a backplane to be used on all future mission-critical computers. The resulting standard is known as Futurebus+, which is a revised and extended version of the original IEEE 896.1-1987 Futurebus standard [BOR90]. It is the intent of the NGCR to help make the Futurebus+ standard become a major commercial success. Should this happen, the effect would be the reduction in the amount of funding the government would need to expend on research and development of system backplanes.

Futurebus+ is a specification for a scalable architecture. Scalable pertains to the width of the data path, which for Futurebus+ may be 32, 64, 128, or 256

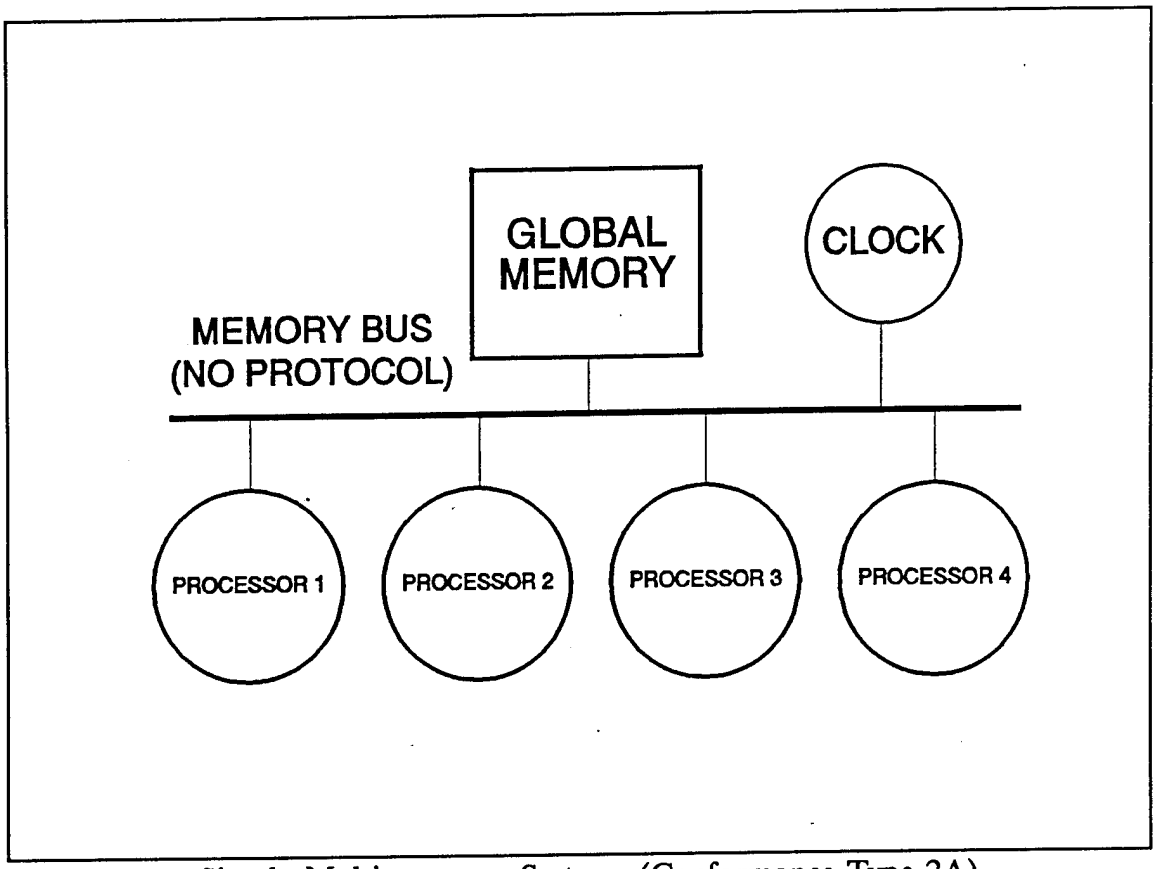

Figure 1 Simple Multiprocessor Systems (Conformance Type 2A)

bits wide. The scalability allows this bus, as the name would suggest, to address future performance requirements in Navy systems. It has been determined that the protocol throughput of this bus in the 32-bit mode in 1990 would be 100 Mbytes/second as opposed to the 256-bit mode in 1995, which would peak at 3.2 GBytes/second [BOR90].

Futurebus+ provides functionality that's not unlike a LAN. The functionality can be described in layers much like the OSI model. The layers of Futurebus+ are shown in Figure 3. A particular implementation of the layers of Futurebus+ is known as a "profile." The Navy plans on specifying a set of profiles to meet its requirements. At this point in time, four profiles have been identified, but only two have been defined. Profile A specifies a 64-bit data-bus width with a default width of 32-bits. Profile B specifies a 128-bit data-bus width with default widths of 64-bits and 32-bits. Profile C will specify the cable interconnection for communications between systems. Profile D will specify the Futurebus+ for personal computer applications. Presently, profile C and D have not been defined. The fact that the Navy has identified standard profiles does not prevent industry from developing proprietary profiles to meet their own requirements [AND90]. Draft 4.0 of the *physical layer* and profile specifications (P896.2) has been released. Draft 8.2 of the *logical layer* and profile specifications (P896.1R) has been released for the required 6-month review. Specifications for connector requirements (P1101.2), BTL interface circuits electrical characteristics (P1194.1),

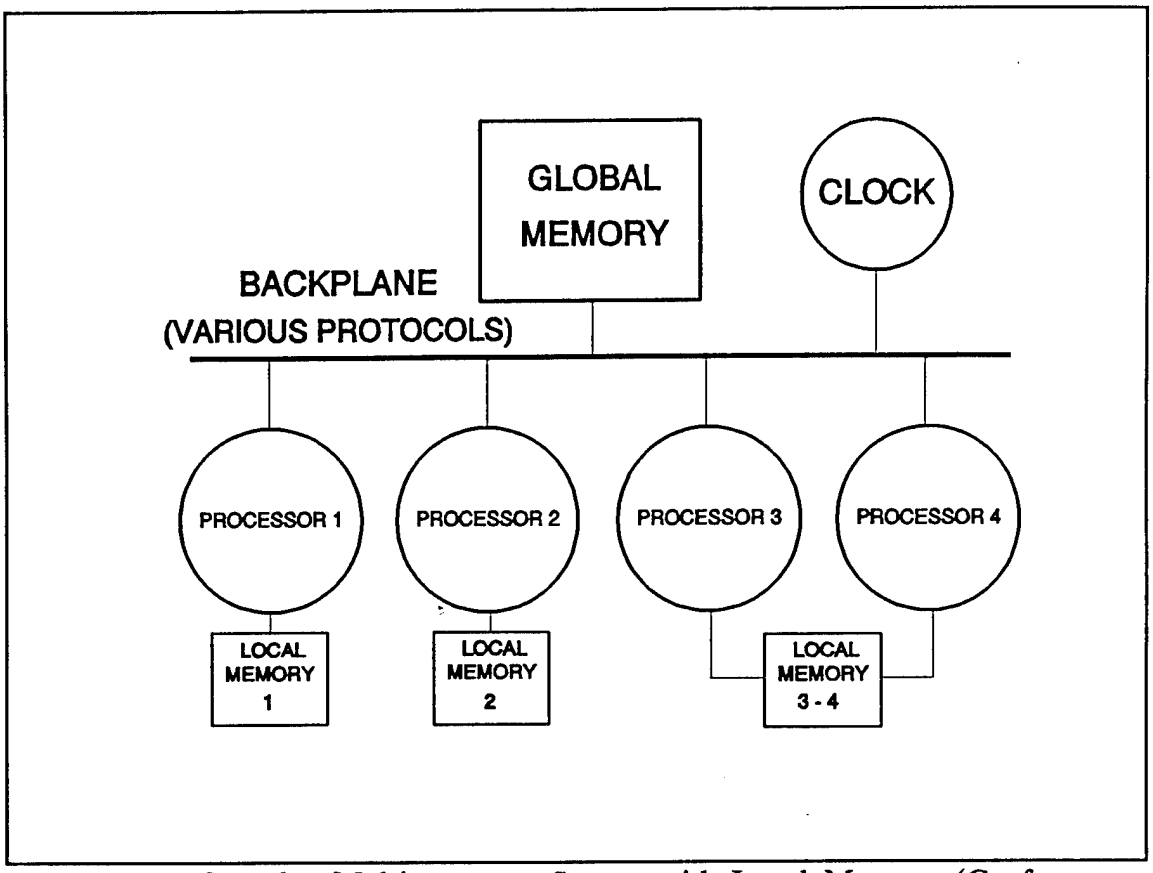

Figure 2 Complex Multiprocessor System with Local Memory (Conformance Type 2B)

and the VME-to-Futurebus+ bridge (P1014.1) have been released in draft form to date.

The Navy, under the NGCR program, has awarded three contracts to develop prototype versions of the Futurebus+ backplane supporting the following six processors:

1. MilVAX
2. MIPS R3000
3. Intel 80486
4. Motorola 68030
5. AMD 29000
6. Intel 88000.

Each of the three contractors has the job of building a single prototype backplane to support two of the above processors, with a commonly addressable memory board, a 1553 serial interface board, a naval tactical data system (NTDS) fast interface board, a survivable, adaptable, fiber-optic embedded network (SAFENET) I interface board, and a SAFENET II interface board. It can be determined that the Navy plans on supporting heterogeneous processors on a

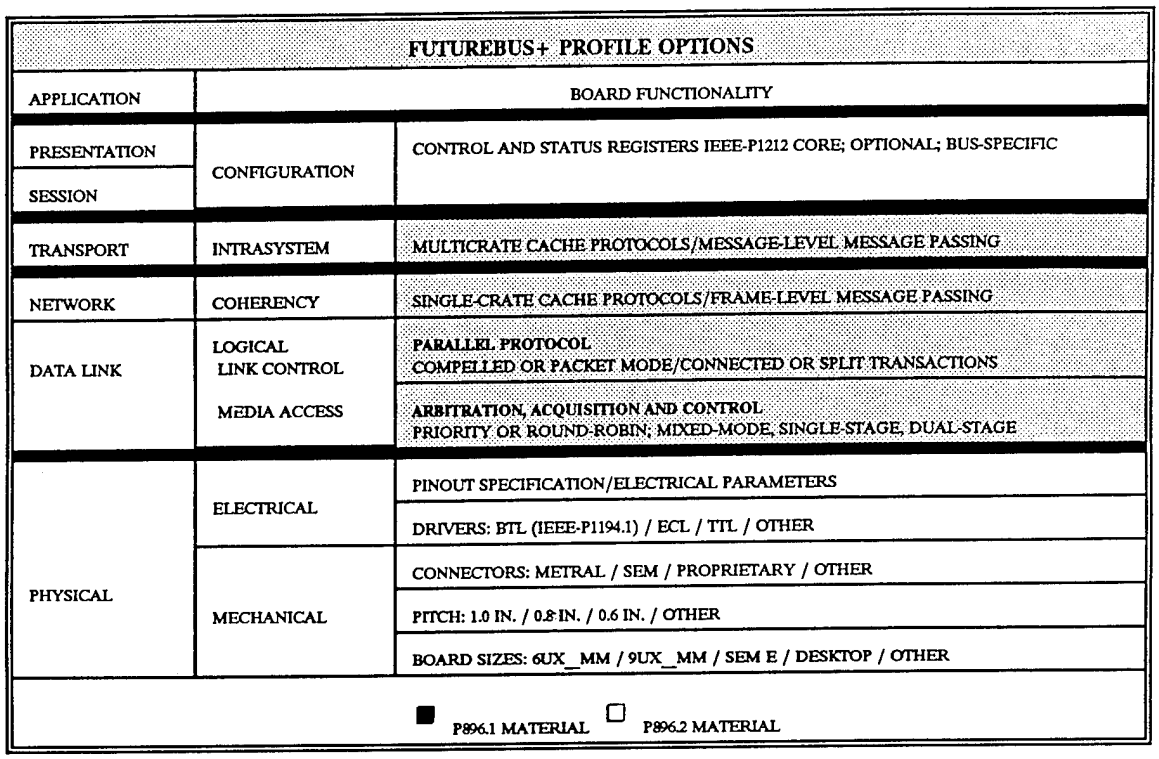

Figure 3 Futurebus+ Versus OSI Layers [AND90]

common backplane with multiple backplanes being connected through a SAFENET network.

2.2.1.2 Scalable Coherent Interface

The scalable coherent interface (SCI) is presently being established as a standard defined by IEEE P1596 [ALN90] [GUS90]. It defines an interface standard for very high performance multiprocessors. Like Futurebus+, SCI supports a cache-coherent-memory model. SCI is scalable to systems with up to 64K nodes and will supply a peak bandwidth per node of 1 gigabyte/second.

2.2.2 Moderately Coupled Multiprocessor Systems

With the advent of backplanes such as Futurebus+ and SCI which support complex protocols when compared to simple protocol memory buses, a new class of systems can be considered. In this class, multiple processors are connected to the same backplane without the availability of global memory. The processors do, however, have access to local memory or locally shared memory. Other resources such as storage device controllers, input/output devices, and network interface devices can, however, be globally shared on the backplane. This architecture is depicted in Figure 4 and is referred to as a moderately coupled system. Since the backplanes support a complex protocol like those supported on LAN's the system can be viewed as a single segment LAN where each node depicts a processor, storage device controller, input/output device, network interface device, etc.

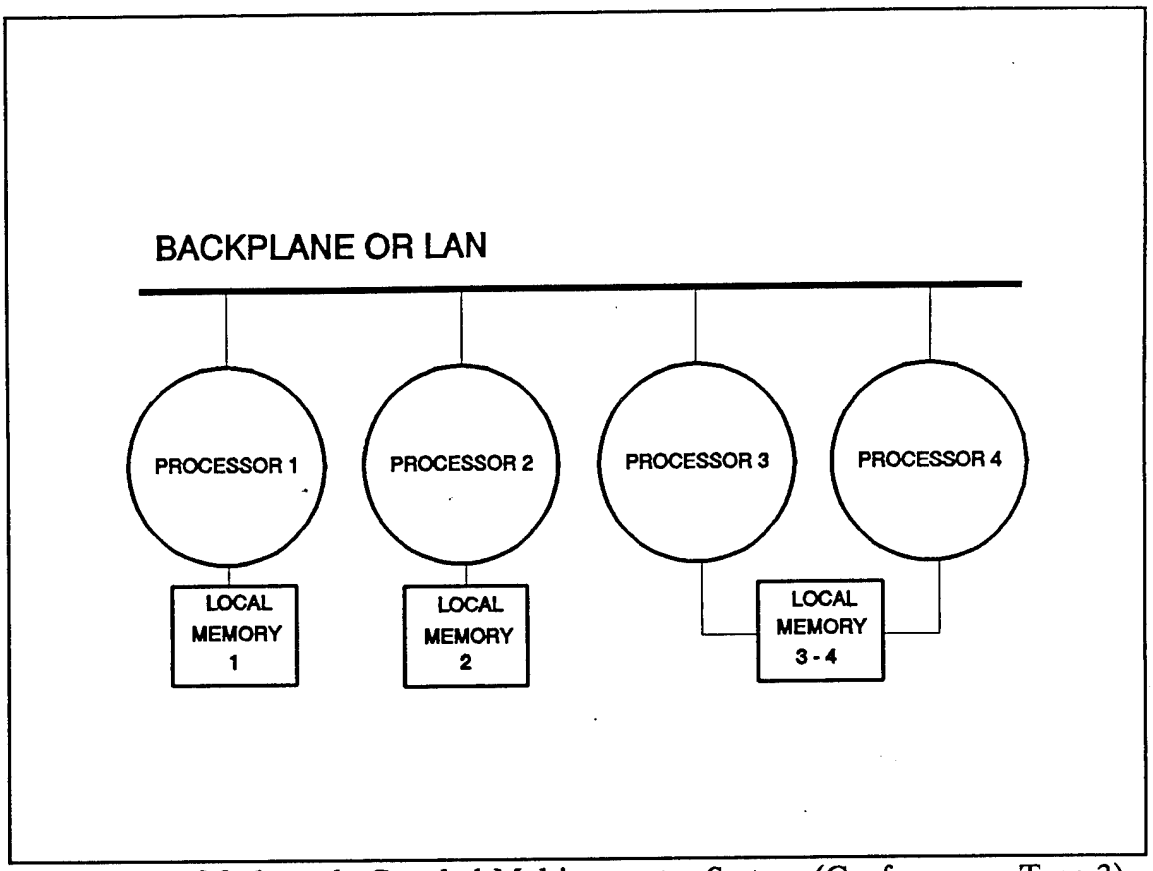

Figure 4 Moderately Coupled Multicomputer System (Conformance Type 3)

2.2.3 Loosely Coupled Systems

In the loosely coupled system model, processors do not share a common LAN or backplane. The common network becomes the only mechanism to share data and control. It is possible to view the network as comprised of multiple physical media connecting a number of nodes. Each node can be comprised of a multiprocessor system or a moderately coupled system. This architecture is depicted in Figure 5. Hereafter, when a loosely coupled system is referred to in this report, it denotes a system with communication between a processor on one node and a processor on another node through a network involving store-and-forward packet delivery mechanisms. Multisegment LANs, bridges, and routers are a few examples that fall into this category. Communications in loosely coupled systems suffer much greater delays (because of the store-and-forward nature) than the moderate delays of a single (moderately coupled) LAN or the minimal delays of a tightly coupled multiprocessor.

2.2.3.1 Network Standards and Protocols

Many different standard network protocols exist that provide a wide range of functions necessary to synchronize actions and communicate data. The DoD has mandated the transmission control protocol/internet protocol (TCP/IP) network protocol as the *de jure* standard for their networking needs [STA88].

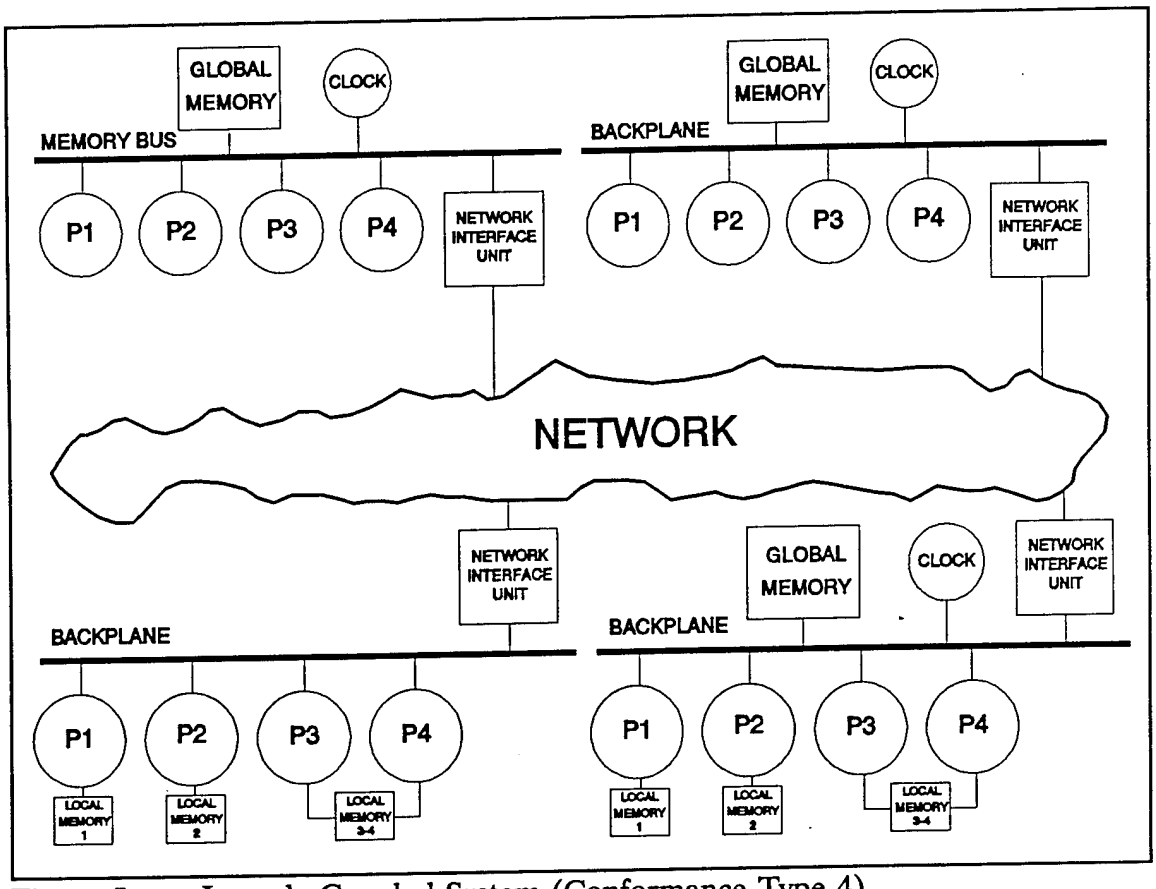

Figure 5 Loosely Coupled System (Conformance Type 4)

Some network standards were developed by a particular company and for the most part remain proprietary, such as DECnet [STE88], but may become *de facto* standards within organizations. The future protocol plans of the DoD are to adopt the International Standards Organization (ISO) Open Systems Interconnection (OSI) reference model. From the outset, the OSI model was designed to become a standard [STA871] [TAN88].

The key advantages of standards are [STA872]:

- They assure that there will be a large market for a particular piece of equipment or software.
- They allow products from multiple vendors to communicate, giving the purchaser more flexibility in equipment or software selection and use.

The tendency to freeze technology is the principal disadvantage of standards. Much effort has gone into developing these standards to provide a mechanism that will communicate between systems. Thus, the standards are worthy of further study in the following sections. The OSI reference model is described first followed by TCP/IP, DECnet, and SAFENET. The latter three are characterized in relation to the OSI model.

2.2.3.1.1 OSI Reference Model

The development of a standard by ISO from initial proposal to final publication involves a seven-step process. This process has built-in delays which allows ample time for review and comment of the proposed standard by a wide audience. This helps to insure that the resulting standard meets the main goal of the ISO, that it will be acceptable to as many countries as possible. The seven steps are briefly described below [STA871].

1. A new work item is assigned to the appropriate technical committee (TC), and within that TC, to the appropriate working group (WG). The WG prepares the technical specifications for the proposed standard and publishes these as a draft proposal (DP). This DP is circulated among interested members for balloting and technical comment. At least 3 months is allowed, and there may be iterations. When there is *substantial support*, the DP is sent to the administrative arm of ISO, known as the central secretariat.
2. The DP is registered at the central secretariat within 2 months of final approval by the TC.
3. The central secretariat edits the document to ensure conformity with ISO practices; no technical changes are made. The edited document is then issued as a draft international standard (DIS).
4. The DIS is circulated for a 6-month balloting period. For approval, the DIS must receive a majority approval by the TC members and 75 percent approval of all voting members. Revisions may occur to resolve any negative vote. If more than two negative votes remain, it is unlikely that the DIS will be published as an international standard (IS).
5. The approved DIS and revision are returned within 3 months to the central secretariat for submission to the ISO council, which acts as the board of directors of ISO.
6. The DIS is accepted by the council as an IS.
7. The IS is published by ISO.

The open systems interconnection reference model was one such standard developed by the ISO. The OSI model defines seven layers [STA871] [TAN88], which are depicted in Figure 6.

The *physical layer* is concerned with the physical (electrical, optical, etc.) transmission of raw bits over a communication channel.

The *data link layer* is concerned with making the physical layer appear free of transmission errors to the network layer. The key function of this layer is to provide the mechanism by which frames are transmitted from source to destination. This layer is composed of two sublayers in the case of a LAN. The *medium access control* (MAC) sublayer defines the lower part of this layer and is concerned with determining who gets access to a communications channel when

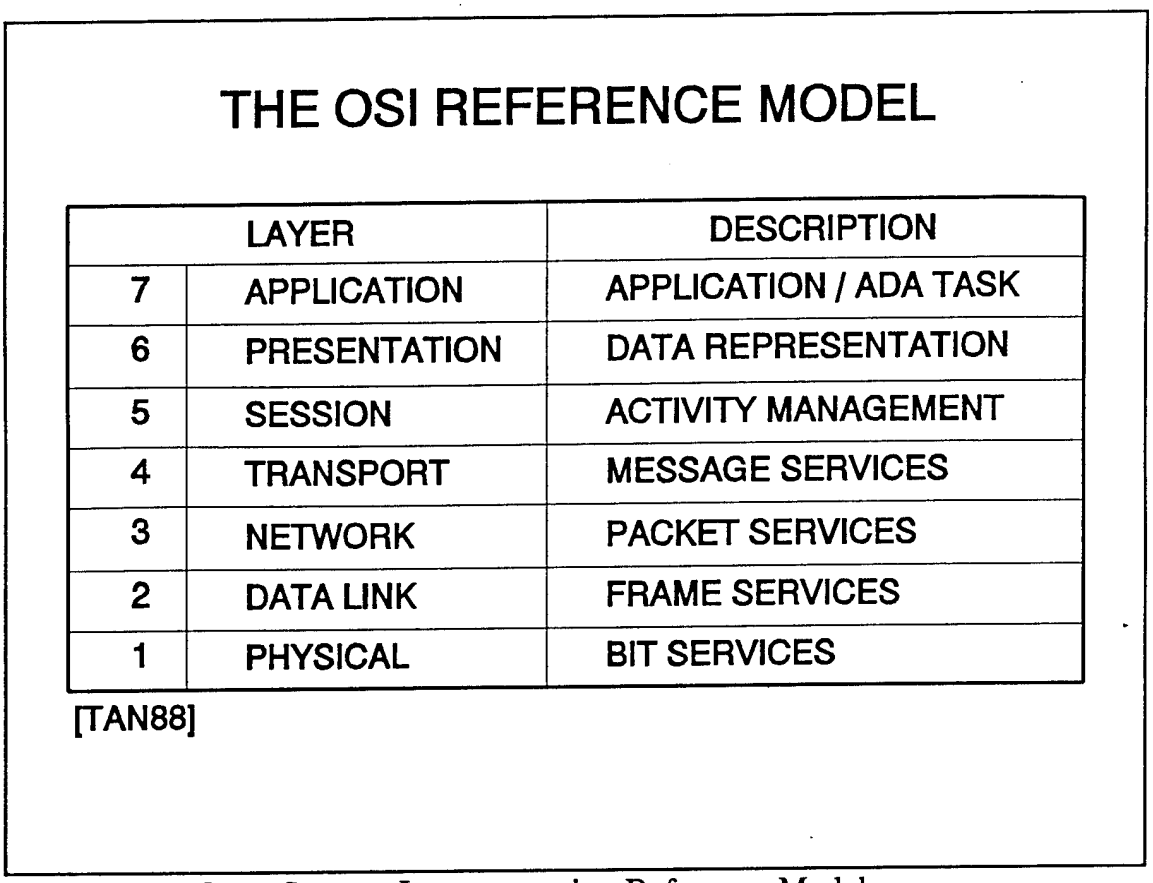

Figure 6 Open Systems Interconnection Reference Model

there is competition for it [TAN88]. The *logical link control* sublayer defines the upper part of this layer and is concerned with providing a uniform interface to the network layer regardless of the MAC protocol used.

The *network layer* is concerned with controlling the operation of subnets. The routing of packets from source to destination, congestion, and flow control are the key issues addressed in this layer.

The *transport layer* is concerned with reliable end-to-end transport of messages. The key issues addressed in this layer are the disassembling of messages into packets to send to the network layer. This layer also assures that packets arrive at the destination and are reassembled in the correct order to form the original message.

The *session layer* is concerned with activity management between peer entities. The key issues addressed in this layer are the use of the connection established between the peer entities such as, token management and synchronization.

The *presentation layer* is concerned with making the underlying differences in data representation appear invisible to the layer above. The key issues addressed in this layer involve insulating the application from the different ways a

system may represent data such as strings, integers, floating point numbers, and so on.

The *application layer* is concerned with high level functions commonly used when communicating between nodes on the network such as electronic mail, virtual terminal, file transfer, and the rest of the application universe.

2.2.3.1.2 TCP/IP

The DoD was faced with the necessity of providing communication between an ever growing population of heterogeneous systems. The DOD determined the need to fulfill the two following requirements [STA88].

- End systems must share a common set of communication protocols so they can interoperate.
- The suite of protocols used for this purpose must support an internetworking capability in a mixed-network environment.

In response to this need and through the evolution of ARPANET, the Defense Communications Agency (DCA) issued the set of military standard protocols listed in Table II. The ARPANET was the creation of ARPA (now DARPA), the (Defense) Advanced Research Projects Agency of the U.S. Department of Defense.

Table II DoD Military Standard Protocols [STA88]

Number	Title	Description
MIL-STD-1777	Internet Protocol (IP)	A connectionless service for end systems to communicate across one or more networks. Does not assume the networks to be reliable.
MIL-STD-1778	Transmission Control Protocol (TCP)	A reliable end-to-end data transfer service. Equivalent to the ISO Class 4 transport protocol.
MIL-STD-1780	File Transport Protocol (FTP)	A simple application for transfer of ASCII, EBCDIC, and binary files.
MIL-STD-1781	Simple Mail Transfer Protocol (SMTP)	A simple electronic mail facility.
MIL-STD-1782	TELNET Protocol	A simple scroll-mode terminal capability

By mandating the use of a standard set of protocols, the DoD was able to prevent smaller organizational units within from adopting varying and potentially non-compatible existing protocols. The layers are not unlike the layers of the ISO model.

The *physical layer* provides the physical media by which raw data are exchanged between hosts.

The *IMP-IMP layer* provides the protocol that is used to transfer packets from one information message processor (IMP) to another.

The *source to destination IMP layer* provides the mechanism that verifies correct reception of packets at the destination IMP.

The *host-to-host layer* provides the mechanism by which data are transferred reliably and in the same order as it was sent from one host to another.

The *telnet, smtp, ftp layer* provides protocols for virtual terminal, simple mail transfer, and file transfer.

Finally, the *process layer* provides the protocols necessary to support a variety of applications.

A comparison of the DoD networking layers and the ISO layers can be seen in Figure 7. Even though this protocol was mandated by DoD, many companies in private industry have adopted the protocol to solve the same problems identified by DoD. It is also becoming more commonplace for computer vendors to produce heterogeneous lines of computer systems such as Digital Equipment Corporation's (DEC) VAX systems and their newer DEC systems. The former is based on the VAX architecture and the latter is based on the MIPS R3000 RISC architecture.

2.2.3.1.3 DECnet

DECnet, a product of Digital Equipment Corporation, employs the Digital Network Architecture (DNA) model. Phase V of the DNA model is an evolutionary step from the previous version of the DNA model known as Phase IV. The Phase V model integrates the OSI model with the Phase IV model. The lower four layers (i.e., the physical, data link, network, and transport layers) are, for the most part, identical to the OSI model in terms of functionality. Two

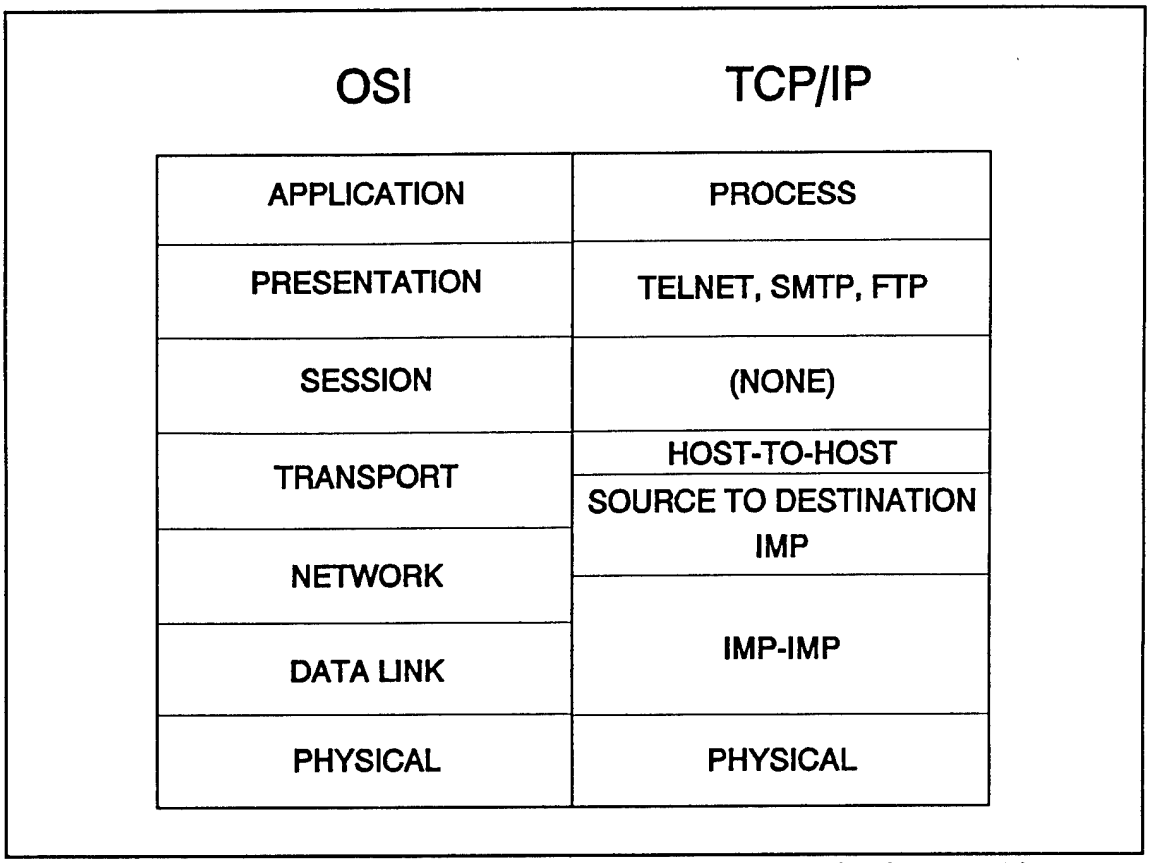

Figure 7 A Comparison of the OSI and DOD Communications Architecture

different protocol stacks reside above the transport layer in the DNA model as shown in Figure 8.

The DNA *physical layer* has always been based on available standards that include EIA RS-232-D, which includes the corresponding ISO standards and the International Telegraph and Telephone Consultative Committee (CCITT) recommendations, and the Ethernet standards as reflected in IEEE 802.3 and ISO 8802-3.

The DNA *data link* layer uses the high-level data link control (HDLC) protocol. This protocol is described by ISO 4335 and ISO 7809. In the case of local area networks, the logical link control protocol of IEEE 802.2 and ISO 8802-2 are employed.

The DNA *network layer* uses ISO 8473 for data transfer. This protocol is the ISO protocol for providing the connectionless-mode network service (CLNS). ISO 9542 is used to facilitate the exchange of routing information between end systems and routers. This protocol is the ISO end system to intermediate system (ES/IS) routing protocol. ISO 8208, X.25 packet layer protocol, and the mapping defined in ISO 8878 are used to provide the connection-mode network service (CONS). ISO 8348 addendum 2 defines the addresses used by the DNA network layer.

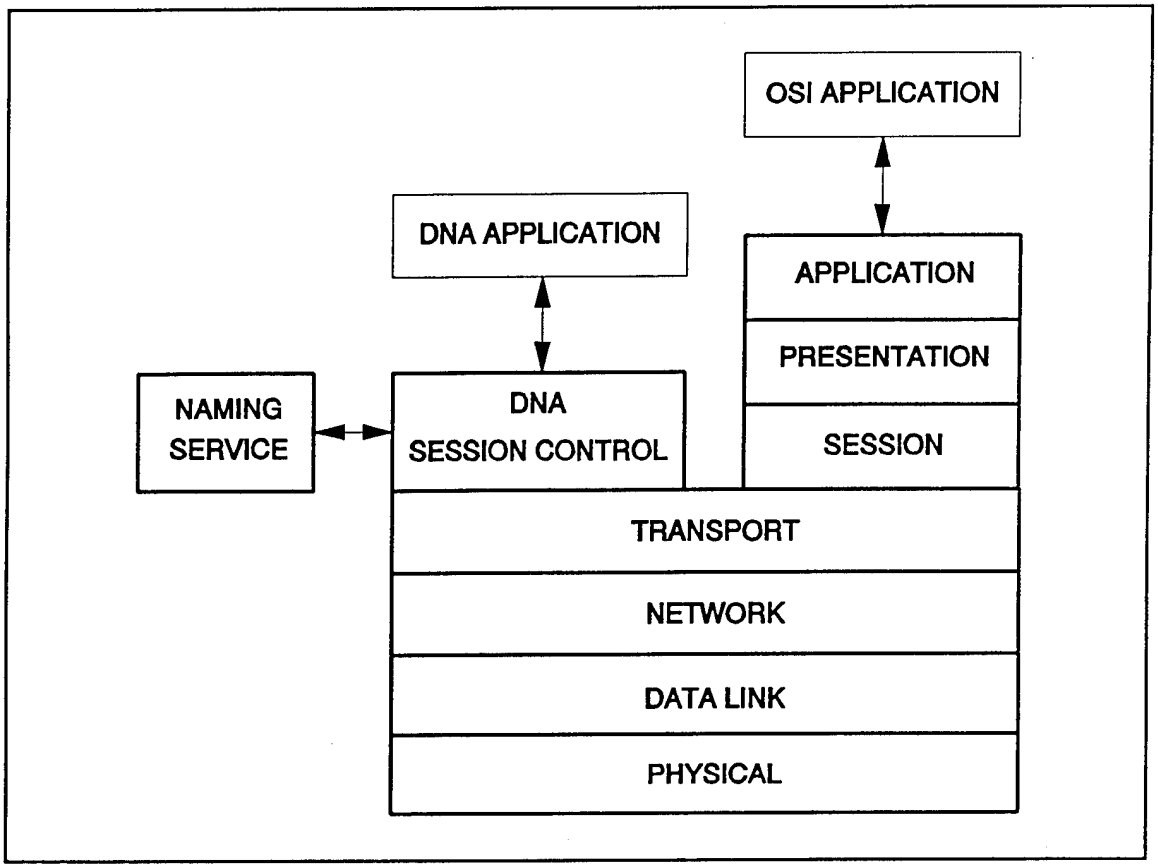

Figure 8 The Layers of DNA [DEC87]

The DNA *transport layer* employs ISO 8073, which includes Class 4 of the ISO transport protocol. Classes 0 and 2 are provided for use over the CONS. ISO 8072 provides the ISO transport service to a DNA application.

The DNA *session control layer* provides the logical links or connections used by the application. This protocol is proprietary and is used to provide upward compatibility to applications being migrated from DECnet Phase IV to DECnet Phase V. All Phase IV DNA applications will continue to run on Phase V without modification.

The *DNA application layer* supports a number of application protocols. The data access protocol (DAP) is provided for accessing and transferring files in a heterogeneous DECnet network. The network virtual terminal (NVT) service is provided to offer standard terminal services and device independence. The Mail-11 protocol is used to provide personal electronic mail capability. A rich set of applications protocols is provided for communicating with systems that conform to IBM's systems network architecture (SNA). VMS services for MS-DOS is a VAX-based remote file server that gives applications running under Microsoft's MS-DOS operating system access to files on a DEC computer system. The Digital time service defines an architecture for providing and maintaining correct time in a distributed system. VAX notes computer conferencing allows users of a DNA network to participate in round-table discussions. VAX system performance

monitor (VAX SPM[tm]) allows a member of a VAXCluster[tm] to communicate with all other members of the cluster. VAX VTX[tm] is an application that provides a computer-based electronic retrieval system, which uses a hierarchical arrangement of information and is video terminal based. VAX distributed queuing service (DQS) allows users to queue print jobs on remote systems connected on the same DNA network. The remote system manager (RSM) is used to aid in the management of distributed DEC VMS and ULTRIX systems.

The DNA OSI *session layer* is not yet specified. When they are specified by the OSI, it is anticipated that the standard will be implemented in an update to DECnet Phase V.

The DNA OSI *presentation layer* is not yet specified. When they are specified by the OSI, it is anticipated that the standard will be implemented in an update to DECnet Phase V.

The DNA OSI *application layer* contains many application service elements (ASE) for which many are currently under development. The association control service element (ACSE) is used to establish associations between applications to be used for the purposes of exchanging information. File transfer, access, and management (FTAM) services define services that facilitate file transfer, read, write, modify, creation, deletion, and attribute modification over the network. Virtual terminal (VT) services are similar to the facilities provided by the previously described DNA NVT service.

2.2.3.1.4 SAFENET

The process of developing standards to meet a wide variety of user needs is a very laborious and time-consuming task. The Navy has determined that the development of the ISO OSI reference model is taking too much time. Therefore, the Navy has decided to adopt standards based on the OSI model called SAFENET I and SAFENET II. The SAFENET (survivable adaptable fiber optic embedded network) standards have been developed by committee with industry and Navy participation. This group is known as the SAFENET working group. SAFENET is a set of standards (a subset of the OSI standard) including additional implementation agreements required to ensure interoperability [GRE89]. SAFENET I is to be specified in MIL Handbook MCCR 0034 (draft) and was delivered to its sponsor, SPAWAR 324, in January 1990. SAFENET II is to be specified in MIL Handbook MCCR 0035 (draft) and was delivered to the same sponsor in January 1991.

The seven layers of the ISO model can be related to the SAFENET layers. The ISO layers are specified at too high a level to allow a complete implementation at this point in time. The DNA model could not specify the session and the presentation layers for this reason. The SAFENET layers, however, are specified in sufficient detail to allow full implementation at this time. Figure 9 depicts the protocol profile for SAFENET I [SAF901] and

Figure 10 depicts the protocol profile for SAFENET II [SAF902]. The mapping between the seven layers of the ISO model and the corresponding SAFENET layers can be seen in Figure 11.

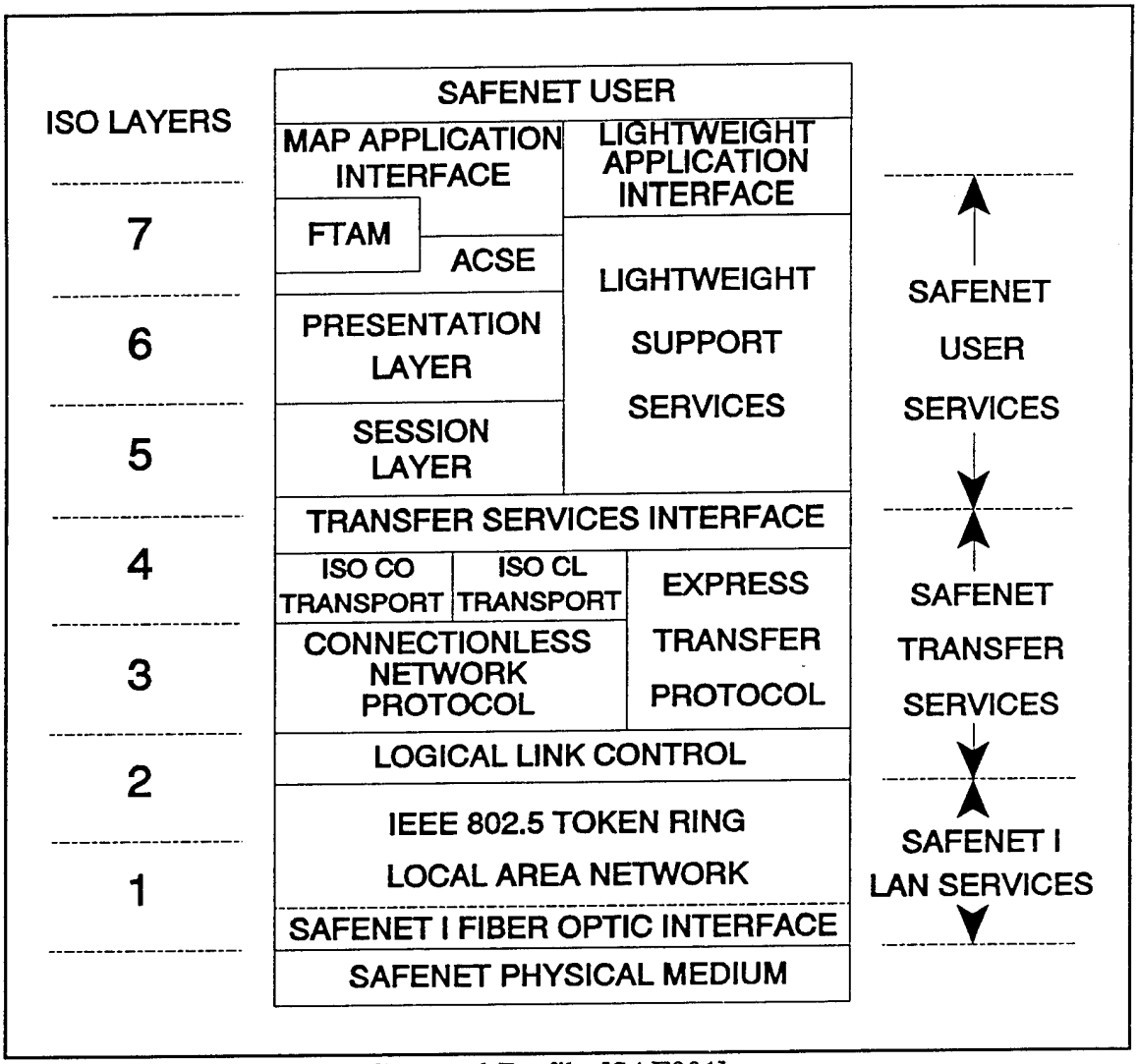

Figure 9 SAFENET I Protocol Profile [SAF901]

SAFENET specifies three protocol suites, the ISO suite, the lightweight suite, and the combined suite. The ISO suite is used to maximize interoperability when dealing with heterogenous systems on the same network. It also provides a rich set of application layer services. In real-time embedded systems, however, this extra functionality usually indicates high and unwanted overhead. Therefore, SAFENET also provides a lightweight protocol suite to remove the unwanted overhead, yet includes enough services to provide the necessary functionality to build distributed systems with minimal communication latency. The combined suite incorporates both the ISO suite and the lightweight suite to provide all capabilities in one suite.

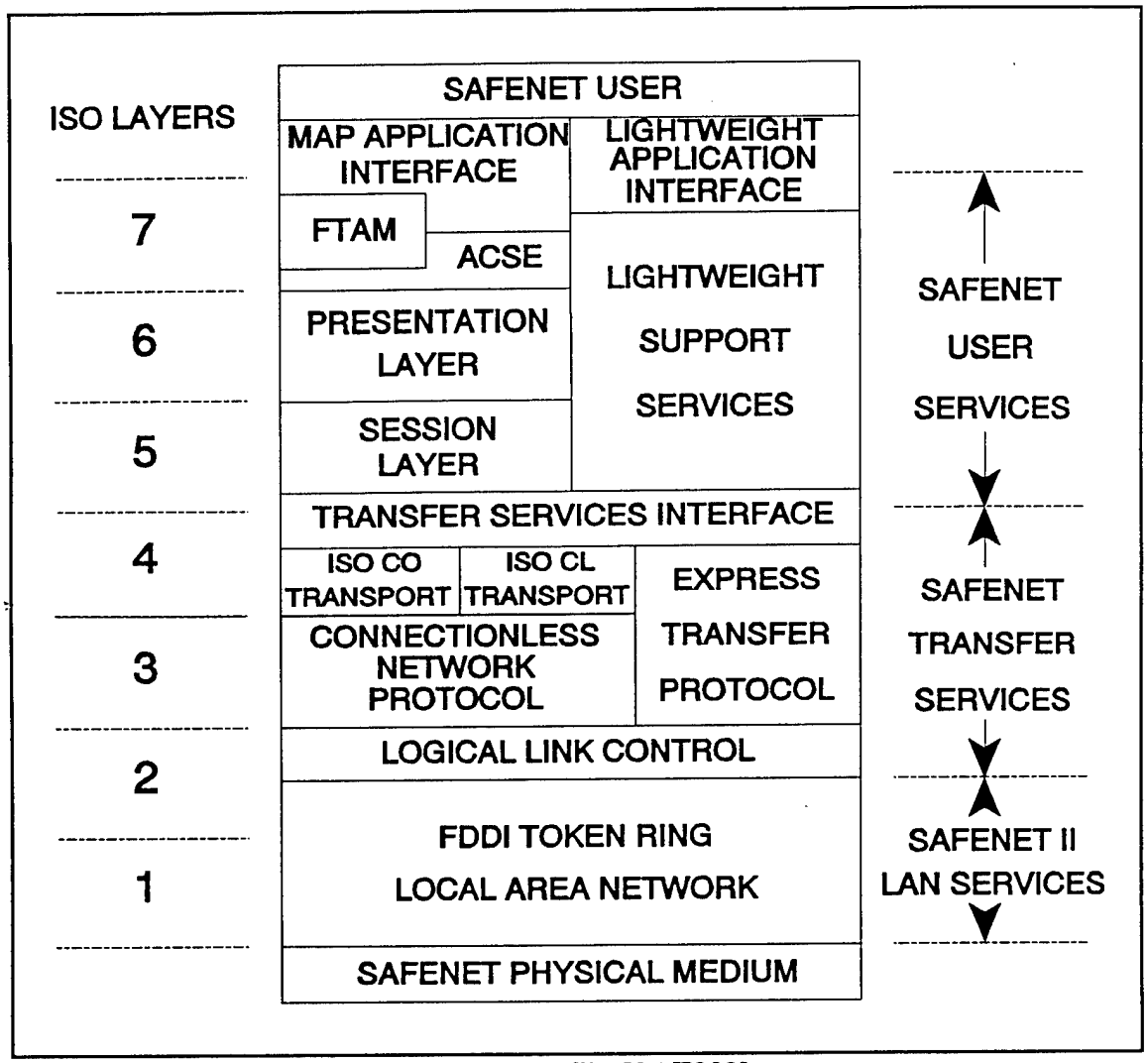

Figure 10 SAFENET II Protocol Profile [SAF902]

The *application process* portion of Figure 11 refers to the Navy's commitment to ensure support of applications written in Ada. This does not preclude applications written in other languages, only that the use of Ada is a firm requirement of DoD and must be addressed. This is not a formal layer of the SAFENET definition. It is shown here to clarify the Navy's view of Ada on distributed systems.

The *operating system* portion of Figure 11 refers to a current effort being pursued by a subcommittee of the NGCR program. The definition of this layer is crucial in that it is here where design decisions concerning the distribution methodology of Ada is implemented. This is not a formal layer of the SAFENET specification. It is an approach which some segments of the Navy are considering for implementing Ada on loosely coupled systems.

The SAFENET ISO protocol suite *user services* encompass the ISO application, presentation, and most of the session layers. The protocol is based on

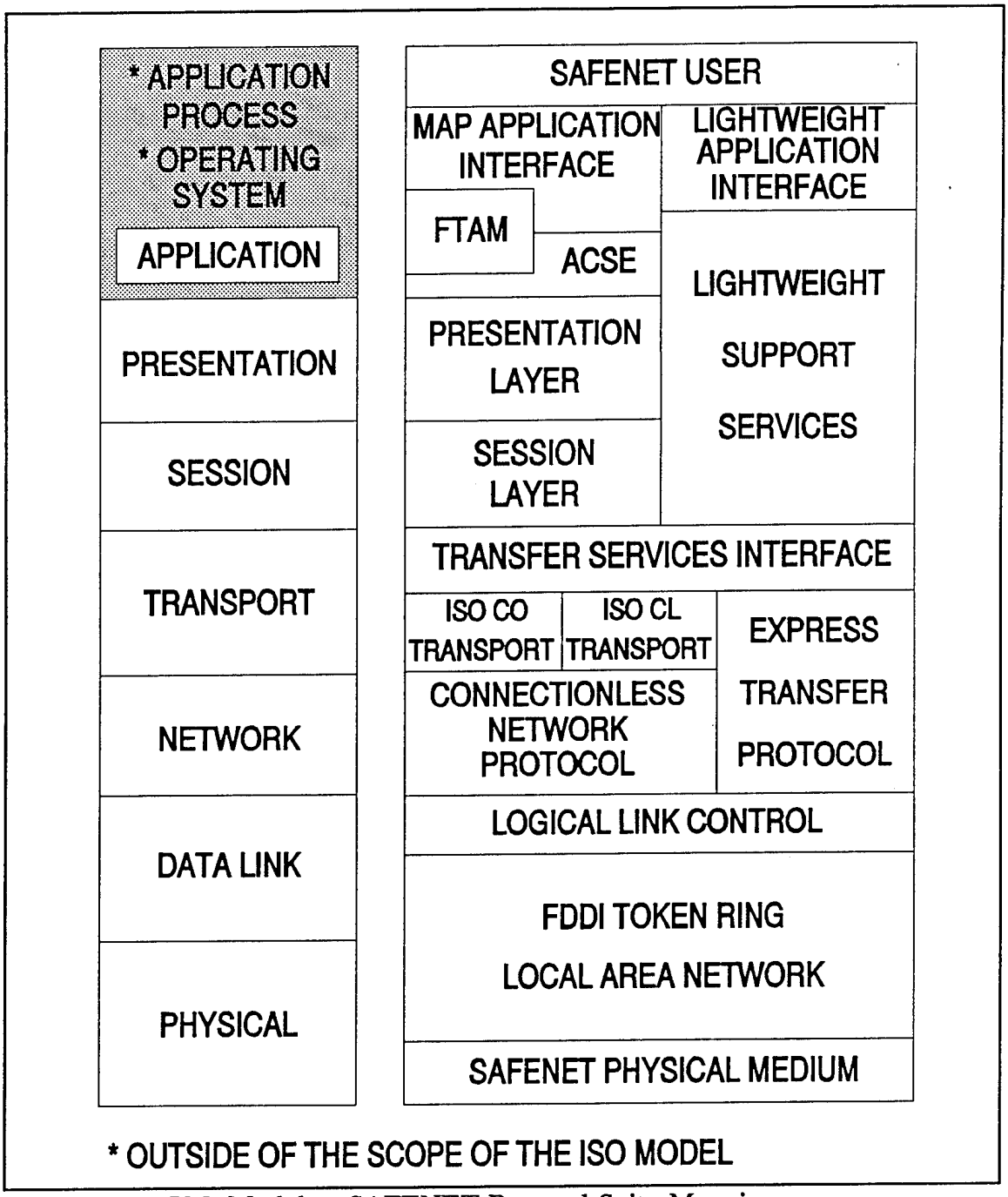

Figure 11 ISO Model to SAFENET Protocol Suite Mapping

the manufacturing automation protocol (MAP), version 3.0. MAP defines many of the upper layer OSI services that are not yet fully defined. In general, MAP 3.0 defines protocols for association establishment, data transfer, file handling, and directory services. MAP 3.0 definitions include the following:

- FTAM, the services used for file handling
- directory services
- ACSE, the services used for association establishment

- *presentation layer* protocol
- *session layer* protocol
- Private communications application interface specification
- Application interface support functions
- Connection management interface specification
- FTAM application interface specification

The lightweight application interface requirements are briefly, if not inadequately, described in [SAF901] and [SAF902] section 6.2.

The SAFENET ISO protocol suite *transfer services* encompass the remainder of the ISO session, the transport, network, and part of the data link layers. The connection-oriented transport protocol is based on ISO 8073, transport protocol class 4 (TP4). The SAFENET lightweight protocol suite transfer services is implemented in accordance with the express transfer protocol (XTP) definition [XTP90]. The ISO connectionless (CL) transport protocol is implemented in accordance with ISO 8602. The CL transport protocol is applicable to single segment LANs only. Multicast transfer has not been defined to date but is a requirement by the SAFENET user interface. The ISO connectionless-mode network protocol is implemented in accordance with ISO 8473. Network layer address formats are implemented in accordance with addendum 2 of ISO 8348. ISO end system/intermediate system (ES/IS) intra-domain routing protocol is used. The IEEE logical link control (LLC) protocol is employed in accordance with IEEE 802.2. The sub-network access protocol (SNAP) is employed in accordance with IEEE 802.1A.

The SAFENET *local area network services* encompass the remainder of the ISO data link, and the physical layers. These services differ for SAFENET I and SAFENET II. The token-ring medium access control (MAC) protocol specified by IEEE 802.5 with options (addendum C) plus ring hop is used for SAFENET I. The fiber distributed data interface (FDDI) MAC protocol specified by ISO 9314-2 is used for SAFENET II.

2.3 Distributed System Software Requirements

Services are required to enable software to utilize the full processing potential of distributed systems. The service requirements of a system are dependent on the architecture of the distributed system. The distributed system hardware architecture conformance types were depicted in Table I.

Specific service primitives are not necessarily appropriate over all architecture conformance types. By way of illustration, consider the two following general classes of service:

1. Synchronization
2. Message transfer.

For synchronization, the semaphore, monitor, Ada rendezvous, and activity management services are considered. For message transfer, read/write primitives, datagrams, acknowledged datagrams, reliable messages, and broadcast datagrams are considered. Table III characterizes the appropriateness of these various services with respect to system architecture.

Table III Distributed Software Service Requirements

	Conformance Type				
	0	1	2	3	4
Synchronization:					
Semaphore	X	X	X		
Monitor		X	X	X	X
Rendezvous		X	X		
Activity				X	X
Message Transfer:					
Read/Write	X	X	X		
Datagram				X	X
Acknowledged Datagram				X*	
Reliable Message					X
Broadcast				X*	

* Well suited for this mechanism

It can be seen by analyzing Table III that some services are inappropriate for some architectures. Of particular interest is the Ada rendezvous which is appropriate for uniprocessor and multiprocessor systems with shared memory. Conformance level 3 very effectively supports acknowledged datagrams and broadcast. When dealing with systems that involve store-and-forward message transfer such as multi-segmented LANs or WANs (wide area networks), other mechanisms not directly supported by Ada runtimes are more appropriate, such as unacknowledged datagrams and reliable messages (virtual circuits). Another mechanism that is very useful and often required in some distributed systems is the broadcast datagram. This concept is not directly supported by Ada or most WAN protocols.

3 TYPES OF DISTRIBUTED SYSTEMS WITH ADA

In light of the discussions in section 2 covering the architectures of distributed systems, this section will discuss the similarities, variations, and characteristics of distributed systems with Ada from the perspective of software.

3.1 Distributed Ada System Models

There are basically two ways in which an Ada program is distributed across a number of processors [RAB90]. The first way is to conceptually place a single Ada application across the entire suite of processors. In this model, the unit of separation may be the Ada task, package, subprogram, or any unit depending on the desired distribution complexity. The Ada programming language reference manual, ANSI/MIL-STD-1815A states in a note that "parallel tasks ... may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor." [LRM83, p. 9-1]. The unit of parallel execution may be a task, but is in fact, arbitrary and unique to each implementation of Ada. The high-level constructs offered in the Ada language must be partitioned onto physical processors either through automatic selection or selection by the systems programmer. This step can be performed independently of normal software development, thus allowing the language to be fully utilized in its support for good software engineering practices. Conceptually, the main program unit serves as the root task in the tree of tasks that may compose the context of the distributed application. The details of the mechanisms used for task synchronization and data communication are integral parts of the Ada runtime kernel and are invisible to the applications programmer.

The second model used to distribute Ada programs is to place one or more Ada programs on each processor. An external mechanism is employed to provide program synchronization and data communication. The Ada runtime generally does not support any of the communication mechanisms directly. Mechanisms are provided externally to the language through packages and subprograms. The job of providing interprogram communication of data and synchronization is the responsibility of the system designer. The mechanisms used for program synchronization and data communication are not integral parts of the Ada runtime and are not invisible to the applications programmer.

3.1.1 Ada on Multiprocessor Systems

Multiprocessor systems have been described as providing a number of processors attached to a common backplane or memory bus. It is usually the case where all of the processors are alike (homogeneous), that is, identical in architecture and instruction set. Global memory can be attached to the common backplane and accessed by all of the processors. It is possible for all of the processors to share data by placing the data in a mutually agreed upon address in global memory. This type of system lends itself well to incorporating the techniques described in the first model of distributed Ada programs [CHO89]

[LIN89] [ELL89] [BAM88] [VAN89] [BAK86]. With minimal changes required to be made to the Ada runtime, which normally executes parallel tasks in interleaved fashion on a single processor, the runtime can now support execution of tasks across the entire set of processors. The following system features are required to minimize, if not eliminate, compiler modifications and minimize runtime modifications [LIN89]:

- Any data shared by tasks running on separate processors must be placed in globally accessible memory.
- Transparent memory bus arbitration is also required to avoid modifying the Ada compiler.
- An atomic test-and-set instruction, or equivalent, to be used for mutual exclusion.
- A mechanism to allow processors to interrupt each other. This requirement is used to implement cross processor rendezvous.
- A mechanism to identify and distinguish between processors. This requirement is used to organize data structures specific to a particular processor.
- A set of timers that all processors can read and set. This requirement simplifies the implementation of the delay statement and package CALENDAR.

The above requirements are essential to implement the Texas Instruments Multiple Processor Ada Runtime (TI MPAR) system.

Selecting the processor on which to execute a task may be determined at program bind time [JHA89] or automatically and dynamically [LIN89] by the kernel of the Ada runtime. The latter has the advantage of possibly incorporating a dynamic load-balancing algorithm to insure peak execution performance regardless of changing execution conditions.

In the case of the Navy's Futurebus+ heterogeneous multiple processor configurations, a number of complications are introduced. The processors are not alike, therefore, it is more difficult to develop an Ada compiler and supporting runtimes across the entire suite of processors. In fact, there are a limited number of compiler vendors that produce a single compiler, which targets a number of different processors. In those cases where a single compiler does target a number of different processors, the runtimes for each targeted processor are unique and do not incorporate the necessary mechanisms to interoperate, such as support for heterogeneous cross processor rendezvous.

The problem of data representation at the machine level complicates matters. One cannot simply share global data in commonly addressable memory without consideration of the cooperating target processors. Byte ordering between processors may differ. For instance, the VAX is a little endian processor [TAN88] and the other five processors being prototyped by the Navy are

configured as big endian. If the intent is to have data shared in commonly addressable memory, the data producing processor, or the data consuming processor, or both would have the responsibility of converting the data. Mechanisms such as the Sun external data representation (XDR), the ISO model's abstract syntax notation 1 (ASN.1) [TAN88], or a unique conversion process, could be used to convert the data from one processor's representation to the other processor's representation, or to a mutually agreed upon neutral representation. This would require shadowing the global data with an analogous data structure in the agreed upon representation, thus, multiplying the memory requirement for sharing data by the number of processors requiring different representations. To make all of these presentation issues invisible to the applications programmer would require extensive modifications to the Ada runtime. Along with the invisibility would come questions from the systems designer of what effect the associated overhead would have on performance.

When building a large embedded distributed system, the system designer does not want to consume time developing or modifying the Ada compiler or runtime. It is the desire of the system designer to implement the application without distractions [VAN89]. It has been suggested that a multiprocessing support kernel can be built external to the Ada runtime. This would provide a timely solution that is more economical, and thus feasible as a short-term alternative.

3.1.2 Ada on Moderately Coupled Systems

Moderately coupled systems do not have shared memory. This makes it more restrictive in the use of Ada tasking in that a programming discipline must be enacted to disallow the use of data objects shared by tasks that may run on separate processors. Enforcement of such a programming system would, in effect, allow a subset of Ada to exist. This would be an undesirable effect and invalidate the Ada compiler being used. Mechanisms could be employed to share data and synchronize actions through communications protocols over the backplane or single segment LAN. These mechanisms are similar to those that would be necessary to implement on a loosely coupled system and will be discussed in more detail in the section 3.1.3.

An Ada runtime was modified for the Hypercube (homogeneous moderately coupled) system [CLA89]. All communication between tasks was accomplished via a message-based approach utilizing the system's existing store-and-forward communication facilities. The allowed units of distribution were library packages and library subprograms. Tasks declared within a library package were also allowed to be distributed. Nested tasks are restricted to execute on the same processor as the parent task. It was determined that the limitations on the allowable distributable units greatly effects the requirements (overhead) of the Ada runtime system (i.e., less restrictions implies greater runtime overhead).

3.1.3 Ada on Loosely Coupled Systems

Loosely coupled systems have been described as containing many processors that do not share global memory nor a single interconnect medium (i.e., a common backplane bus or single segment LAN). Moreover, they employ store-and-forward with the attendant delays. The mechanism employed to share data and to synchronize actions is implemented through communications protocols.

The Navy's NGCR effort has clearly identified SAFENET I and SAFENET II as the networking standard to be used in future systems. SAFENET is applicable to all shipboard, aircraft, and landbased interconnection problems, with either OSI standard compatibility or real-time data transfer requirements [SAF901] [SAF902].

For most Navy applications, the communication protocols are implemented on multiple LANs. It is increasingly becoming the case where the processing units connected to the same LAN are not alike. Commonly addressable memory is not available to the cooperating processors. This type system does not lend itself well to incorporating the techniques described in the first model of distributed Ada programs.

Some additional general considerations for Ada compiler and runtime modifications to support this first model of a monolithic Ada program across heterogeneous processors are summarized in Table IV. An Ada development system would need to be evolved to support program development that targets all heterogeneous processors of interest. There are two basic approaches that can be taken to accomplish this. In the first approach, a single Ada compiler can be developed to support multiple back-ends and runtime systems that target each processor. In the second approach, separate existing Ada compilers, that are hosted on different processors and separately target all processors of interest, could be modified to support a single program library that all compilers store information and object code in. Tools would have to be developed that can extract the appropriate object modules from the program library to link and download executable modules to all targeted processors. The program library would have to support uniform analogous data types across all processors to overcome the compiler differences, some of which are summarized later in Table V of section 4.1.4. Code would have to be generated to convert data, where necessary, that are communicated across processor bounds in support of remote object references, remote subprogram calls, and remote entry calls.

Runtime environments targeted for each processor would have to be developed/modified to support the level of desired distribution. Restrictions on the level of distribution would have a large effect on the resulting runtime overhead (i.e., greater distribution flexibility would result in larger runtime overhead and size). Mechanisms within the runtime are required to support memory management, time management, tasks, and exceptions. Other

Table IV Ada Runtime Modification Requirements

Ada Development System	An Ada compiler that targets multiple processors accompanied by tools for linking, downloading, executing, and debugging; or, Multiple Ada compilers that can share information through a common program library to support interface and type checking.
Units of Distribution	Restrictions must be determined on allowable Ada units of distribution.
Memory Management	A mechanism is required to support the shared-memory model of Ada.
Time Management	A mechanism is required to support global time of day as well as delays.
Task Communication	A mechanism is required to support the communication of data between tasks running on distributed heterogeneous tasks.
Task Activation and Termination	A mechanism to support the activation and termination of distributed tasks.
Task Scheduling	A mechanism to support the scheduling of tasks.
Exception Handling and Propagation	A mechanism to support the handling and propagation of exceptions across processor bounds.
Other	Support for interrupts, I/O, predefined packages, generic units, and compiler attributes.

mechanisms are needed to support processor interrupts, I/O, predefined packages, generic units, and compiler/processor independent/dependant attributes.

It is not impossible, however, to implement the first model for homogeneous processors as suggested by Brennan [BRE89]. This model would leave the detail of distribution mostly invisible to the applications programmer. It may require extensive modifications, depending on the desired level of

distribution, to the Ada runtime to support a distributed network of homogeneous processors. To support a distributed network of heterogeneous processors would require an even greater effort determined by the number of unlike cooperating processors. The addition of presentation layer services to accommodate heterogeneous processors was briefly mentioned in Brennan's model and not included in his proposed communications kernel. One consideration would be to standardize the communications kernel such as that suggested by Brennan within the Ada runtime. This could make it possible for the products, in particular Ada runtimes, of many Ada vendors to interoperate.

Clearly, a lot of effort would have to go into the design, development, or modification of the Ada compiler, program library, support tools, and runtime environment to overcome the requirements needed to support this model of distribution.

The second model used to build distributed systems with Ada is much more practical when dealing with loosely coupled heterogeneous systems. The external mechanism used to provide the distributed functionality can simply be an Ada package. This package can be an interface to a standard network protocol [RAB89] [RAB90] [STE88] [STE89]. In this way, distributed architectures and systems of the past can be migrated into Ada with minimal, if any, changes to the Ada runtime. Some of Ada's tight type checking is lost across heterogeneous processor (program) bounds, however.

The following requirement is needed to ensure an interface that mates well with the semantics of Ada:

- The Ada runtime must be capable of suspending a task that is waiting for I/O, and must also be capable of resuming the task upon completion of the I/O if the task has the highest priority [RAB90]. The task is marked ready to run otherwise.

Frequently, implementations of Ada do not provide a program asynchronous task synchronous (PATS) I/O capability. Because of this, when an I/O function is called, the entire Ada task tree is blocked. Clearly, this is an undesirable side effect and leads system designers and programmers to solutions which do not involve Ada tasks. What is needed is the capability for just the task to block that is calling a blocking network message passing primitive. The Ada runtime would then schedule other tasks to run while waiting for the I/O to complete. Upon completion of the message passing primitive, the calling task would be preemptively rescheduled to run if the currently running task is of lower priority. If the currently running task is of equal or higher priority, then the task would wait its turn to run (this is dependent on the scheduling algorithm employed). Higher levels of inter-processor communication such as remote procedure calls (RPC) can be built on top of this message passing mechanism.

VAX Ada provides a package called TASKING_SERVICES that provides a PATS implementation of most of the system services of VMS, that cause a VMS process to block. The underlying blocking VMS system service used for I/O is the QIOW (queue an I/O request and wait). Through the use of the Ada procedure TASKING_SERVICES.TASK_QIOW, only the calling task blocks, not the entire VMS process. This technique was employed by the author in the implementation of an Ada interface to DECnet in the package DECnet_IO [STE88] and led to desirable results. This technique was not employed, however, to an Ada interface to TCP/IP in the package TCP_IO [STE89] and led to the undesirable results described above. These problems were averted through an implementation modification to the package body of TCP_IO for the purposes of this report (use of the procedure TASKING_SERVICES.TASK_QIOW was employed) to gain the desired PATS functionality.

In light of this analysis, it is recommended that the most cost effective and timely solution to developing distributed heterogeneous systems in Ada, for the near future, would be to use single Ada programs that communicate with each other through a well-defined standard network interface in the form of an Ada package. Section 4 will illustrate this technique by describing two such packages implemented on heterogeneous systems.

4 PROGRAMMING TECHNIQUES

Application programming techniques are not much of an issue when distributed execution is handled invisibly by the Ada runtime. An application programmer does not worry about distribution or specifies the distribution later, at system bind time. In the case where this is not so, all of the responsibility of distribution lies on the system designer and programmer. This section identifies techniques that aid the programmer in developing distributed Ada programs in a consistent fashion.

4.1 A Network Interface

This section will discuss two Ada interfaces to the TCP/IP network protocol. One interface was developed as part of the Advanced Combat System Interactive Design Laboratory (ACSIDL) which was developed at the Naval Underwater Systems Center (NUSC) in Newport, Rhode Island [STE89]. The second interface was developed for the purpose of illustrating the conclusions of this study.

The need for the first interface to the TCP/IP network protocol stemmed from a requirement to develop a distributed simulation involving heterogeneous processors. The simulation was to be coded in VAX Ada on a VAX/VMS computer system. The VAX was chosen for its production quality Ada compiler and mature software development tools. The rational for this decision was to help reduce the risk of implementing a large piece of code in Ada by a team of designers and programmers new to the language. The display systems that were employed for the man-machine interface which were used to observe and manipulate the simulation where Silicon Graphics Iris Workstations. The Iris was chosen for its superior graphics handling capabilities. All systems involved were connected to a common Ethernet cable. Each system supported the TCP/IP network protocol. The TCP/IP network protocol was fortunately designed with heterogeneity in mind.

The second interface was developed on a Sun workstation to illustrate the points and considerations being discussed in this report. An attempt was made to develop the interface on the Silicon Graphics Iris workstation, but this attempt was unsuccessful because of the immaturity of the MIPS Ada compiler. The MIPS Ada compiler used the Verdix Ada compiler front end, together with a MIPS-developed code generator, and Ada runtime. I decided to develop the interface on the Sun workstation as an alternative due to the similarities of the underlying data representations used on both the Sun and Iris workstations. The Verdix Ada compiler hosted on and targeted for the Sun workstation was used for the implementation of this network interface. The interface for the most part is identical in the functionality provided to that of the interface developed on the VAX.

The VAX and Sun systems have some fundamental differences in architecture that had to be contended with. The VAX is a little endian system, which implies that its bytes are numbered with byte 0 being the low-order rightmost byte, whereas the Sun is a big endian system, which implies that its bytes are numbered with byte 0 being the high-order leftmost byte [TAN88]. In addition, the VAX uses a proprietary F_FLOATING single-precision floating-point representation, whereas the Sun uses a standard IEEE single-precision floating-point representation. The low level representation of system time differed on both systems. However, both systems use ASCII character and string representation and the same bit representation for integers.

The interface is implemented as an Ada package called TCP_IO on the VAX and TCPIO on the Sun. The set of primitives make up a subset of the primitives available in a full TCP/IP implementation. The primitives do, however, provide all of the functionality needed to perform reliable connection-oriented message transfers.

4.1.1 TCP_IO Interface Primitives

The following is a list of primitives contained in the VAX-based package TCP_IO. The list briefly describes the functionality of each primitive along with its Ada interface. The primitives represent a subset of all primitives available in most implementations of TCP/IP. Those that provide virtual circuit capabilities were implemented.

Socket This procedure is used to create a socket that is an end point for communication.

```
procedure SOCKET (
  S        :    out integer;  -- socket
  DOMAIN   : in     integer;  -- communications domain
  S_TYPE   : in     integer;  -- communication semantics
  PROTOCOL : in     integer); -- specifies a particular protocol
```

Bind This procedure is called after a socket has been created. Bind assigns a port name or number to be used as a reference by other processes on the network.

```
procedure BIND (
  STATUS  :    out integer;    -- return status
  S       : in     integer;    -- socket
  NAME    : in     SOCKADDRS; -- name to assign to socket
  NAMELEN : in     integer);   -- length of name
```

Connect This procedure called by a process to establish a connection to a remote process. The node name and the port name or number used in the BIND call by the remote process is used in this call.

```
procedure CONNECT (
  STATUS  :    out integer;   -- return status
  S       : in     integer;   -- socket
  NAME    : in     SOCKADDRS; -- remote socket name
  NAMELEN : in     integer);  -- name length
```

Listen This procedure determines the allowable backlog of incoming connection requests.

```
procedure LISTEN (
  STATUS  :    out integer;  -- return status
  S       : in     integer;  -- socket
  BACKLOG : in     integer); -- max length of queue of pending
                             -- connections
```

Netclose This procedure is used to close a socket and end the communication session.

```
procedure NETCLOSE (
  S : in     integer); -- socket
```

Netread This procedure is used to read messages from a socket that has been connected to another socket.

```
procedure NETREAD (
  CC     :    out integer;        -- return length
  S      : in     integer;        -- socket
  BUF    : in     SYSTEM.address; -- buffer address
  NBYTES : in     integer);       -- buffer length
```

Netread_Buffered This procedure is used to buffer successive NETREADs to result in a message of user-specified length. A producer process may send a message of greater length than is received by NETREAD. Therefore, multiple NETREADs must be performed by the consumer process. This procedure hides the details of this operation.

```
procedure NETREAD_BUFFERED (
  CC     :    out integer;        -- return length
  S      : in     integer;        -- socket
  BUF    : in     SYSTEM.address; -- buffer address
  NBYTES : in     integer);       -- buffer length
```

Netwrite This procedure sends a message on the specified socket.

```
procedure NETWRITE (
  CC      :    out integer;         -- return length
  S       : in     integer;         -- socket
  BUF     : in     SYSTEM.address; -- buffer address
  NBYTES  : in     integer);        -- buffer length
```

Rhost This procedure is used to look up an internet host by name and return a 32-bit internet address.

```
procedure RHOST (
  IADDR :    out integer; -- 32-bit internet address
  ANAME : in out string); -- host name
```

TCP_Accept This procedure is used to accept inbound connection requests from a socket. It returns a new socket to be used in communicating to the requesting process.

```
procedure TCP_ACCEPT (
  NS      :    out integer;    -- new socket returned
  S       : in     integer;    -- socket
  ADDR    :    out SOCKADDRS; -- address of the connecting entity
  ADDRLEN :    out integer);   -- length of address returned
```

Put_Bin Two overloaded procedures used to print the binary values of integers or 32-bit array types to the screen. These procedures are predominantly used for debugging.

```
procedure PUT_BIN (
  BUFFER : in     integer);

procedure PUT_BIN (
  BUFFER : in     SYSTEM.BIT_ARRAY_32);
```

Htons This function converts the host byte ordering to network byte ordering for a two-byte word.

```
function HTONS (BUFFER : in short_integer)
  return short_integer;
```

Htonl This overloaded function converts the host byte ordering to network byte ordering for a four-byte long word.

```
function HTONL (BUFFER : in integer)
  return integer;

function HTONL (BUFFER : in SYSTEM.BIT_ARRAY_32)
  return SYSTEM.BIT_ARRAY_32;
```

4.1.2 Package VAX_IEEE

The package VAX_IEEE was written to deal with the architectural and data representation differences between the VAX and Sun computer systems. In a sense, this package can be viewed as providing the presentation layer services needed for this application. The design decision was made to perform all data representation conversions on the VAX. This was done to make all data representation issues completely invisible to the man-machine interface system designers and application programmers utilizing the Sun (and the Iris in the case of ACSIDL) system. All responsibility was assumed by the system designers and application programmers implementing the simulation that utilized the VAX system.

The following is a list of primitives contained in package VAX_IEEE. The list briefly describes the functionality of each primitive along with its Ada interface.

Convert_IEEE_Float_To_VAX_On_VAX This function converts the float bit representation of IEEE single-precision floating point numbers to the bit representation of VAX F_FLOATING single-precision floating point numbers.

```
function CONVERT_IEEE_FLOAT_TO_VAX_ON_VAX (
  BUFFER : in SYSTEM.BIT_ARRAY_32)
  return float;
```

Convert_VAX_Float_To_IEEE_On_VAX This function converts the float bit representation of VAX F_FLOATING single-precision floating point numbers to the bit representation of IEEE single-precision floating point numbers.

```
function CONVERT_VAX_FLOAT_TO_IEEE_ON_VAX (
  BUFFER : in float)
  return SYSTEM.BIT_ARRAY_32;
```

LIB_DAY This procedure returns the number of days since the system zero date of 17 November 1858, or the number of days from system zero date to a user supplied date. This is a direct interface to the VAX/VMS operating system runtime library routine LIB$DAY.

```
procedure LIB_DAY (
  DAY_NUMBER :    out integer;
  USER_TIME  : in     CALENDAR.TIME;
  DAY_TIME   :    out integer);
```

Get_Integer_Time This function converts a time value into an integer value, which represents the number of seconds from base time 1 January 1988.

```
function GET_INTEGER_TIME (
  TIME : in CALENDAR.TIME)
  return integer;
```

4.1.3 TCPIO Interface Primitives

The following is a list of primitives contained in the Sun based package TCPIO. The list briefly describes the functionality of each primitive along with its Ada interface. The primitives in this package are similar to the primitives in the VAX based package. The differences are due to the differing methods used to facilitate the pragma INTERFACE in both compilers. The Verdix compiler only allows parameters of mode "in". All parameters must be 32 bits in size.

Socket This function is used to create a socket that is an end point for communication.

```
function Socket (
  Domain   : in     integer; -- communications domain
  S_Type   : in     integer; -- communication semantics
  Protocol : in     integer) -- specifies a particular protocol
  return integer;            -- socket
```

Gethostbyname This function is used to look up an Internet host by name and return information about the name including a 32-bit Internet address.

```
function Gethostbyname (
  Name : in     string) -- host name
  return HOSTENT_PTR;   -- host name record including address
```

Bind This function is called after a socket has been created. Bind assigns a port name or number to be used as a reference by other processes on the network.

```
function Bind (
  S       : in     integer;       -- socket
  Name    : in     SOCKADDRS_PTR; -- name to assign to socket
  Namelen : in     integer)       -- length of name
  return integer;           -- status
```

Listen This function determines the allowable backlog of incoming connection requests.

```
function Listen (
  S       : in     integer; -- socket
  Backlog : in     integer) -- queue length
  return integer;           -- status
```

Connect This function called by a process to establish a connection to a remote process. The node name and the port name or number used in the BIND call by the remote process is used in this call.

```
function Connect (
  S       : in     integer;        -- socket
  Name    : in     SOCKADDRS_PTR; -- remote socket name
  Namelen : in     integer)        -- name length
  return integer;                  -- status
```

TCP_Accept This function is used to accept inbound connection requests from a socket. It returns a new socket to be used in communicating to the requesting process.

```
function TCP_Accept (
  S       : in     integer;        -- socket
  Addr    : in     SOCKADDRS_PTR; -- remote address
  Addrlen : in     ADDRLEN_PTR)    -- address length
  return integer;                  -- status
```

Recv This function is used to perform an unbuffered read of a message from a socket that has been connected to another.

```
function Recv (
  S       : in     integer;        -- socket
  Buf     : in     SYSTEM.address; -- buffer address
  Len     : in     integer;        -- buffer length
  Flags   : in     integer := 0)   -- flags
  return integer;                  -- return length or status
```

Send This function is used to perform an unbuffered write of a message to a socket that has been connected to another.

```
function Send (
  S       : in     integer;        -- socket
  Msg     : in     SYSTEM.address; -- buffer address
  Len     : in     integer;        -- buffer length
  Flags   : in     integer := 0)   -- flags
  return integer;                  -- return length or status
```

Close This procedure is used to close a socket and end the communication session.

```
procedure Close (
  S       : in     integer); -- socket
```

Netread This function is used to read messages from a socket that has been connected to another socket.

```
function Netread (
  S       : in     integer;         -- socket
  Buf     : in     SYSTEM.address; -- buffer address
  Nbytes  : in     integer)         -- buffer length
  return integer;                   -- return length or status
```

Netwrite This function sends a message on the specified socket.

```
function Netwrite (
  S       : in     integer;         -- socket
  Buf     : in     SYSTEM.address; -- buffer address
  Nbytes  : in     integer)         -- buffer length
  return integer;                   -- return length or status
```

Netread_Buffered This function is used to buffer successive NETREADs to result in a message of user-specified length. A producer process may send a message of greater length than is received by NETREAD. Therefore, multiple NETREADs must be performed by the consumer process. This function hides the details of this operation.

```
function Netread_Buffered (
  S       : in     integer;         -- socket
  Buf     : in     SYSTEM.address; -- buffer address
  Nbytes  : in     integer)         -- buffer length
  return integer;                   -- return length or status
```

Perror This procedure prints the actual error message to standard output. This procedure is usually called after a status of -1 is returned from other TCPIO primitives.

```
procedure Perror (
  S       : in     string); -- error string
```

4.1.4 Compiler Differences

There are differences evident in the implementations of the network package interfaces described in the sections above. Part of this is because the TCP_IO interface developed for the VAX was completed before knowledge of the Sun Ada compiler existed. Pragma INTERFACE for the VAX Ada compiler can support all possible parameter passing mechanisms (value, reference, string descriptor) and modes (in, out, in out), whereas the Verdix Ada compiler on the

Sun and the MIPS Ada compiler on the Iris (an interface was attempted on this system without success because of compiler bugs) support only pass by value of mode IN and each parameter is limited to a 32-bit size. Had this been known from the beginning, some careful design of the network package interface on each system could have possibly resulted in an identical interface.

Support for the built-in data types by Ada in package STANDARD on each system, also contained some fundamental differences. The preferred method used to deal with this problem is to define a portable derived type for all required types and refrain from using the predefined types. The following type declaration is an example of a portable derived type:

```
type MY_FLOAT is new float;
-- or
type MY_FLOAT is new short_float;
```

This declaration would be the only change necessary to port the program from one system to the next. Table V illustrates some of these differences.

Table V Ada Compiler Type Differences

Size (bits)	**VAX Ada**	**Verdix Ada (SUN)**	**MIPS Ada (IRIS)**
8	short_short_integer	tiny_integer	tiny_integer
16	short_integer	short_integer	short_integer
32	integer	integer	integer
32	float (F_FLOATING)	short_float (IEEE)	float (IEEE)
64	long_float (D_FLOATING)	float (IEEE)	long_float (IEEE)

The declarations shown in Table V would aid in making the Ada source file portable, but would not help the situation where the distribution is provided in the runtime. If there is no global agreement between vendors of Ada runtimes on the textual representation of analogous data, it would be impossible to implement distributed capability in a heterogeneous environment.

4.1.5 Primitive Calling Sequence

The following sections will describe how to use the previously specified network procedures by stating their required calling order by processes on both sides of a communications channel. The processes can be thought of as a client and a server. The client being the process requesting a communication connection to a server, and the server being the process accepting incoming communication connection requests from clients. For the purposes of this report, the process is an Ada task and will be referred to in the following sections as simply a task.

4.1.5.1 Server Calling Sequence

To establish a network connection between two tasks on the network, the task must first create a TCP socket using the procedure SOCKET. This socket can be thought of as one end of a communication channel, much like a telephone.

To enable other tasks on the network to connect to this socket, a task must make its socket known to the network by use of the procedure BIND. This procedure basically associates the socket with a port on the node in which the task resides, much like publishing ones telephone number. This task will be referred to as the server.

The server task must next specify the size of the incoming request queue before a connection request can be accepted. This is accomplished through use of the procedure LISTEN. This determines the number of pending incoming connection requests allowed on the socket.

The server task must then call TCP_ACCEPT. This call will block the task until an incoming connection request is initiated by a client for this known TCP port. This is analogous to a person waiting for the telephone to ring. It is at this point that I would like to reemphasize the need for a PATS implementation of the blocking primitives such as TCP_ACCEPT. Without it, other Ada tasks within the program would be needlessly blocked, resulting in undesirable runtime effects.

When a request from a client task is received, a new socket (NS) is automatically returned from TCP_ACCEPT. This is analogous to a person answering the telephone. Bidirectional communication is now possible between the server and client tasks over the new socket. This is analogous to two persons holding a conversation over the telephone. This may be accomplished through use of the procedures NETREAD, NETREAD_BUFFERED, and NETWRITE. The old socket still exists and can be used to accept inbound connection requests from other tasks.

When termination of communication is desired, the procedure CLOSE is called to close the communications channel, much like hanging up the telephone. The sequence of calls and their relationship in time are shown in Figure 12.

4.1.5.2 Client Calling Sequence

The client task must also create a TCP socket by calling SOCKET to establish its end of the communication channel. The client task then requests a connection to an existing known TCP port by calling the procedure CONNECT. This is similar to dialing the telephone where the telephone number relates to the network node address and port number of the server task. If an unknown node address or port number is specified, an error condition is returned indicating that a time-out has occurred.

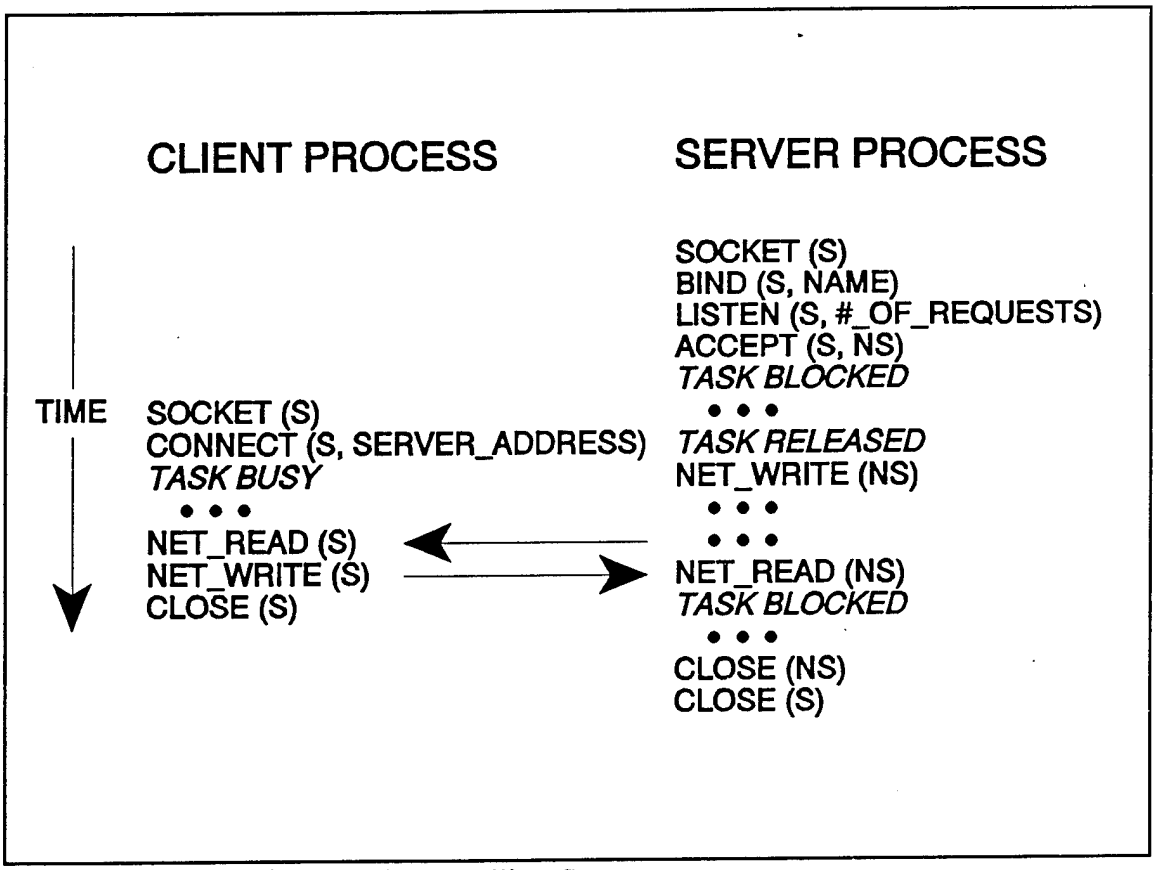

Figure 12 TCP/IP Primitive Calling Sequence

When a successful connection is established, the client and server tasks are able to communicate with each other. The client like the server may call NETREAD, NETREAD_BUFFERED, and NETWRITE as if two people were having a conversation on the telephone. When termination of communication is desired, the procedure CLOSE is called to close the communications channel, much like hanging up the telephone.

4.1.6 Programming Considerations

One of the more attractive features of Ada is its tasking feature that allows concurrent operations to take place in the context of a single program. It would be desirable to maintain concurrent programs in such a way as to present a seamless and coherent system. It would also be desirable to maintain the Ada semantics of calling subprograms and task entries across program and processor boundaries within the software system. If the interface is designed such that the primitives execute in PATS format, then the interface can be used in conjunction with Ada's calling semantics. TCP_IO provides a PATS interface to the network allowing asynchronous, bidirectional communication of messages between programs.

4.1.7 Ada Program Communication

Each pair of Ada programs or processes in a software system requiring network communication will provide a sender task with its peer process providing a cooperating receiver task. Both the sender and the receiver tasks will communicate over the same network virtual circuit. The sender task will send messages to the receiver task. For heterogeneous systems, the messages must be converted by either the system sending the message or the system receiving the message. If the messages are constructed as Ada variant records with a default descriminant value, it is possible for the receiver task, upon receiving a message, to determine the type of the incoming message. It is possible for the receiver task to perform the appropriate action once the message type is determined.

Messages can be coded as variant record types where the discriminant specifies the message type being sent or received as indicated previously. This makes it possible to invoke multiple types of actions in a remote Ada program, or process, over a single network circuit. Each cooperating network process residing within the software system would have both a sender task and a receiver task. In the case of the ACSIDL simulation, the Ada program performing the simulation on the VAX contained one sender task and many receiver tasks for each process, i.e., each Iris display system participating in the distributed application. The single sender task would communicate with all of the remote receiver tasks by maintaining a socket channels list containing one socket channel for each remote receiver task. Future versions of the ACSIDL simulator are anticipated to employ broadcast datagrams defined by user datagram protocol (UDP) (part of TCP/IP) to implement data sharing more efficiently. It is not necessary to maintain a single sender task, however. Many sender tasks, one for each remote process, may be implemented. The general architecture of the sender/receiver task communication scheme is depicted in Figure 13.

Some of Ada's strong type checking is lost in a heterogeneous environment. The necessary type checking across processor bounds must be carefully maintained by the developer.

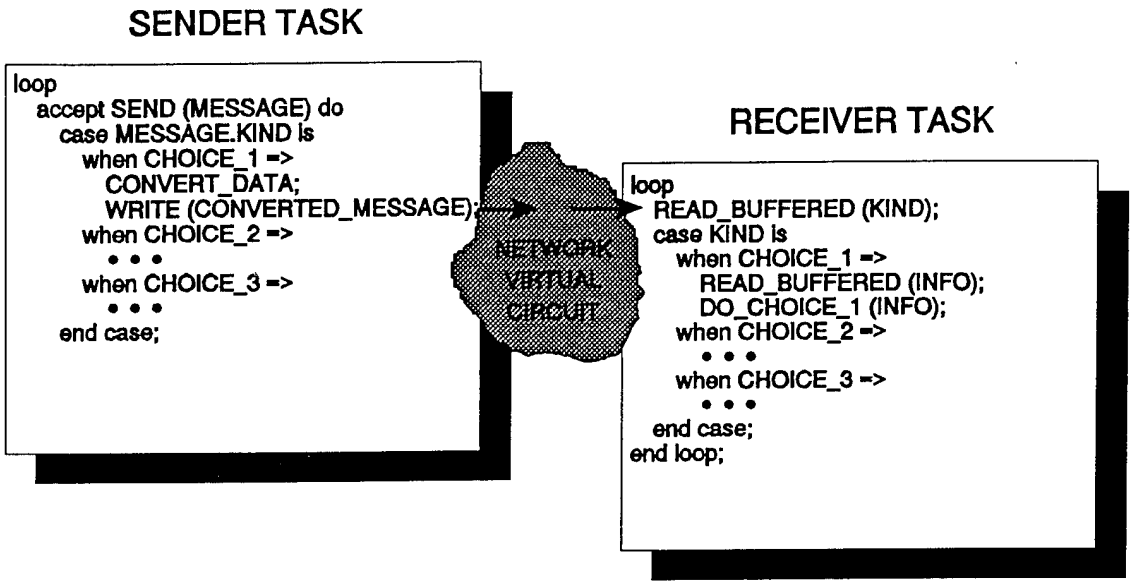

Figure 13 Sender/Receiver Tasks

The sender task waits to rendezvous with the application when communication is desired. When the rendezvous occurs, the sender task determines the message kind. This allows the sender to implicitly determine the size of the message and the representation conversion to be performed on the message before transmitting to the remote process. Since the Ada attribute 'size will return the size of the largest variant of the network message, it is necessary for the programmer to maintain the size of each variant component of the message. This can be accomplished by specifying a separate Ada type for each variant part. The size attribute will return the correct length of the variant component type. This size, plus the size of the discriminant itself, determines the size of the entire message to be sent. The network primitive NETWRITE contains a parameter for specifying the length of the buffer to send. Specifying the actual length of the message (the size in bytes of the discriminant plus the size of the type of the information field) makes it possible to optimize the use of the network. The Ada attribute 'size applied to the variant record itself would return the size of the largest variant. This length could be used for all variants sent over the network and would result in correct information being received by the receiver. But, a lot of bytes containing no information at all would be sent and received, possibly consuming valuable network bandwidth. Once the data representation is converted and the size is determined, the message is sent over the network to the receiver task. The sender task then loops around and waits for another request (rendezvous) from the application to send another message, thus repeating the process described above.

Because TCP circuits are stream-oriented pipes used for communication, the receiver task must operate a little differently. The receiver task will receive the discriminant from the network using a call to NETREAD_BUFFERED specifying a message length that is the size of the discriminant of the message. This information determines the kind of message to follow and its implicit size.

Once the kind of incoming message is determined, it is then possible for the receiver task to read the remaining portion of the message using NETREAD_BUFFERED by specifying a message length that represents the size of the information field for the determined kind of message. The data representation should be converted here if necessary, and the appropriate actions for the message received should be executed. The receiver task then loops around to call NETREAD_BUFFERED, again specifying a message length that is the size of the discriminant, and repeats the entire process described above.

Since network messages are coded as variant record types (records with discriminants), the discriminant is used to describe the type of message, and the corresponding portion contains the information for that message kind. An example of a network message coded in Ada is shown below.

```
type KINDs is (ALERT, SIGNAL, A, B, C);

type ALERTs is
  record
    WHICH  : integer;
    ACTION : ACTIONs;
  end record;

type SIGNALs is
  record
    WHICH  : integer;
    ACTION : SIGNAL_ACTIONs;
  end record;

subtype As is integer;

subtype Bs is float;

subtype Cs is boolean;

type MESSAGEs (KIND : KINDs := ALERT) is
  record
    case KIND is
      when ALERT  => ALERT_INFO  : ALERTs;
      when SIGNAL => SIGNAL_INFO : SIGNALs;
      when A      => A_INFO      : As;
      when B      => B_INFO      : Bs;
      when C      => C_INFO      : Cs;
    end case; -- KIND
  end record; -- MESSAGEs

-- declare the actual network message object

NETWORK_MESSAGE : MESSAGEs;
```

4.1.8 Implementation of Remote Procedure Call

Higher levels of abstraction can be placed on top of the mechanism described in the previous section such as remote procedure call (RPC). RPC is a mechanism widely used in the implementation of distributed systems. Even higher levels of abstraction are placed on top of the RPC mechanism such as the Sun network file system (NFS). In RPC the client calls a local procedure to perform a service. The local procedure, in turn, calls another procedure, which actually resides on another node of the network to perform the service required. The goal of RPC is to provide the services of a procedure to a client regardless of where the procedure actually resides. The procedure appears local to the client, thus making the RPC mechanism transparent.

Figure 14 depicts a method that can be used to implement RPC on heterogeneous systems using the facilities of TCP/IP through the interfaces described in previous sections. It is shown that a client calls a local procedure, which is labeled the pseudoprocedure. This procedure can in fact be an Ada subprogram (procedure or function) or an Ada task entry (which presents the same semantics as the procedure). This provides tremendous flexibility in implementing distributed use of all the callable facilities provided by Ada. The job of the pseudoprocedure is to assemble all of the parameters, if any, into a message. This operation is known as parameter marshalling [TAN88]. It is then necessary for some heterogeneous systems to convert the data in the message into a form that is usable by the remote system. This operation is depicted by the convert data box. The message is then handed to the sender task that was described in the previous section. The sender calls the facilities of the network interface (in this case TCP_IO or TCPIO) to send the message to the remote system via a virtual circuit. It is the responsibility of the implementor of the local procedure to provide for the desired blocking semantics. It is possible to have the following client-pseudoprocedure calling semantics.

- Local procedure returns to the client after the RPC is completed by the remote procedure. Parameters are passed to the remote procedure and the resulting parameters are returned. This is a fully acknowledged semantic.

- Local procedure returns to the client as soon as the message is passed to the sender. Parameters may be passed to the remote procedure but none are returned. There is no acknowledgment that the remote procedure was truly called.

- Local procedure returns to the client after the remote procedure is called, but before the remote procedure completes. This provides an acknowledgment that the remote procedure was called.

The message is received by a receiver task on the remote system through the facilities of the network interface. The receiver task is able to determine the

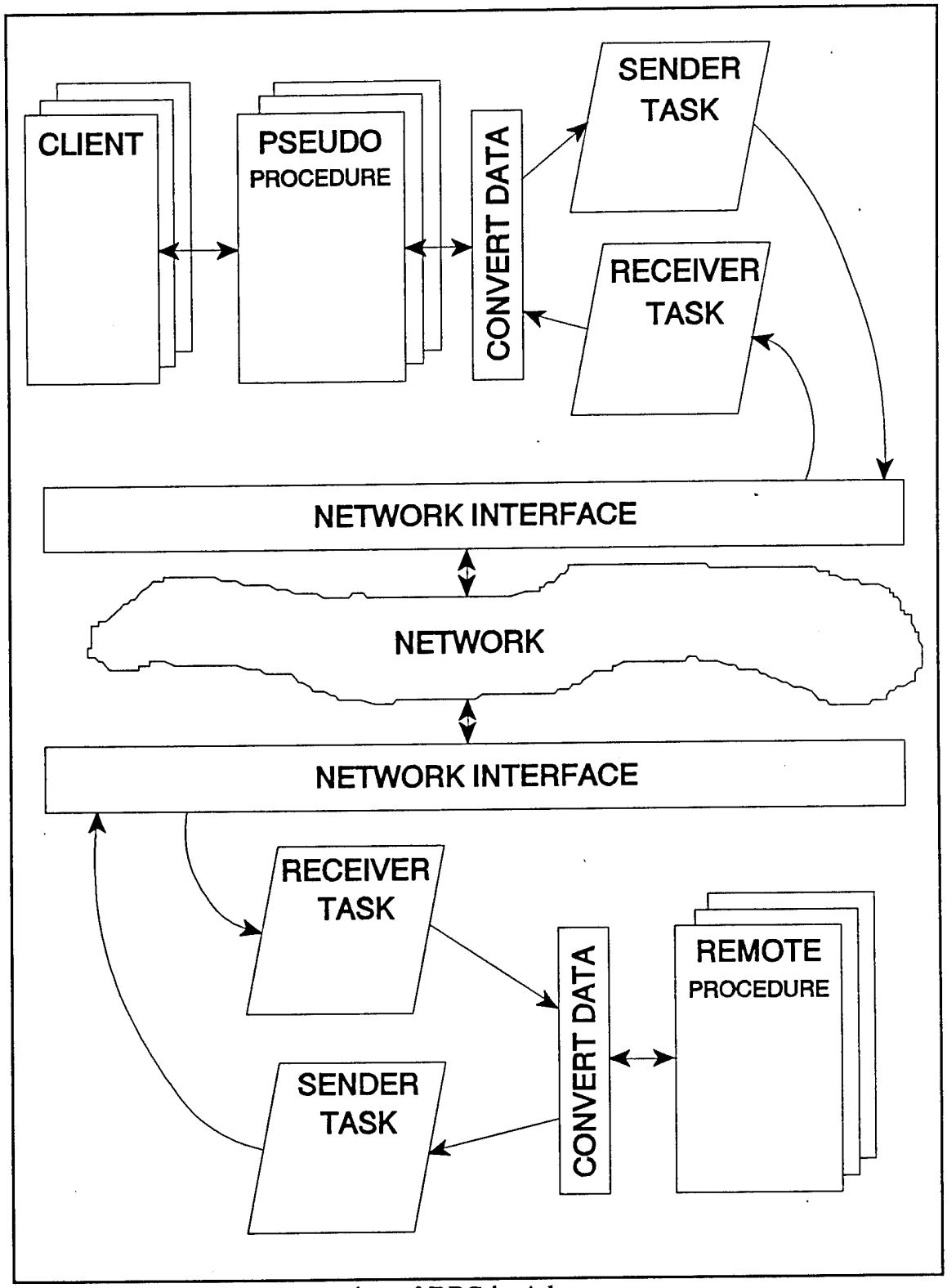

Figure 14 An Implementation of RPC in Ada

message kind and thus, the actions needed to be performed as described in the previous section. The data are converted, if necessary, to a usable form. The remote procedure is then called. The implementor must consider whether it is desirable to block the receiver task at this point. If this is undesirable, then an intermediate task must be placed between the receiver task and the remote procedure. This would allow the receiver to continue to receive incoming messages while the RPC is being serviced. The remote procedure can be an Ada subprogram or task entry just as the local procedure was. In fact, it would be desirable to have a one-for-one match between pseudoprocedure and remote procedure type. Upon completion of the service, the return parameters are assembled into a message (parameter marshalling), which is sent back via the sender task on the remote system.

It is possible to have many simultaneous RPCs, at various stages of completion, in progress at any moment in time, by any two systems connected via a virtual circuit. This is depicted by the multiple copies of the client, pseudoprocedure, and remote procedure boxes. The call can also be made by a client on either system to the server on the other. There is no limitation to the possibilities offered by this mechanism. However, responsibility for all the implementation details are not transparent and must be attended to carefully. Tools can be developed to provide the illusion of a single Ada program running across multiple systems. The tools would split the Ada program into multiple Ada programs, one targeted for each node on the network. The underlying mechanism needed to provide transparent subprogram calls or rendezvous could be implemented using the RPC mechanism presented here. An Ada runtime written in Ada could very well employ the mechanism described here.

5 CONCLUSIONS

A taxonomy of Navy system architectures was presented showing the requirements for support of heterogeneous systems connected in various degrees of distribution, along with a high-level overview of processor interconnects such as memory buses, backplanes, and networks. It was seen that backplanes can support complex protocols that could lead one to find the backplane indistinguishable from a LAN. This feature becomes especially important when dealing with heterogeneous processors because the presence of shared memory may be inefficient as a medium for sharing data between processors. This feature would be needed for an efficient implementation of an Ada runtime across multiple processors. Since there is a high likelihood of being unable to find an Ada compiler vendor to support all conceivable target processors and even if so, not providing the hooks and handles in the Ada runtimes to implement a distributed Ada runtime across the heterogeneous suite of processors, another method of implementing distribution should be considered in the interest of conserving time and money in the development process. This does not mean that a standard could not be developed specifying a uniform Ada runtime interface to the hooks and handles needed to implement distributed Ada. Therefore, it would be possible for multiple vendors to supply Ada compilers and runtimes for different target processors and allow the application developer to supply the connecting pieces to the runtimes to implement the needed distribution invisibly. This does not, however, answer the needs of some applications that require the full functionality of some of the network protocols that were discussed, such as broadcast, or higher-level abstractions, such as distributed name services. The broadcast feature is not provided by the Ada runtime and thus, would be an extension to the language to provide a broadcast entry point call and accept. This could, however, be solved by some of the additions that are being, or could be, proposed by the ongoing Ada9X effort.

A matrix of services versus distributed systems architecture provided an alternate view, which illustrated that the types of service needs differ with distributed systems architectures. These services do not always map to the features that are directly provided by the Ada runtime. Some of these needs could be addressed through additions to the Ada language and runtime as previously suggested. But, for the near future, it would be timely and fiscally astute to provide the means for distribution through a well-defined standard network interface (Ada package) employing the techniques described in this report.

REFERENCES

[ALN90] Alnaes, Knut and Ernst H. Kristiansen, David B. Gustavson, "Scalable Coherent Interface," SLAC-PUB-5184, Stanford Linear Accelerator Center, Stanford, CA, January 1990.

[AND90] Andrews, Warren, "Futurebus+ now: Profiles Defined, Support Expanded," *Computer Design*, Vol 29, No 7, 1 April 1990, pp 22 - 26.

[BAM88] Bamberger, Judy and Roger Van Scoy, "Distributed Ada Real-Time Kernel," *Proceedings of the IEEE 1988 National Aerospace & Electronics Conference: NAECON 88*, May 1988, pp 1510 - 1516.

[BAK86] Baker, T. P., and K. Jeffay, "A Lace for Ada's Corset," TR 86-09-05, Department of Computer Science, University of Washington, Seattle, WA, 25 October 1986.

[BOR90] Borrill, Paul L., "What is Futurebus+ ?," *Proceedings of the BUSCON Conference*, 14 - 16 February 1990, pp 303 - 315.

[BRE89] Brennan Jr., J. W., "Issues in the Design of Distributed Ada Programs," Masters Thesis, University of Rhode Island, August 1989. Also published as "Issues and Approaches in the Design of Distributed Ada Programs," NUSC Technical Report 6834, Naval Underwater Systems Center, Newport, RI, 11 October 1989.

[CHO89] Cholerton, Andrew, "Ada For Closely Coupled Multiprocessor Targets," *TRI-Ada '89 Proceedings*, September 1989, pp 450 - 461.

[CLA89] Clapp, Russell M., and Trevor Mudge, "Ada on a Hypercube," *Ada Letters*, Vol IX, No 2, 1989, pp 118 - 128.

[DEC87] "DECnet DIGITAL Network Architecture (Phase V)," Digital Equipment Corporation, Maynard, MA, September 1987.

[ELL89] Ellis, John R., "A Periodic Ada Control Kernel (PACK)," *TRI-Ada '89 Proceedings*, September 1989, pp 464 - 473.

[GRE89] Green, Daniel T. and David T. Marlow, "Application of LAN Standards to the Navy's Combat Systems," white paper, Engineering and Technology Division, Combat Systems Department, Naval Surface Warfare Center, Dahlgren, VA, 1989.

[GUS90] Gustavson, David B., "Applications for the Scalable Coherent Interface," SLAC-PUB-5244, Stanford Linear Accelerator Center, Stanford, CA, April 1990.

[JHA89] Jha, Rakesh, "Distributed Ada - Approach and Implementation," *TRI-Ada '89 Proceedings*, September 1989, pp 439 - 449.

[LIE86] Liebein, Edward, "The Department of Defense Software Initiative - A Status Report," *Communications of the ACM*, Vol 29, No 8, August 1986, pp 734 - 744.

[LIN89] Linnig, Michael, and Donna Forinash, "Ada Tasking and Parallel Processors," *TRI-Ada '89 Proceedings*, September 1989, pp 426 - 438.

[LRM83] "Ada Programming Language," ANSI/MIL-STD-1815A, 22 January 1983.

[RAB89] Rabbie, Harold, "An Operating System for Real-Time Ada," *TRI-Ada '89 Proceedings*, September 1989, pp 490 - 497.

[RAB90] Rabbie, Harold, "Meeting Today's Requirements With Real-Time Ada," *Proceedings of the BUSCON Conference*, 14 - 16 February 1990, pp 271 - 275.

[SAF901] *Survivable Adaptable Fiber Optic Embedded Network I*, Military Handbook MIL-HDBK-0034 (Draft), January 1990.

[SAF902] *Survivable Adaptable Fiber Optic Embedded Network II*, Military Handbook MIL-HDBK-0036 (Draft), March 1990.

[STA871] Stallings, William, *Handbook of Computer-Communications Standards, Vol 1.* New York: Macmillan Publishing Company, 1987.

[STA872] Stallings, William, *Handbook of Computer-Communications Standards, Vol 2.* New York: Macmillan Publishing Company, 1987.

[STA88] Stallings, William, Paul Mockapetris, Sue Mcleod, and Tony Michel, *Handbook of Computer-Communications Standards, Vol 3.* New York: Macmillan Publishing Company, 1988.

[STE88] Stevens, Bruce W., "DECnet Ada Binding", Technical Memorandum No. 88-2152, Naval Underwater Systems Center, Newport, RI, 28 September 1989.

[STE89] Stevens, Bruce W. and Pamela R. Perras, "An Ada Interface to Networking with TCP/IP", Technical Memorandum No. 89-2036, Naval Underwater Systems Center, Newport, RI, 19 April 1989.

[TAN88] Tananbaum, Andrew S., *Computer Networks*, 2nd ed. New Jersey: Prentice-Hall, 1988.

[VAN89] Van Scoy, Roger, Judy Bamberger, and Robert Firth, "An Overview of DARK," *Ada Letters*, November/December 1989, pp 91 - 101.

[XTP90] "XTP Protocol Definition, Revision 3.5," PEI 90-120, Protocol Engines Inc., 10 September 1990.

BIBLIOGRAPHY

"Ada Programming Language," ANSI/MIL-STD-1815A, 22 January 1983.

Alnaes, Knut and Ernst H. Kristiansen, David B. Gustavson, "Scalable Coherent Interface," SLAC-PUB-5184, Stanford Linear Accelerator Center, Stanford, CA, January 1990.

Andrews, Warren, "Futurebus+ Now: Profiles Defined, Support Expanded," *Computer Design*, Vol 29, No 7, 1 April 1990, pp 22 - 26.

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

Baker, T. P., and K. Jeffay, "A Lace for Ada's Corset," TR 86-09-05, Department of Computer Science, University of Washington, Seattle, WA, 25 October 1986.

Baker, T. P., "A Corset for Ada," Version 1.1, TR 86-09-05, Department of Computer Science, University of Washington, Seattle, WA, 8 February 1987.

Borrill, Paul L., "What is Futurebus+ ?," *Proceedings of the BUSCON Conference*, 14 - 16 February 1990. pp 303 - 315.

Brennan Jr., J. W., "Issues in the Design of Distributed Ada Programs," Masters Thesis, University of Rhode Island, August 1989. Also published as "Issues and Approaches in the Design of Distributed Ada Programs," NUSC Technical Report 6834, Naval Underwater Systems Center, Newport, RI, 11 October 1989.

Carver, Richard and K. C. Tai, "Deterministic Execution Testing of Concurrent Ada Programs," *TRI-Ada '89 Proceedings*, September 1989, pp 528 - 544.

Cholerton, Andrew, "Ada For Closely Coupled Multiprocessor Targets," *TRI-Ada '89 Proceedings*, September 1989, pp 450 - 461.

Clapp, Russell M., Louis Duchesneau, Richard A. Volz, Trevor N. Mudge, and Timothy Schultze, "Toward Real-Time Performance Benchmarks For Ada," *Communications of the ACM*, Vol 29, No 8, August 1986.

Clapp, Russell M., and Trevor Mudge, "Ada on a Hypercube," *Ada Letters*, Vol IX, No 2, 1989, pp 118 - 128.

Clapp, Russell M., and Trevor Mudge, "Parallel and Distributed Issues," *Ada Letters Special Edition -- Ada Performance Issues*, Vol X, No 3, 1990. pp 33 - 37.

Cohen, Norman H., *Ada as a Second Language*, New York: McGraw-Hill, 1986.

Day, John D. and Hubert Zimmermann, "The OSI Reference Model," *Proceedings of the IEEE*, Vol 71, No 12., December 1983.

"DECnet DIGITAL Network Architecture (Phase V)," Digital Equipment Corporation, September 1987.

De Francesco N., G. Perego, G. Vaglini, and M. Vanneschi, "Framework For Data Flow Distributed Processing," *Calcolo*, Vol 17, No 4., October 1980, pp 333 - 363.

Dowling, E. J., "Testing Distributed Ada Programs," *TRI-Ada '89 Proceedings*, September 1989, pp 517 - 527.

Ellis, John R., "A Periodic Ada Control Kernel (PACK)," *TRI-Ada '89 Proceedings*, September 1989, pp 464 - 473.

Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol 32, No 2, April 1985, pp 374 - 382.

Green, Daniel T. and David T. Marlow, "Application of LAN Standards to the Navy's Combat Systems," white paper, Engineering and Technology Division, Combat Systems Department, Naval Surface Warfare Center, Dahlgren, VA, 1989.

Griest, Thomas E., "Limitations on the Portability of Real Time Ada Programs," *TRI-Ada '89 Proceedings*, September 1989, pp 474 - 489.

Gustavson, David B., "Applications for the Scalable Coherent Interface," SLAC-PUB-5244, Stanford Linear Accelerator Center, Stanford, CA, April 1990.

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

Inter-Process Communication Primer, Sun Microsystems, Mountain View, CA, 17 February 1986.

Jha, Rakesh, "Distributed Ada - Approach and Implementation," *TRI-Ada '89 Proceedings*, September 1989, pp 439 - 449.

Kenah, Lawrence J., and Simon F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, Maynard, MA, 1984.

Kim, K. H., "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions*, Vol SE-8, No 3, May 1982, pp 189 - 197.

Lamport, L., "The Weak Byzantine Generals Problem," *Journal of the ACM*, Vol 30, No 3, July 1983, pp 668 - 676.

Liebein, Edward, "The Department of Defense Software Initiative - A Status Report," *Communications of the ACM*, Vol 29, No 8, August 1986, pp 734 - 744.

Linnig, Michael, and Donna Forinash, "Ada Tasking and Parallel Processors," *TRI-Ada '89 Proceedings*, September 1989, pp 426 - 438.

Mao, T. William, and Raymond T. Yeh, "Communication Port: A Language Concept for Concurrent Programming," *IEEE Transactions on Software Engineering*, Vol SE-6, No 2., March 1980, pp 194 - 204.

Nielsen, Kjell W. and Ken Shumate, "Designing Large Real-Time Systems with Ada," *Communications of the ACM*, Vol 30, No 8, August 1987, pp 695 - 715.

Notkin, David, Norman Hutchinson, Jan Sansislo, and Michael Schwartz, "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity," *Communications of the ACM*, Vol 30, No 2, February 1987, pp 132 - 140.

Pease, M., R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, Vol 27, No 2, April 1980, pp 228 - 234.

Perrin, Mark, "What Does Ada Offer The Embedded Systems Programmer?," *Proceedings of the BUSCON Conference*, 14 - 16 February 1990, pp 291 - 301.

Rabbie, Harold, "An Operating System for Real-Time Ada," *TRI-Ada '89 Proceedings*, September 1989, pp 490 - 497.

Rabbie, Harold, "Meeting Today's Requirements With Real-Time Ada," *Proceedings of the BUSCON Conference*, February 14 - 16, 1990, pp 271 - 275.

Royce, Walker, "Reliable, Reusable Ada Components for Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)," *TRI-Ada '89 Proceedings*, September 1989, pp 500 - 516.

Survivable Adaptable Fiber Optic Embedded Network I, Military Handbook MIL-HDBK-0034 (Draft), January 1990.

Survivable Adaptable Fiber Optic Embedded Network II, Military Handbook MIL-HDBK-0036 (Draft), March 1990.

Sammet, Jean E., "Why Ada is not Just Another Programming Language," *Communication of the ACM*, Vol 29, No 8, August 1986, pp 722 - 732.

Sauer, Charles H., and K. Mani Chandy, *Computer Systems Performance Modeling*, New Jersey: Prentice-Hall, 1981.

Schonberg, Edith, and Edmond Schonberg, "Highly Parallel Ada - Ada on an Ultracomputer," *Proceedings of the Ada International Conference*, Paris, 14-16 May 1985, pp 58 - 71.

Sha, Lui, and John B. Goodenough, "Real-Time Scheduling Theory and Ada," Technical Report CMU/SEI-89-TR-14, ESD-TR-89-22, Software Engineering Institute, Pennsylvania, April 1989.

Shin, Kang G., and Mark E. Epstein, "Communication Primitives for a Distributed Multi-Robot System," *IEEE International Conference on Robotics and Automation*, 25 - 28 March 1985, pp 910 - 917.

Stallings, William, *Handbook of Computer-Communications Standards, Vol 1.* New York: Macmillan Publishing Company, 1987.

Stallings, William, *Handbook of Computer-Communications Standards, Vol 2.* New York: Macmillan Publishing Company, 1987.

Stallings, William, Paul Mockapetris, Sue Mcleod, and Tony Michel, *Handbook of Computer-Communications Standards, Vol 3.* New York: Macmillan Publishing Company, 1988.

Stevens, Bruce W., "DECnet Ada Binding," Technical Memorandum No. 88-2152, Naval Underwater Systems Center, Newport, RI, 28 September 1989.

Stevens, Bruce W. and Pamela R. Perras, "An Ada Interface to Networking with TCP/IP," Technical Memorandum No. 89-2036, Naval Underwater Systems Center, Newport, RI, 19 April 1989.

Tananbaum, Andrew S., *Computer Networks*, New Jersey: Prentice-Hall, 1981.

Tananbaum, Andrew S., *Computer Networks*, 2nd ed. New Jersey: Prentice-Hall, 1988.

Van Scoy, Roger, Judy Bamberger, and Robert Firth, "An Overview of DARK," *Ada Letters*, November/December 1989, pp 91 - 101.

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

Wegner, Peter, and Scott A. Smolka, "Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives," *IEEE Transactions on Software Engineering*, Vol SE-9, No 4, July 1983, pp 446 - 462.

Whiddett, Dick, "Distributed programs: an overview of implementations," *Microprocessors and Microsystems*, Vol 10, No 9, November 1986, pp 475 - 484.

Williamson, Ronald, and Ellis Horowitz, "Concurrent Communication and Synchronization Mechanisms," *Software-Practice and Experience*, Vol 14, No 2, February 1984, pp 135 - 151.

XTP Protocol Definition, Revision 3.5, Protocol Engines Inc., PEI 90-120, 10 September 1990.

Zhou, Chang-Lin, and Zhou Gang. "A New Language Feature For Concurrent Programming," *International Symposium on New Directions in Computing*, 12-14 August 1985, pp 311 - 317.

INITIAL DISTRIBUTION LIST

Addressee	No. of Copies
SPAWAR (231 (1), 2312 (4), 2312--Chan (1))	6
CNA	1
DTIC	1