Interprocess Communication for Distributed Robotics

David Gauthier

Bachelor of Engineering (McGill University), 1983
Bachelor of Science (McGill University), 1978

Department of Electrical Engineering

McGill University

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Engineering

July 1986

UNIVERSITÉ McGILL

FACULTÉ DES ÉTUDES AVANCÉES ET DE LA RECHERCHE

Date _____

NOM DE L'AUTEUR: _____

DÉPARTEMENT: _____ GRADE: _____

TITRE DE LA THÈSE: _____

_____

_____

1. Par la présente, l'auteur accorde à l'université McGill l'autorisation de mettre cette thèse à la disposition des lecteurs dans une bibliothèque de McGill ou une autre bibliothèque, soit sous sa forme actuelle, soit sous forme d'une réproduction. L'auteur détient cependant les autres droits de publications. Il est entendu, par ailleurs, que ni la thèse, ni les longs extraits de cette thèse ne pourront être imprimés ou reproduits par d'autres moyens sans l'autorisation écrite de l'auteur.

2. La présente autorisation entre en vigueur à la date indiquée ci-dessus à moins que le Comité exécutif du conseil n'ait voté de différer cette date. Dans ce cas, la date différée sera le _____

_____

Signature de l'auteur

Adresse permanente:

_____

_____

_____

_____

Signature du doyen si une date figure à l'alinéa 2.

(English on reverse)

Abstract

Robot tasks have come to demand flexible sensory-based 'intelligent' behaviour. As a result, one must confront a growing number of processing elements for control and sensing. It has recently become practical to link these elements with a local area network which standardizes the physical interconnection and the protocols for sending and receiving information.

The Computer Vision and Robotics Laboratory of McGill University is a distributed computing environment based on an Ethernet local area network of host computers operating under the UNIX 4.2BSD operating system, and encompassing multiple robots, vision systems, and other sensing and positioning peripherals. Programming in such a distributed system requires that user processes be provided with a reliable and efficient means of interprocess communication that would also be easy to use.

This thesis describes the design philosophy, architecture, implementation, and performance analysis of an interprocess communication programming environment that can serve as a tool for implementing distributed robot work-cell applications. The environment consists of two useful communication architectures which vary in complexity and performance, and offer a consistent network interface. The primary communication architecture presented, a Session Layer implementation based on the International Standards Organization's Open System Interconnect Basic Reference Model, offers a standard format for integrating interprocess communication primitives into sophisticated application programs; it also ensures compatibility so that programs written by different users can be integrated in work-cell applications. The secondary communication architecture, called the Network Interface Library, offers a lower level interface with very efficient end-to-end process communication.

An analysis of the implementations' performance showed that we have achieved end-to-end interprocess communication rates which meet the needs of our research community.

Résumé

Les applications robotiques avancées exigent de la part des robots un comportement dit ≪intelligent≫, basé sur l'utilisation de capteurs. En conséquence, les concepteurs de tels systèmes doivent faire face à des systèmes comportant un nombre croissant de processeurs pour la commande des robots et le traitement des données. Il est maintenant commode d'utiliser des réseaux locaux pour standardiser l'interconnection physique de ces éléments et les protocoles de communication.

Le laboratoire de Visionique et de Robotique (Computer Vision and Robotics Laboratory), au Département de Génie Electrique de McGill University possède un système informatique distribué, basé sur un réseau local Ethernet reliant des ordinateurs opérant sous le système d'exploitation UNIX 4.2BSD, comportant plusieurs robots, des systèmes de visionique et autres capteurs sensoriels. La programmation d'un système distribué tel que celui décrit requiert que les processus aient accès à des moyens de communication sûrs, efficaces et d'utilisation commode.

Cette thèse décrit la conception, l'architecture, la mise en œuvre et l'analyse des performances d'un environnement de programmation distribuée pouvant servir de base à la mise en œuvre d'applications robotiques distribuées parmi plusieurs processeurs et incluant plusieurs robots et capteurs. L'environnement offre deux méthodes de communication qui diffèrent en complexité et en performance, mais qui présente une interface au réseau uniforme. La méthode de communication primaire, ≪Session Layer≫ (Couche de Session), est basée sur le ≪Open System Interconnect Basic Reference Model≫ de l'Organisation Internationale de Normalization. Cette couche présente un format standard pour intégrer les primitives de communication entre tâches d'un programme d'application complexe; ceci permet aussi d'assurer la compatibilité entre programmes écrits par différérents usagers pour qu'ils puissent être intègrés dans une application de cellule de travail robotisée. L'architecture de communication secondaire offre une interface indépendente, d'un niveau inférieur mais très efficace pour la communication entre tâches.

Acknowledgements

Financial assistance for this work was provided by the Natural Sciences and Engineering Research Council of Canada and the Conseil d'Administration de la Fonds pour la Formation de Chercheurs et l'Aide à la Recherche.

I would like to express my sincere thanks and appreciation to my supervisor, Dr. Alfred Malowany, who provided assistance, advice, and encouragement throughout the duration of my work. This work has benefited from the contributions of many people at our laboratory. In particular, I would like to thank my colleagues, Gregory Caryannis and Paul Freedman, who participated in the development of the Session Layer and who co-authored several papers based on this research. I would also like to thank John Lloyd and Mike Parker for their comments in the initial stages of this work, and Mike Parker for his programming assistance.

Contents

List of Figures

Chapter 1 Introduction

1.1 Introduction and Motivation

Commercial robots were first introduced by Plannet Corporation in 1959. They were programmable devices, controlled by limit switches and cams, and capable of performing simple "pick-and-place" and painting tasks. The development of robotic units employing servo systems for general path control was led by Unimation Incorporated during the 1960's. Commercial robot controllers based on digital computers were realized in the early 1970's, with the introduction of T3 by Cincinnati-Milicron Corporation [Ayres and Miller 83]. This provided a convenient means of integrating external digital and analogue based equipment to the controller, allowing more complex tasks to be performed within the robot work-cell.

During the 1980's, work-cells have evolved to include enormously varied and complex configurations. The single robot has been followed by multiple robots, cooperating in three dimensional dynamic environments which allow them to accomplish complex and intricate tasks. Intelligent computer based peripherals such as vision systems are being employed for inspection of objects and robot supervision through motion coordination. Sophisticated sensing devices such as lasers and acoustic, ultrasonic, strain and pressure sensors are being incorporated for proximity determination. Controllers for lighting, motorized stages, and end effector tools are frequently components of state of the art work-cells

[Goldwasser 84] Motion planning and adaptive control are essential to these complex configurations. custom expert systems have been employed to coordinate the various elements [Gonzalez and Safabakhsh 82]

Because of the complex nature of the tasks performed, centralized control is no longer practical. The role of the robot controller as the nucleus of the work-cell has been superseded, with control now distributed amongst many elements. The robot controller has become simply another peripheral device Work-cells have become centres of distributed computing, networks of many devices This is exemplified by IBM's modular 7575 Manufacturing System which is composed of a robot manipulator, a control computer, and a servo power module. The control computer can be linked through a network to any number of other host systems.

Inherent with the increased complexity of the robotic work-cell are the problems associated with any distributed computing system. The need for an easy to use interprocess communication (IPC) environment which integrates individual elements both within and between work-cells is immediate This structure must afford efficiency in its application while facilitating user access to any given element. In addition, device dependent features should be made transparent to the user, encapsulated in such a way as to conform to a consistent interface.

In general, industry has been slow in proposing and implementing robotic work-cell communication standards. This might be attributed to the complex nature of the subject and its wide range of applications. There are as many "standards" as there are robot manufacturers. Communication remains largely vendor dependent; both hardware and software interfaces are frequently unique for each individual element. This uniqueness incurs high costs to the end user when interfacing elements. Incompatibility and installation problems can sometimes be circumvented by purchasing all equipment from a single vendor. Frequently though, each system must reflect a particular set of device requirements which are not met by a single manufacturer. In addition, companies tend to avoid single sourcing and purchase work-cell elements from different vendors. Even if a manufacturer were to

meet all the device requirements, their products may not be the most cost effective.

This situation is gradually changing with the adoption of the so called "open systems interconnect" philosophy put forth by the International Standards Organization (ISO). An open system is one for which standards have been published, allowing system components from different vendors to be integrated. The ISO has proposed a heirarchical communications framework, called the Open System Interconnect Basic Reference Model (OSI), which partitions the tasks involved in communication over a computer network into seven logical layers. Protocols describing the function of each layer have also been specified. The result of this development is a strong desire to standardize an industry that is rapidly expanding. Standardization makes communication more productive by providing a stable hardware and software model.

One example of a robot communication standard based on the ISO's open communication reference model is the Manufacturing Automation Protocol (MAP) [MAP 85], proposed by General Motors. The MAP specifications represent the first attempt at a concerted effort on an international basis by major industries acting in conjunction with professional societies to establish protocol profiles for automated manufacturing networks and to test and evaluate them in practical applications. The proposal appears to have gained considerable momentum although its adoption as a robotic communications standard is not expected in the near future. Although recently available commercial systems supporting the MAP specifications are well suited to large industrial applications such as factory floor communications, they may not be attractive for full implementation in certain research oriented environments. However, since the specifications are well designed, it is advantageous to consider adopting any applicable features into a robot communications environment.

1.2 Scope and Goals

This thesis describes the development of a communication programming environment for the Computer Vision and Robotics Laboratory (CVaRL) of McGill University.

The CVaRL communications environment is a hierarchical structure which can be described with the OSI Reference Model. The OSI Reference Model divides the tasks involved in the communication between two systems into a layered hierarchical structure composed of the Physical, Data Link, Network, Transport, Session, Presentation, and Application Layers. This model will be discussed in greater detail in Chapter 2.

The environment was expanded to include two useful communication architectures which vary in complexity and performance, and offer a consistent user interface to the network. The primary focus of this presentation is to describe the design philosophy, architecture and implementation of a Session Layer which provides a standardized and straight forward means of effecting interprocess communication. The secondary communication architecture, which is provided by a software package called the Network Interface Library, offers a flexible and very efficient but nonstandardized means of achieving end-to-end process communication.

The Session Layer provides the user with a easy to use programming interface that facilitates access to the interprocess message passing services supported by the Transport Layer. The Session Layer appears to the user as a black box which links communicating processes and acts as an intermediary in passing messages between them as shown in Figure 1.1. Communication services are presented to the user in terms of *logical* taps on the network called *endpoints* and *logical* channels between endpoints called *links*. Network communication is a complex issue, and to achieve it in an efficient way often requires an in-depth knowledge, on the part of the user, of the low level aspects of network communication. The Session Layer alleviates this through a set of high level communication primitives which allow the establishment, management, and termination of endpoints and links.

Figure 1.1 Interprocess communication mediated by the Session Layer

The Session Layer's standardized format aids in integrating interprocess communication primitives into sophisticated application programs. It ensures compatibility so that programs that were developed independently can be shared amongst users and easily integrated into their work-cell applications. Standardization is a most desirable feature to have in a research environment as it promotes productivity. Users do not have to maintain their own set of communication routines and new users can benefit from an existing library of work-cell utilities.

Implementation of the Session Layer required the development of a set of network primitives to communicate with the transmission control protocol facilities in the Transport Layer. These primitives are available to the user in the form of an interprocess communication library called the Network Interface Library (NIL). NIL supports direct user-process to user-process communication: that is, interprocess communication that is not mediated by an intermediary such as the Session Layer.

NIL simplifies the implementation of interprocess communication applications by providing an interface to the system network which is easier to use than that provided by the Transport Layer, as well as by furnishing the building blocks necessary to implement some of the most commonly used communication architectures. NIL provides a lower level network interface than the Session Layer; it is not as powerful and is somewhat more difficult to use. On the other hand, the primary advantage of the library over the Session Layer is its increased speed. This package may be used independent of, or in conjunction with, the facilities provided by the Session Layer, according to the user's demands.

The goals of this presentation are to survey methods of interprocess communication with an emphasis on those common in both general computing and distributed robotic applications; to discuss CVaRL's communication environment; to describe the design and implementation of a Session Layer and a Network Interface Library suited to our environment; to present an evaluation these of implementations; and to make suggestions for future research.

1.3 Outline of the Presentation

The remainder of this chapter is devoted to a discussion of the research goals of the robotics laboratory, a description of the implementation environment, and justification for why it should be changed.

Chapter 2 discusses of interprocess communication in the context of distributed computing. Chapter 3 presents a survey of interprocess communication in the context of distributed robotics. Chapter 4 identifies the interprocess communication services most useful to our research community, and then describes the design philosophies and architectures of our Session Layer and Network Interface Library. In Chapter 5 we use qualitative and quantitative criteria to evaluate the communication services. Chapter 6 summarizes the presentation and makes suggestions for future research.

1.4 Research Goals of the Robotics Laboratory

At the present time, the principal focus of the robotic group at CVaRL is the inspection and repair of hybrid integrated circuits. The intent is to investigate the hardware and software requirements of a robotics work-cell whose purpose is to visually inspect a target die and repair certain kinds of hybrid circuit defects. Related work is aimed at the inspection and repair of printed circuit boards, including the insertion and removal of components and the inspection of solder joints.

To this end, we are exploring certain aspects of distributed computing, especially how networking a set of work-cell elements can influence real-time performance, and how to design algorithms and data structures for a distributed environment.

1.5 Implementation Environment

In order to evaluate performance results and place them in their proper context, it is necessary to become acquainted with the implementation environment. We can char-

acterize the computer network as mini and micro computer based, running a concurrent operating system with modest input and output requirements. The primary reason for the distributed environment is to enhance performance through parallel processing. The laboratory currently has twelve computers linked by a 10MHz Ethernet local area network. The network configuration is composed of a VAX 11-780, three VAX 11-750's, two microVAX II's, and six Sun System II's. All systems operate under the University of Berkeley at San Diego's UNIX version 4.2 operating system (UNIX 4.2BSD), except for the VAX 780 which concurrently runs VMS 3.7 and EUNICE 3.1, a UNIX emulator.

A typical work-cell configuration of computer, robot, and vision and other sensing systems is shown in Figure 1.2. One VAX 750 is dedicated to robot control applications while the other systems are directed towards vision and the control of peripheral devices.

Three robots are currently available in the laboratory: a Unimation PUMA 260, a Microbo ECUREUIL, and an IBM 7565 Manufacturing System. The IBM system incorporates a gantry-type robot for light assembly and is programmed in its native language, AML. The PUMA and ECUREUIL robots are both have six degrees of freedom; the configuration of the PUMA is anthropomorphic whereas the ECUREUIL is cylindrical. Both the PUMA and the ECUREUIL share the same programming environment, RCCL, which replaces their respective native robot programming languages, VAL and IRL. RCCL, a library of robot control primitives written in C, was originally developed at Purdue University [Hayward and Paul 83] and has since been enhanced and adapted for our research environment by Lloyd [Lloyd 85].

1.5.1 UNIX 4.2BSD Interprocess Communication Facilities

1.5.1.1 Socket Communication

The standard UNIX 4.2BSD distribution includes a Transport Layer protocol called Transmission Control Protocol/Internet Protocol (TCP/IP) which provides a low level network IPC interface [UCB 83], [SRI 82], [Tuthill 85]. All computers on the CVaRL

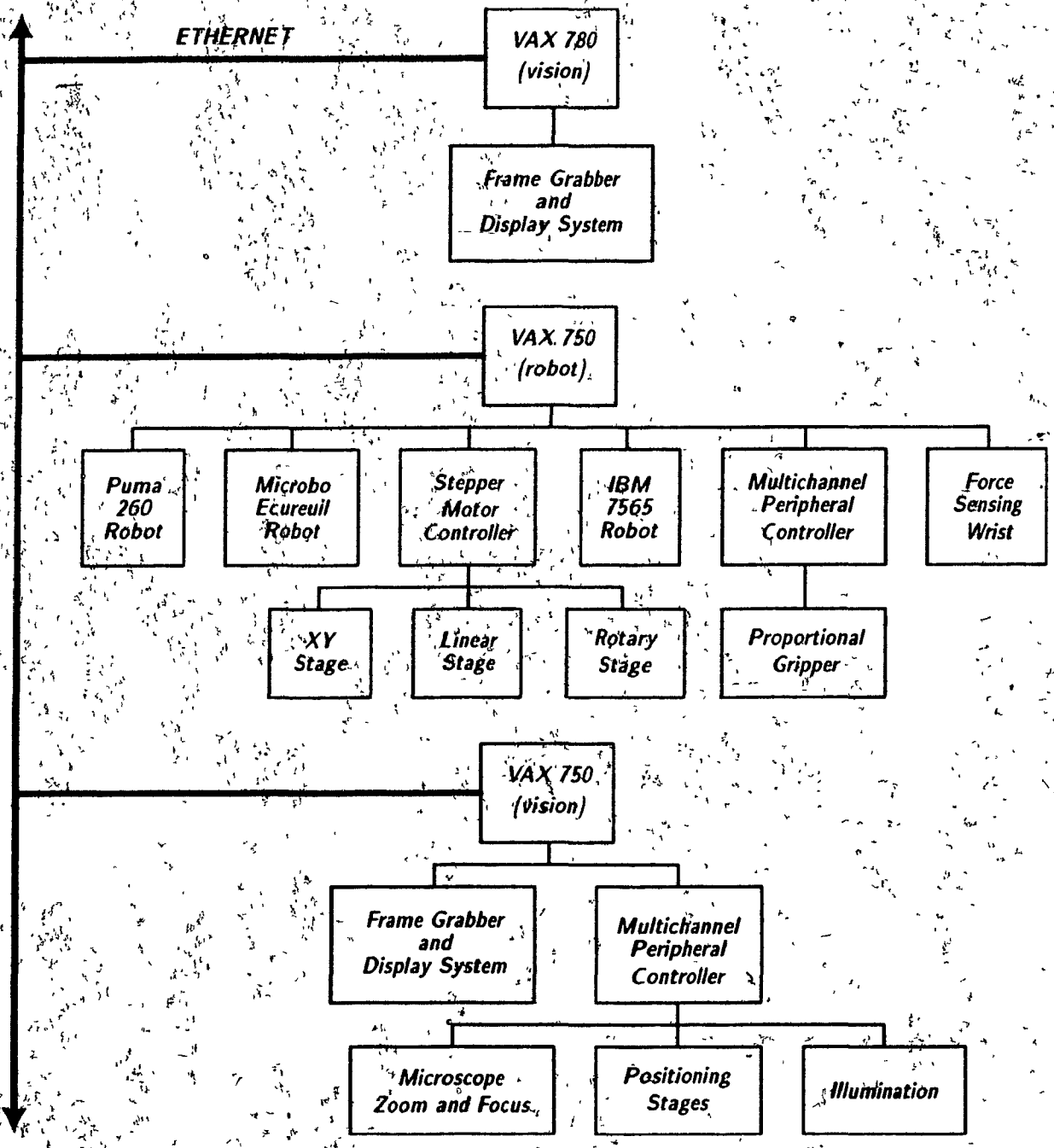


Figure 1.2 A typical CVaRL robot work-cell configuration

Ethernet support the TCP Transport Layer protocol. In this section, we will provide a cursory review of some of the more important TCP facilities and demonstrate that although they are complete, they do not support interprocess communication at a sufficiently high user interface level.

UNIX 4.2BSD supports several communication *domains* which represent different IPC environments. Within each domain, communication links may be established between endpoints known as sockets. Sockets are classified according to their *abstraction* type, which specifies the semantics of communication. In most applications there are two useful types of socket services offered within the *internet* domain, *datagram* and *stream*. Datagram services are connectionless; the message delivery is not guaranteed to be sequenced in the correct order, reliable, or unduplicated. Stream sockets provide reliable, flow-controlled, full duplex data transmission over virtual circuits. A virtual circuit refers to a communication link that may or may not be a physical channel between two processes. Distributed robotics demands reliable message transfer; so we will henceforth restrict our considerations to stream type sockets.

Sockets are classified as either active or passive, and may be either connected or disconnected. All sockets are created active and unconnected, by the UNIX function *socket*. A socket is made passive by binding it to a port identifier through the function *bind*. A pair of unconnected active and passive sockets may be connected, establishing a reliable, flow-controlled, full duplex communication link referred to as a virtual circuit. Connections are initiated from active to passive sockets by the function *connect*. Passive sockets *listen* for and *accept* connection requests from active sockets. Data transmission between pairs of connected sockets is conducted by *read*, *write*, *recv* and *send*.

A socket appears as a file to the operating system. When a socket is created, a socket descriptor, which is similar to a file descriptor, is returned. In establishing a connection from an active to a passive socket, the active socket is referenced by this descriptor while the passive socket is referenced by its port identifier. This implies that the process requesting the connection must know the full port identifier, that is, the concatenation of the host identifier with the TCP port, which has been assigned to the passive socket. When a passive socket accepts a connection, it actually creates and accepts the connection at a duplicate socket; a descriptor of this duplicate is returned. The original passive socket remains open and may listen for and accept additional connection requests from other ac-

tive endpoints. Thus, more than one apparent connection may be established to a socket. Data is transmitted or received at a connected socket by referencing its socket descriptor.

1.5.1.2 Pseudo Terminal Communication

In addition to the socket oriented approach to interprocess communication, UNIX 4.2BSD provides support for a device-pair termed a pseudo terminal. A pseudo terminal is a master/slave pair of character devices. The operating system views the slave device as it would any other terminal; all system software for controlling terminal device drivers can also be applied to the slave device. Data sent from a slave device serves as input for the master device and vice versa.

Pseudo terminals are intended for *conversational* computing between programs that require terminals for standard input and output. Processes communicate between pseudo terminals over the network, through sockets. Pseudo terminals are normally used for Application Layer programs such as remote logins (rlogin), file transfers (ftp), and user-to-user phone facilities (talk).

1.5.1.3 Justification for a Higher Level Network Interface

It is apparent that the UNIX 4.2BSD IPC facilities provide an adequate means of achieving interprocess network communication. However, attaining efficient network communication in a distributed processing environment is not a trivial task with these functions alone. Many programmers find the TCP IPC primitives difficult to understand and to work with. Programs using the TCP network primitives can sometimes be long. It is not uncommon for a program which implements the communication architecture of a single work-cell application, incorporating two robots, a vision system, and positioning stages, to require over one thousand lines of program code. Furthermore, some applications demand a knowledge of system level programming to be efficiently implemented.

These problems are compounded by the complex nature of the IPC communication structures that are often associated with sophisticated robot work-cells. Couple

this with the problems inherent with debugging extensive code that spans more than one machine boundary and one has to question the adequacy of the native communication programming environment.

The Session Layer and NIL promote efficient network communication by providing the user with a set of higher level network functions, thus avoiding the low level system primitives based on sockets. Users do not have to concentrate their efforts on the fundamentals of interprocess communication.

Chapter 2 Interprocess Communication and Distributed Computing

2.1 Introduction

In this chapter, we review some of the approaches to interprocess communication from the general perspective of distributed computing environments. We describe the complex nature of distributed systems and how communication is achieved. Two general categories of communication mechanisms, synchronous and asynchronous, are reviewed and examples of each mechanism are presented. Layered communication structures and the ISO's Open System Interconnect Basic Refence Model, an emerging standard for hierarchical architectures, are also discussed.

2.2 Distributed Systems

A distributed computer system is a collection of processing elements which are physically interconnected, controlled by system-wide resource managers, and capable of executing application processes in a coordinated manner. Coordination is accomplished through communication and synchronization of the processes associated with each processing element. Factors which have influenced the development of distributed systems include performance, processing throughput, flexibility, reliability, resource sharing, decentralization, and growth potential [Joseph 74].

Interprocess communication is a complex issue which is compounded by the fact that there is no unique approach to communication or synchronization. This problem is a result of the wide variety of hardware and software configurations which are possible.

The type of communication implemented is usually dependent on the amount of coupling present, that is, the degree to which the system's processors share resources. The two extremes of processor coupling are *tight* and *loose*.

A tightly coupled distributed computer system is characterized by its reliance on shared memory as an intermediary, with no facility for direct message transfer between processes, a single common operating system which coordinates and synchronizes interactions between processors; some degree of resource sharing; and processing power divided equally amongst the systems autonomous processors [Fathi et al. 83]. Such systems have been built around *concurrent* programming languages such as Concurrent Pascal [Brinch-Hansen 75] and Modula [Wirth 77], which assume that programs execute on a single processor and communicate through shared variables.

This is in contrast to loosely coupled systems whose processors communicate without the benefit of shared memory. Loosely coupled systems are composed of autonomous computers which employ message based mechanisms, which are effected in accordance with network communication protocols, and must therefore contend with such issues as network access, communication protocols, and remote procedures. These systems are often controlled by *distributed* operating environments which incorporate communication interfaces such as that provided by BSD4.2 UNIX [Tuthill 85]. The early context of this work was *data transfer* over long-haul networks. It has since evolved towards distributed systems and local area networks (LAN's). Both concurrent and distributed environments seek to hide communication details from the user, making remote operations resemble local ones.

The requirements of a robotic work-cell can be viewed from the perspectives of concurrent and distributed software as well as tightly and loosely coupled hardware. A

loosely coupled system offers greater flexibility. However, some elements in the work-cell must work together in a tightly coupled way, perhaps requiring specific kinds of interprocess communication in order to meet special constraints on speed and reliability. Since it is desirable to present a simple and uniform programming interface which would hide the underlying aspects of communication, selecting the appropriate IPC approach is often difficult.

2.3 Synchronous and Asynchronous Communication

Interprocess communication can be classified as either *synchronous* or *asynchronous*. A synchronous protocol is one whose communication primatives employ blocking techniques to regulate the flow of traffic. A primative is said to be blocking if once a message has been transmitted, the source process is suspended until an acknowledgement has been received from the destination. This is analogous to the function call implemented in most programming languages. Blocking techniques are most often used for command/response or query/status signalling [Silverman 84].

Blocking protocols have their advantages as well as their disadvantages. Since only one message can be in transit at a given time in a synchronous system, extensive message queuing at both the source or destination is not necessary, affording simplicity in its implementation. However, blocking protocols can be inefficient in that they restrict parallel processing. When a process sends a message, it must wait until the entire message has been transmitted and the remote process has acknowledged its receipt. Since distributed components are not always in close proximity, the time to transfer the message could be significant. Timeout features must be included in the send and receive primitives to prevent indefinite blocking should the destination program be unable to respond. Programming languages such as ADA [Baker et al. 85] and CP [Mao and Krishnamurti 80] support a 'disconnect' mechanism that allows the programmer to control the acknowledgement delay.

Another model of synchronous message passing is the Remote Procedure Call (RPC) [Birrell and Nelson 83], [Holmgren 85]. The RPC model allows user programs to

invoke 'remote' procedures in a manner which is similar to that for local procedure call models. When invoking the local procedure call, the caller places arguments in a procedure call buffer, control is passed to the procedure, and then eventually returned to the calling procedure. At that point, return parameters are extracted from the procedure call buffer. With the remote model, RPC services manage message passing to and from user processes by incorporating dedicated 'slave' servers that perform specialized IPC services for 'master' user programs. The user therefore avoids direct use of the low level system primitives for communication. An RPC is like most system function calls, invocation blocks the user program until the procedure returns, as shown in Figure 2.1. This decreases the power of the computing environment somewhat by removing the possibility of concurrent sender/receiver activities. Despite this shortcoming, RPC allows the user to approach distributed programming in a very simple way.

As discussed in [Andrews and Schneider 83], one can also provide a communications facility by embedding special IPC primitives in a programming language. A number of attempts have been made to to design general purpose distributed programming languages with IPC primitives that use the Remote Procedure Call model. Three examples, ADA, PLITS, and CSP, are compared in [Shatz 84].

Synchronous protocols are suitable for tightly-coupled distributed programming environments, but are often excessively restrictive in loosely-coupled concurrent environments. Additional flexibility can be provided by asynchronous or *message passing* protocols [Stankovic 82]. Under asynchronous protocols, the sending process does not have to wait for the receiving process to acknowledge the message, suspension of programs pending acknowledgement of transmitted messages is therefore not required. However, this might necessitate the queuing of message transmissions. The extra complexity can be buried deep within the operating system or hidden from the user through the use of dedicated servers. Nonetheless, asynchronous IPC tends to demand a higher level of programming sophistication from the user than synchronous IPC. For example, the user must anticipate the possible blocking of transmissions at the source or destination.

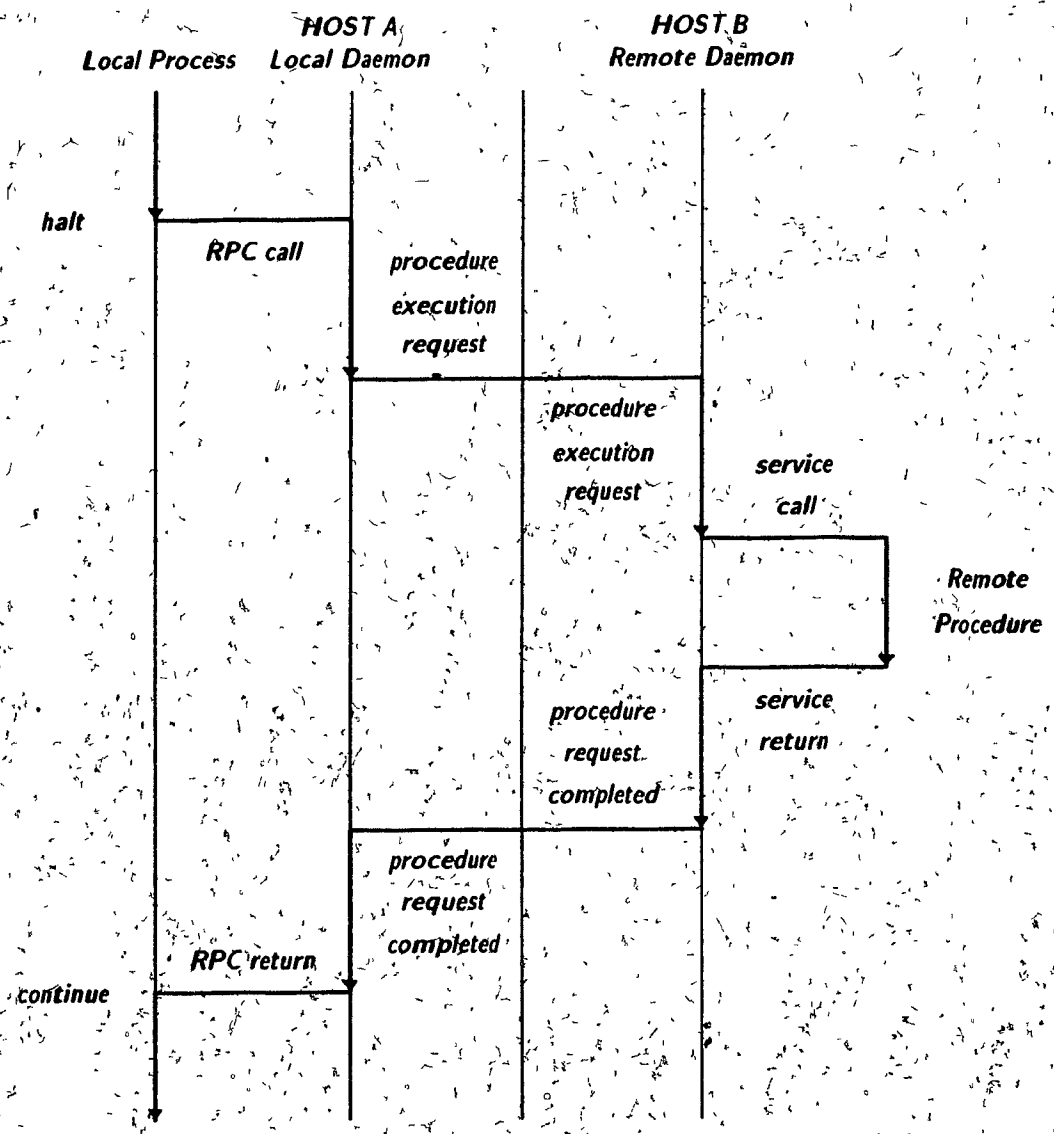


Figure 2.1 Remote procedure call sequencing

The simplest model of asynchronous user/server interaction is *dialogue*, whereby the user 'requests' services which are 'granted' by the server. This model differs from the Remote Procedure Call in that a master/slave relationship is not established and thus, the server may not always be available. Of course, the dialogue model presupposes a facility for reliably sending and receiving messages. In general, we find two kinds of message passing services: the *datagram*, and the *virtual circuit*. Datagram services are connectionless; a

connection is established temporarily, only for the duration of the message transmission. Following transmission, the communication link is broken. Datagram message delivery is not guaranteed to be sequenced in the correct order, reliable, or unduplicated. This is sometimes called 'transaction-oriented' communication, because each datagram represents a complete and independent transaction.

In contrast are virtual circuit services, which maintain the communication link until the user decides to terminate it. Stream sockets provide reliable, flow-controlled, full duplex data transmission over virtual circuits. Each transmission is acknowledged by its recipient. In the event that an acknowledgement is not returned to the source within a fixed timeout period, the transmission is automatically repeated; thus, message delivery is guaranteed. This requires that transmissions must be buffered at the source and that complicated error detection and recovery protocols must be in place. The associated overhead results in a slower service than that provided by datagrams. Virtual circuit communication services are sometimes referred to as 'end-to-end' or 'connection-oriented'.

Another user/server model of asynchronous communication is the *mail* service. This service represents a general class of IPC models that are frequently encountered in tightly coupled systems which have multiple processors utilizing shared memory [Faro et al. 85]. Programs exchange information in the form of mail messages which are sent to and retrieved from assigned mailboxes in common memory, as illustrated in Figure 2.2. Associated with each processor is a send and a receive mailbox which resides in a bank of common memory; that is, memory which is shared by all processors on the system. A process sends a message to a local server which packetizes the data and transfers it to the appropriate receive mailbox. The destination processor is signalled when the message has been deposited. It responds by issuing a receive command to a local server which then fetches the message.

Mailing can appear to be very responsive but it actually involves significant operating system overhead. Incoming and outgoing mail queues must be managed; reading a message involves reviewing the headers of numerous irrelevant messages until the required

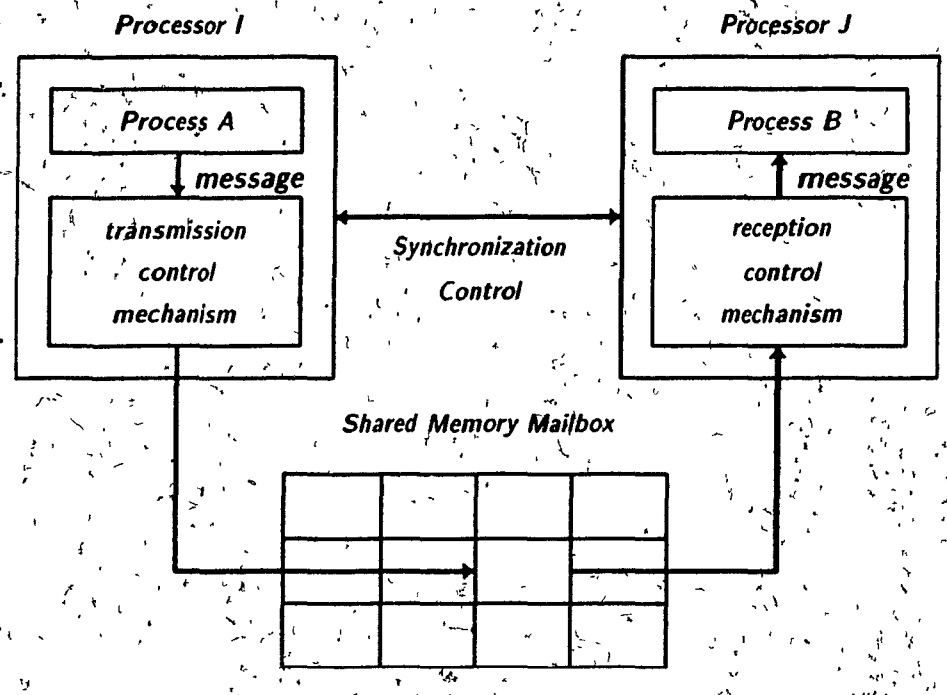


Figure 2.2 An example of a mailbox architecture

one can be located; some systems packetize the data; and the problem of common memory contention must be addressed. Typically, only one processor can access the common memory at any one time, creating a bottleneck. Even when multiport memory is used, contention is still a serious problem. This problem is further aggravated by the fact that when transferring data between processors, the common memory must be accessed both when depositing and retrieving the message. This tends to restrict the number of processors which can be effectively supported by a single operating system.

Shared memory systems that are implemented in loosely coupled network systems impose a significant strain on the network as well as higher operating system and computing overhead than that of bus oriented systems. An example of a shared memory network system based on XENIX is discussed by Devarakonda [Devarakonda et al. 85]. The implementation mechanism parallels that of the bus systems but there is the addi-

tional burden of message packetization, error detection, and correction that is inherent to the network transport protocols.

2.4 Layered Communication Standards

Layering serves to decompose the mechanism of communication into smaller sub-tasks. Each layer provides services to the layer above and below it, in a way which is independent of how the services are performed. This permits the independent implementation of each layer, thus making it possible to integrate products from different vendors. By classifying communication services into standardized layers, modifications can also be made to one layer, for example, to improve efficiency or speed, without affecting the operation of other layers.

However, not everyone agrees with the layered approach. Detractors warn of excessive communication overhead, and point to functions, such as flow control and error detection, which are duplicated in each layer [Clark 82]. High level protocols must sometimes undo the error recovery of lower levels, as for example when a message must be completely retransmitted despite retransmission of message fragments.

Opponents to layering believe that better performance can be achieved by building IPC into a distributed operating system which is supported by special software resident on each host; the software is sometimes bundled as separate 'servers', or simply incorporated into the kernel itself. The actual exchange of messages, usually of fixed length, is carried out using a simple data link layer or network datagram protocol. Others have added more sophistication to this basic service by providing explicit management of virtual circuits. User processes obtain IPC services by sending system requests to the kernel over a link which was dedicated to that process when it was first created. The underlying network service guarantees delivery of all messages.

As a rule, all the designs described thus far ignore the actual contents of messages. However, there are some designs which provide different types of communication

services for different kinds of messages. For example, a distributed system might offer datagrams, virtual circuits, and a special 'expedited' service for high priority messages. In addition, the message type could dictate whether synchronous or asynchronous protocols are employed. Some designs automatically provide different IPC services on the basis of message type [Andrews 82]. Rather than leaving the selection of the IPC mechanism to the user, a set of message *types* is provided and each one is passed in a different way.

2.4.1 The Open System Interconnect Basic Reference Model

The Open Systems Interconnection (OSI) Basic Reference Model of the International Standards Organization is a strong contender to become *the* standard hierarchical communication framework [Zimmermann 80]. [ISO 7498]. The OSI model originated in Europe where it quickly gained widespread acceptance. In North America however, IBM's System Network Architecture [Hoberecht 80] is the accepted standard because it provided a means for integrating IBM's mainframe computer products.

The OSI model breaks the major functions involved in communicating between two systems into an orderly sequence of seven layers [Tanenbaum 81]. Standards specify the services and protocols of each layer. The Basic Reference Model, defined by ISO 7498 and adopted in 1984, is shown in Figure 2.3.

The Physical Layer [ISO 8802.4] is the lowest level of the Reference Model. It specifies the electrical and mechanical aspects of the communication hardware as well as the functional control of the data circuits. This layer activates, maintains, and deactivates the physical connection.

The Data Link Layer [ISO 8802.2] establishes, maintains, and releases data links. It also performs point-to-point error and flow control of frames.

The Network Layer [ISO 8473] is responsible for network routing, switching, segmenting, blocking, error recovery, and flow control of packets. It is in this layer that

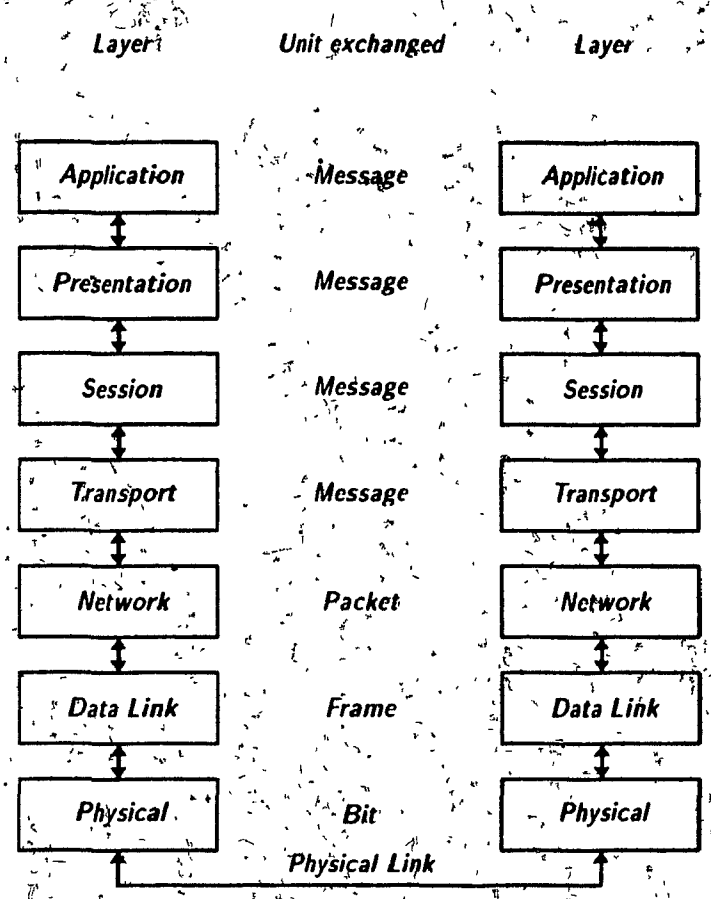


Figure 2.3 The OSI Basic Reference Model for layered communication

the network traffic is managed by either relaying data up to an application process or down along a physical network path.

The Transport Layer [ISO 8072], [ISO 8073] furnishes the base for high-level protocols by effecting data transfer and end-to-end reliability control. This layer provides various communication services to the Session Layer, multiplexes messages over logical connections, and segments data for the Network Layer.

The Session Layer [ISO 8326], [ISO 8327] establishes and terminates logical links between processes and manages the dialogue over those links. An important function of this layer is to synchronize data between the application processes.

The Presentation Layer [ISO 8649] is responsible for the interpretation and

manipulation of structured data. It ensures that data is converted to a form that the system can understand. This could include data encryption or conversion of data to device dependent codes.

The highest level of the Reference Model [ISO 8571] is the Application Layer. It is intended for user application and management functions such as file transfers, virtual file systems, and virtual terminal emulators. It should be noted that user programs are not considered part of this layer; they reside immediately above it.

Numerous layered communication architectures have been proposed in an attempt to address the problem of standardization of interprocess communication. Several standards, including the ISO's OSI model, IBM's Systems Network Architecture (SNA), the US Department of Defense's ARPAnet, CCITT's packet switching protocols such as X.25, and the IEEE's 802 local area network standards, are compared by Voelcker [Voelcker 86]. The OSI, SNA and ARPAnet standards specify more or less the same functions but are partitioned differently; the X.25 protocol specifies the Network, Data Link, and Physical Layers while the IEEE 802 protocols specify only the Data Link and Physical Layers.

2.5 Summary

It is obvious that interprocess communication in a distributed environment is complicated by the number of different types of communication that can be considered. Distributed systems employ different types of communication depending on their architecture. Shared memory and mailbox type systems are best suited to tightly coupled synchronous systems, remote procedure calls and message passing are best suited to loosely coupled systems. The layered architecture adopted by the ISO provides a highly desirable framework for communication in a loosely coupled distributed processing environment. It is generally accepted that a hierarchical, layered architecture is desirable for research as it provides the most flexibility; layering makes it convenient to experiment with different levels of the implementation.

Chapter 3 Interprocess Communication and Distributed Robotics

3.1 Introduction

A manufacturing work-cell should ideally consist of one or more robots, programmable workpiece positioning devices, sensory elements such as vision and haptic systems, and a programming environment which includes facilities for interprocess communication both within and between work-cells.

In this chapter, we discuss three basic approaches to interprocess communication in a distributed robotics environment. We begin with a discussion of *programming language* approaches to robotic IPC, followed by system developments aimed at *network* systems, and end with a review of *backplane bus* systems.

3.2 Programming Language Approaches to IPC in Robot Work-Cells

Most programming languages developed for industrial manufacturing applications fail to address the twin issues of communication between work-cells and communication within work-cells [Gruver et al. 83], [Bonner and Shin 82]. This makes interfacing the necessary intelligent sensor and actuator systems of the work-cells very difficult. Until recently, most commercial robot controllers provided only very primitive communication facilities such as parallel and serial ports which served as interfaces for binary input/output,

a display terminal, and perhaps a teach pendant; and no host controller interface [Jarvis 84], [Nagel 83].

This situation has become inadequate for two main reasons. As programming tasks become more difficult, there is a need for a richer programming environment than that which is provided by the native robot controller. Secondly, the need for more flexible or intelligent behavior has meant the incorporation of the robot itself into a larger framework under the control of other hosts. These systems are sometimes referred to as *flexible manufacturing systems*.

The simplest solution to the problem has been the development of special software to link an external host computer to the robot controller through its terminal port. This enables robot software to be developed on an external host and then downloaded to the robot's controller via an RS-232 or remote control link. Examples of wireless links based on infrared frequency shift key signalling have been presented by Dwivedi and Pearson [Dwivedi 83], [Pearson and Green 84].

Alternatively, more complex designs with varying degrees of supervisory ability have been implemented, based on dedicated host controller links. For example, terminal emulation programs which send commands to the robot controller in the robot's native language are described by Carayannis and Michaud [Carayannis 82], [Michaud 85]. These programs serve to make the robot an integral part of the distributed environment, and also make it possible to program the robot, using cross development facilities, in languages other than the native one of the controller. An advantage of working on an external host system is the availability of powerful debugging and simulation facilities.

Some designs have sought to bypass the native programming environment entirely by directly controlling the joint servos. We will now review some of these programming language efforts.

A set of process level communication and synchronization primitives for an integrated multi-robot system is described in [Shin and Epstein 85]. Five categories of

industrial processes are identified: independent, loosely coupled, tightly coupled, serialized motion, and work coupled; specific communication strategies are developed for each. Each process manages a set of tasks, a task being the smallest element of control activity. *Independent* processes can simply send state update messages to tasks they have in common. With *loosely coupled* processes, the actions of one depends upon the other. Thus, they must query each other, or a common task, for state information. *Tightly coupled* processes use a master/slave control approach: the slave always acts in accordance to the master, and returns a status message after each directive. In *serialized motion* processes, one or more processes must sometimes be performed before another can begin. Communication is used for synchronization and event signalling. Finally, with *work coupled* processes, each one must maintain a state description of all the others; once a process completes a task, it must broadcast the state change and then wait until all state descriptions have been updated before resuming execution

A modular command language for industrial robots, called LMAC, provides message passing facilities for communication and synchronization of processes in a multi-robot industrial system [Bonin et al. 83]. To provide the communication between processors, a LAN is structured as a four-layer hierarchy: network, transport, session, and application. LMAC has been implemented on a set of LSI/11's with the programming language PASCAL.

A dialect of PASCAL with new IPC primitives for robot programming is described in [Baird et al. 84]. The target work-cell consists of a single PDP/11 control computer linked to two robots and two vision systems via RS-232 serial lines. Communication between the controller and a device is modelled as an asynchronous series of commands and replies. There are also language primitives to allocate and deallocate the devices. The actual IPC within the controller is based on the message passing primitives provided by the RSX operating system.

Programming of robot based manufacturing cells using ADA is discussed in [Volz et al. 84]. They identify issues concerning the software aspects of robot cell control

and how ADA addresses these issues. Included in their discussion is ADA's ability to manage large complex software systems, the efficiency of code produced for real time applications, multitasking, interprocess communication and task synchronization, portability, and program debugging in a real time environment. A distributed run time ADA package called DARP, which extends communication and task synchronization across machine boundaries, is currently under development [Fisher and Weatherly 86].

Instead of extending a base language, one can also construct a procedure library to perform specialized services. One example of this is RCCL, a library of C functions for robot programming [Hayward and Paul 83], [Lloyd 85] A servo program resident in the kernel of the supervisory computer delivers setpoints to the individual joint controllers at prescribed intervals. Motion *requests* from the user program are translated into joint motions and then queued for transmission over a buffered high speed parallel link.

Finally, we review two *second generation* robot programming languages which have recently been developed. These designs attempt to present a truly integrated system level design. AML [Taylor et al. 82] is IBM's answer to intelligent robot programming: all sensing and control is managed by a Series/1 computer. Unimation's VAL II [Shimano et al. 84] provides extended facilities for network communication and sensor interfacing. The robot controller has a port dedicated to supervisory control over a LAN. There is an additional high speed serial port for online modifications to the planned path of the robot from an external computer. When the 'external alter' mode is active, the system sends messages to the external computer once every control cycle, to enquire if changes should be made to the planned path. VAL III is expected to be released in late 1986. It is reportedly compatible with such networks as MAP, SECS III, and Intel's Bit Bus and has facilities to integrate a high speed vision system [UNIMATION 86].

3.3 Network Oriented Approaches to IPC in Robot Work-Cells

A Session Layer for a LAN intended for robotic manufacturing cells has been described by [Bruno et al. 84]. MODIAC consists of clusters of Z8001 microprocessors

linked by a high speed serial bus. A distributed operating system called MODUSK [Garetti et al. 82] manages activities within each cluster through shared memory. A Virtual Network Operating System is planned for inter-cluster process coordination.

The concept of an *activity controller* for a multiple robot work-cell is developed in [Maimon and Nof 83]. The activity controller serves to sequence as well as synchronize the actions of multiple robots sharing tasks and auxiliary devices, such as feeders, in an assembly work-cell. One application is presented in [Maimon 85], wherein two robots and auxiliary devices execute asynchronous concurrent tasks; both robots are controlled from a VAX 11/780 host computer running UNIX, and are programmed in C.

A network for distributed robotics, based on three host computer systems linked by an Ethernet [Metcalfe and Boggs 76] LAN is described in [Goldwasser 84] and [Goldwasser and Bajcsy 83]. An expert system resides on each host and is responsible for vision, control, or haptic functions. The haptic expert provides an interface to a robot equipped with a sophisticated articulated hand. The hand consists of three fingers, each with two joints and three tactile pads. For each finger, separate processors are used to drive the joints and process tactile information. Communication among the experts is hierarchical. Each level can issue orders to experts on the level below and pass responses back to the one immediately above. There is also a facility for high priority message passing.

In the context of LAN oriented developments, a network interface designed for supervisory control associated with Unimation's VAL II robot controller has been described by Shimano [Shimano et al. 84]. The interface has three layers: the 'bottom', which performs physical input/output, the 'middle', which serves to multiplex multiple messages for transmission over a single link, and the 'top', which associates logical addresses with input/output devices. These layers do not directly map onto the OSI reference model; for example, the 'bottom' layer is a combination of Physical and Data Link layers.

A layered approach to networking within a robotics environment based on the OSI model has been described by Faro and Messina [Faro and Messina 83]. [Faro and

Messina 84]. Within a work-cell, they suggest a simplified message passing facility based on shared memory. A transport layer is maintained for associating devices with addresses, and a session layer is proposed to provide high level communication services such as opening and closing of channels, and managing data transfer over the channels. A single host within each work-cell is designated 'master' and serves as a gateway to the supernet which is based on redundant busses. Overall, three kinds of modules are identified to support robot cooperation within and between work-cells: one for message passing, one for remote job execution, and one for file management. Of the three, only the message passing module would be required everywhere.

The concept of networking work-cells within a factory and the Factory Area Network (FAN) are explored in [Holland 83]. The author identifies a hierarchy of communication needs which parallel the hierarchy of devices within the factory. At level 1, the device level, messages are exchanged over dedicated point-to-point links for binary sensing and servo control. At level 2, messages are used for synchronization and data exchange between device controllers. Ethernet LANs are employed at the physical level to effect message transfer. At level 3, the work-cell level, message passing is in response to operator commands such as production monitoring and data logging. Finally, at level 4, we are concerned with factory wide data processing.

One example of a FAN is Allen-Bradley's Distributed Network Architecture. A single *data highway* or LAN links distributed devices on the shop floor such as computerized numerical control (CNC) and robot stations, and gateways are used to tie networks of supervisory subsystems to the data highway. A flexible manufacturing work-cell based on Allen-Bradley equipment is described in [Hanlon and Weston 82]. Two programmable controllers are used to manage a robot arm and a CNC machine, and they are linked by a data highway. But the data highway is used for more than communication within the work-cell; a remote file server, and a network supervisory station are also connected. The LAN is a high speed baseband serial bus, with a contention based protocol. When a station on the net desires to transmit, it must wait until the current network master has transmitted

all of its messages. All active stations are then polled; mastership of the net is granted to the one with the highest priority which has something to transmit. In fact, the polling is done twice: first, all high priority stations are polled, and then stations with 'normal' priority.

Most manufacturers of flexible automation still provide proprietary solutions to the communications problem, but this situation is changing. One recent development is MAP, a set of manufacturing automation protocols promoted by General Motors [Leopold 84], [MAP 85], [Kaminski 86]. MAP is a layered communication standard for linking distributed elements within a factory. The MAP specification is based on the communication structure defined by the Open Systems Interconnect (OSI) layered architecture [Zimmerman 80]. The OSI Reference Model was developed by the International Standards Organization (ISO) to provide a framework for connecting open systems (an open system is one for which standards are published), allowing system components from different manufacturers to be inter-connected.

MAP is rapidly becoming a *de facto* standard in factory automation, and has already gained widespread endorsement. MAP's physical transport system utilizes the IEEE's token-passing-bus network standard, although the Honeywell Plant Management System uses an Ethernet [Iversen 85]. Eventually, MAP will incorporate all seven layers of the OSI model. A pilot implementation which links 200 data acquisition units monitoring 10,000 sensors at the General Motors car assembly plant in Oshawa, Ontario is described in [Storoshchuk and Szabados 83].

The token-passing-bus media access control is also used by IBM's SNA standard and is favoured over Ethernet in a pure industrial application as it is better suited to real time applications. The token-passing-bus protocol controls when an element can have access to the network. This is based on a predefined order which is established to correspond to real time events in the manufacturing process. This is in contrast to Ethernet, which relies on collision detection with randomized backoff, that is, all parties wait a random amount of time after a collision, to resolve network contention. Ethernet is not capable of message

prioritizing and is generally better suited to applications where real time performance is not required.

Despite MAP, Ethernet is still a very popular LAN technology. For example, many manufacturers of Automatic Test Equipment (ATE) such as Fairchild and Zehntel have developed Ethernet compatible networks to link programming, testing, and repair stations [Milne 83]. In a distributed robot system for assembly described by Barthes and Zavidovique, a vision system, a robot arm, and a planning module communicate via message passing over a custom Ethernet LAN [Barthes and Zavidovique 81].

A distributed approach to complex robot systems based on message passing is described in [Harmon et al. 84]; the target problem domains are an experimental autonomous vehicle and an automated welding work-cell. The authors define three classes of subsystems: *sensors*, which translate raw data into symbolic information, *controllers*, which translate symbolic plans into commands, and *knowledge bases*, which 'reason' about symbolic information and prepare plans. These subsystems are connected by a LAN in a bus topology, and interact by passing messages with specific formats. Each message has a source, a priority, and a body. The body can be of type *plan*, that is, a command, or of type *report*, which is a response. A more elaborate message formatting scheme is described in [Harmon 83] and [Harmon and Gage 80], in connection with a layered robot to robot communication. A *message content* layer formats information to be communicated using three fields. The first is called the *address*, which contains the identifiers of the source entity and one or more destinations. The second field is called the *content*, which contains the message itself and a context for the message. The third field, the *message state*, contains message parameters indicating message type, priority, and the 'state' of the source entity.

3.4 Backplane Bus Oriented Approaches to IPC in Robot Work-Cells

In contrast to loosely coupled networks are tightly coupled multiprocessor common bus systems. Several examples of such systems are now described.

A log loading robot crane is described in [Kärkkäinen and Manninen 83]. A set of three 68000 microcomputer modules are configured as a sensor processing unit, a coordination unit, and a servo processing unit. They are linked by a VME bus and communicate via shared memory. The coordination unit serves to reconcile the sensory information with the desired goal state. The servo unit is the heart of the system, and it supervises the robot crane, a sensory subsystem, and the man/machine interface. The sensory subsystem is a daisy chain of sensors, each having a single chip microcomputer interface [Kärkkäinen 83]. The link to the servo unit is a high speed RS-232 serial line.

The design of another multi-microcomputer network for generalized robotics control, consisting of four 68000 microprocessors, is described in [Plessmann 83]. Data exchange between nodes takes place at the Data Link level over a custom parallel bus. Synchronization is performed via an interrupt scheme.

Another example is the CHORUS system, developed at the National Research Council of Canada [Green 83], which is based on the MULTIBUS. Communication is via message passing primitives supplied by the HARMONY operating system [Gentleman 83]. A sample application is described in [Elgazzar et al. 84] to locate, recognize, and then insert regularly shaped blocks into matching holes. The robot application task on one host and the vision application task on another host communicate for mutual synchronization and exchange of data.

A multi-robot real time environment with intelligent sensors requires an efficient communications mechanism which promotes parallelism, partitioning of tasks, and reconfigurability. Two parallel bus designs which meet these requirements are RAPIDbus [Willis and Sanderson 84] and REPLICA [Ma and Krishnamurti 84].

In [MacWilliams et al. 84], a high speed serial bus from Intel called Bitbus is described. The authors suggest how a Bitbus could be used to link related physical devices such as multiple joint servos, and then a Multibus to connect distinct but cooperating subsystems within a work-cell composed of robot and vision systems. Finally, they propose

separate work-cells be interconnected via a factory wide Ethernet.

In this context of tightly coupled systems, we also mention a project underway at Carnegie Mellon University to develop an autonomous land vehicle [Elfes and Talukdar 83]. A set of slave processors dedicated to drive train control and sensing systems for sonar, cameras, and proximity devices are linked to expert modules resident in a central processor. The experts share information via shared memory, but communication between a slave and its master is via message passing. This is managed by a small real time kernel resident in the each processor.

An industrial work-cell for intelligent assembly is described in [Stauffer 85]. One robot acts as a material feeder for two others which weld pads on ignition modules. A vision system is used to accurately locate the pads for welding. The material handling robot acts as the master; it communicates with the welding robots via eight bit parallel ports, and with the vision system via a serial port.

A simple robotics testbed for small scale laboratory experiments is described in [Wainwright and Moss 85]. The direction and speed of each robot joint servo is controlled by pulse width modulation, using a custom robot controller. Communication between the controller and the supervisory host is over an 8 bit parallelled bus.

In [Chen et al. 83], another small scale testbed is described, consisting of two robot arms with separate controllers and a master controller with a voice recognition front end. The master passes messages to the two arms via dedicated serial lines to their controllers.

An experimental testbed for coordinated robot and vision work, which is described in [Makhlin 82], is based on a Westinghouse vision system and an Olivetti robot with two arms. Communication between the two LSI/11 based systems is via high speed DMA message transfers. Extensions to the Olivetti robot programming language SIGLA to support vision related commands is also described.

A testbed at the Jet Propulsion Laboratory for generalized bilateral control of two robot arms is described in [Bejczy and Lee 84]. Each robot is controlled by a set of three National Semiconductor microcomputers which communicate with each other over a shared parallel bus.

A tightly coupled three level hierarchical network of sixteen 8086 based microcomputers is described in [Alford and Belyeu 84] for the coordinated control of two PUMA arms. The multi-arm *coordination computer* transmits new position commands to each robot via high speed block transfers to intermediaries called *prediction computers*. The prediction computers use a simple handshaking protocol to issue new setpoints to the robots' joint controllers.

3.5 Summary

Although the need to standardize communication mechanisms is apparent, reaction from the robotics industry has been slow and limited. A survey of relevant work suggests that few attempts attempts have been made; the survey presented was more a review of *research* activities than *industrial* efforts. Each of the approaches presented satisfies a particular aspect of interprocess communication in a distributed robotics environment. Depending on the system requirements, one has the choice of a programming language versus an operating system approach complimented with either a loosely coupled network system or a tightly coupled multiprocessor system. The literature indicates that the most favoured mechanism of interprocess communication is message passing entrenched in a hierarchical framework. The reason for this is probably because of the general purpose hardware and software configurations found in many laboratories. Our work is based on this approach and in the following chapter we describe an easy to use means of achieving efficient IPC in this type of environment.

Chapter 4 The Communication Environment

4.1 Introduction

This chapter presents the design strategy, architecture, and software implementation of a system for interprocess communication in our dist uted robotics environment. The system is designed to provide a useful mechanism for effecting interprocess communication amongst user and system programs controlling the activities within the robot work-cell.

Interfaces to the system are available at three levels through a series of functions that are called from the user's program. The top level is called the Session Level; it is followed by the Network Interface Level; and finally the Transmission Control Protocol (TCP) Level (Figure 4.1). Each level permits the same degree of interprocess communication to be achieved, but in a different way. The higher the level, the easier it is to implement and manage interprocess communication. The lower levels offer increased speed and flexibility. The average user would confine their program development to functions available in the top two levels. System level applications are generally implemented at the TCP level.

Sample programs demonstrating the use of the Session Layer and Network Interface Library functions to effect IPC amongst three processes distributed over two hosts are presented.

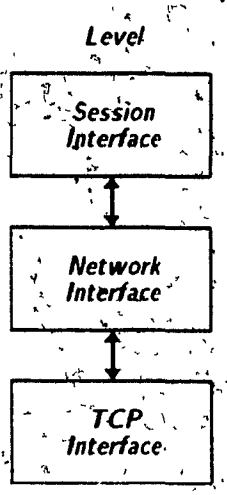


Figure 4.1 The three levels of the network interface

4.2 Design Goals

The communication environment has the following design goals:

- Flexibility: Since a research environment is dynamic, it is important that new equipment can be integrated easily and in a consistent way.

- Modularity: Robotic work-cells are composed of a number of distinct elements, each of which is responsible for its own subset of tasks required for the work-cell's operation. The model of communication used should complement the modular nature of the work-cell.

- Ease of use: The communication interface should be simplistic, consisting of a small number of easy to use functions that are independent of which work-cell elements are communicating. The user should be shielded from all aspects of the low level IPC mechanism.

4.3 Design Constraints

The following design constraints emerged from an analysis of our robotics environment.

Interprocess communication will be achieved at the level of the operating system rather than through a programming language. Although a language approach eases implementation of communication primitives and is generally more transportable than an operating system, we consider it unwise and impractical to restrict all programming to be carried out with a single language in order to achieve IPC. BSD 4.2 UNIX, which includes TCP, is well suited to this application as it supports the necessary low level IPC primitives to build a high level communications environment.

The physical link between distributed devices will be accomplished using a network approach rather than a common bus. A 10 MHz Ethernet will be used to link the work-cell elements. Although the backplane bus offers increased speed through shared memory and signalling, it represents a less flexible design. The backplane limits the number of connectable elements, they are restricted to a common location, and there are always problems with system integration. As VLSI pushes down the cost of equipment, it is becoming more practical to distribute intelligence within the work-cell and tie devices together with a network.

Like most of the network based distributed applications surveyed, we favor the user/server paradigm. In this scheme, each host incorporates a communication *server* that provides the user with set of communication primitives which are *independent* of the programming language being used. We favor *asynchronous message passing* over more restrictive alternatives such as the synchronous remote procedure call or mailbox type systems. Utility processes controlling the work-cell will reside on different host computer systems. IPC will be used to emphasize co-ordination of sets of parallel tasks, rather than simple data exchange. Users will logically structure their jobs; the vision programs will execute on one host, robot programs on other hosts, and so on. With this distributed

intelligence: one would expect, for the most part, that messages transferred between elements will be short and infrequent, keeping the amount of network traffic between entities low. Thus, the high bandwidth provided by a backplane bus system is not necessary.

The communication framework will be based on the ISO's Open System Interconnect Basic Reference Model. The Physical, Data Link, and Network Layers are implemented by the Ethernet; the Transport Layer will be provided by TCP; a Session Layer will be implemented; and the Presentation and Application Layers are inherent to BSD 4.2 UNIX.

The network's user interface will consist of a partial implementation of the Session Layer Model. We anticipate that by providing only the most useful features of the OSI model, we will achieve both simplicity, through a high level interface, and more efficient IPC. By adopting OSI principles, the design can take advantage of the applicable specifications and protocol algorithms.

In implementing the Session Layer, the integrity of UNIX will be preserved; the implementation will not involve any modifications to the operating system kernel. We recognize that this may have a negative effect on the communication performance characteristics but expect that this will be more than offset by a less complicated design and a shorter implementation time. In addition, activity within the work-cell is presently bound by the speed of the robots and positioning devices, as well as by vision processing. In our applications, the rate of IPC is at least an order of magnitude faster and is therefore not of prime concern. However, we would like to minimize the overhead associated with the Session Layer.

Working with UNIX means that the C programming language is the logical choice for the Session Layer implementation. We shall also follow UNIX conventions where possible, as for example with error handling, to maintain a consistent system interface to the user. The Session Layer will be compatible with any programming language, such as Fortran, Pascal, LISP or assembly, whose object modules can be linked with C.

The session layer design will be based on the virtual circuit service of TCP. We expect that message passing will be extremely reliable, since TCP itself is a robust protocol and since the LAN represents a relatively noise-free network. TCP was originally designed for the 'long-haul' packet-switched ARPAnet which is used to reliably transport messages to points around the world [McQuillan and Walden 77]. The need for transmission error detection and recovery procedures, in addition to those provided by TCP, is therefore not anticipated.

4.4 The Session Layer Design

4.4.1 Overview

The period of activity during which a connection is established between two or more *entities* or processes for the purpose of communication is called a *session*. The Session Layer provides the user with the first level of *sophisticated* interprocess communication services [Emmons and Chandler 83], [Neumann 83]. This is accomplished through a minimal set of high level functions which are available to user processes to facilitate network access by providing standardized mechanisms to establish, manage, and terminate the communication services provided by the Transport Layer.

The basic function of the Session Layer includes:

- mapping logical Session Layer addresses onto Transport Layer addresses
- providing a means for user IPC programs to be host independent
- establishing and terminating logical links between entities, and controlling access to do so
- managing dialogue over the established links

- providing a means of process synchronization
- supporting error detection and recovery

The Session Layer hides the low level aspects of communication from the user, appearing as a black box which links processes and acts as an intermediary in exchanging messages. Our Session Layer interface consists of communication *endpoints* and *links* between the endpoints. An endpoint is a Session Layer logical tap on the network which is uniquely associated with a user assigned name. The user references endpoints by specifying only their logical names; links are referenced by pairs of names, identifying source and destination endpoints. The Session Layer translates the logical names into physical network addresses, which are specially ordered concatenations of host addresses and network port identifiers. These addresses are expressed in a standardized form of external data representation that is necessary for different languages, operating systems, and machine architectures to be able to communicate. It is the responsibility of the Session Layer and not the user to identify on which host an endpoint resides; the user only has to identify the endpoint's logical name. An important consequence of logical addressing this is that machine independence of Session Layer processes is supported.

Because the Session Layer manages dialogue over the links, it can be used to synchronize entities residing in the Presentation and Application layers as well as in user programs. This is necessary in coordinating tasks within the work-cell.

A necessary function of the Session Layer is error detection and recovery; both inter and intra host error conditions must be considered. For example, the Session Layer must respond to the loss of a host on the network, so that links to that host may be carefully terminated while continuing to manage communication among the remaining hosts.

4.4.2 Session Layer Architecture

The Session Layer structure on each host is hierarchically organized into two

tiers: the top level is occupied by an Administrative Server and the lower level by Dialogue Server. The Session Layer is complemented by a set of Session Layer primitives which are linked to the user's application program. These functions bridge the servers and user processes, providing the user with a means of creating and deleting endpoints, establishing and terminating communication links between endpoints, and transmitting and receiving messages over those links. The Administrative Server is responsible for managing endpoints while the Dialogue Servers link the endpoints and coordinate dialogue over the links.

The Administrative and Dialogue Servers as well as the Session Layer primitives are discussed below. Additional technical details can be found in [Gauthier et al. 85].

4.4.2.1 Session Layer Administrative Server

The Session Layer Administrative Server is responsible for managing endpoint information. This includes maintaining records of current endpoints and sharing this information with servers on other hosts. In addition, the Administrative Server keeps the Session Layer viable through inter-host error detection and recovery procedures.

The Administrative Server maintains a table which is used to map user specified logical names to port identifiers of passive sockets. This table makes the port identifiers of passive sockets available to servers across the network, allowing the user to identify endpoints by their names only. The table is updated through the user program by either appending descriptions of newly created endpoints or deleting those that are no longer needed. This includes removing entries created by user processes that have subsequently terminated. These table entries, which we call 'abandoned', result from the user program aborting unexpectedly or terminating without first having removed endpoint descriptions from the table. Upon termination of a process, any associated TCP ports and open sockets are freed by the operating system; the corresponding table entries must be removed by the Administrative Server.

Separate Administrative Servers reside on each network host. Through inter-server communication, the tables are maintained to be identical, each containing entries

for endpoints across the network. Multiple servers offer speed and redundancy which helps guard the Session Layer against collapse in the event that a host becomes lost to the network, that is if it crashes or its network interface becomes disabled. The servers also exchange Session Layer status information and therefore play an important role in error recovery procedures.

Eight data fields are required to completely describe an entry in the Session Layer Administrative Table (Figure 4.2). The server uses the first three fields of the table, the logical name assigned to the endpoint by the user process, the TCP port number assigned by the operating system, and the name of the host on which the endpoint resides, to map the name of an endpoint onto its socket address. The socket address is the concatenation of the TCP port number with the host's Internet address, which is derived through a system call by the server based on the host's name.

The fourth field of the table, the user identification, specifies who owns the endpoint. The motivation for this will be described later.

In maintaining the Administrative Table, the server is responsible for ensuring that 'abandoned' endpoint entries are deleted. This requires the inclusion of the next two table fields: the identification number of the process that created the endpoint, and the endpoint type. The endpoint type may be either *reserved*, which identifies a permanent endpoint, or *user*, which identifies a endpoint whose entry should be removed from the table when its user process terminates. Reserved endpoints are intended to be associated with processes controlling work-cell utility modules. For example, assume that a system utility process exists that controls an XY stage and its endpoint's logical name is *stage.1*. A user would simply establish a connection from one of their processes to *stage.1* and control the stage by sending and receiving messages to and from it.

The remaining two fields are useful only to the Administrative Server that was involved in the creation of the endpoint. These fields specify the process identification number of the Dialogue Server, a child process of the Administrative Server which is ded-

icated to the endpoint, and the socket descriptor of the link between the Administrative and Dialogue Servers.

Endpoint Name	TCP Port Number	Host Name	User Id	Process Id	Endpoint Type	Dialogue Socket	Dialogue Id
robot	1029	curly	0	2343	reserved	5	2344
vision	1031	larry	120	2345	user	6	2346
inspect	1023	larry	120	231	user	7	2021
stage 1	1027	curly	120	345	reserved	64	2350

Figure 4.2 An example of an Administrative Table

The Administrative Servers safeguard against the duplication of names in the Administrative Tables. Prior to updating its table, the server must verify that the requested name is not already entered in the table. To eliminate the possibility of name duplication resulting from concurrent actions of Administrative Servers on distinct hosts, the servers adhere to a synchronous inter-server table update protocol. Only one server may initiate changes to the table at any time. A server requires permission from the other servers before proceeding with the addition of names to the table and must notify the other servers of the changes in order that their tables may be updated.

4.4.2.2 Session Layer Dialogue Server

The Dialogue Server is responsible for managing communication links, which includes establishing and terminating connections between endpoints, and coordinating dialogue over the links. In addition, the server communicates with the Administrative Server and other Dialogue Servers to exchange both Session Layer messages as well as status information.

A user program which consists of multiple processes may have several Dialogue Servers associated with it. Each process involved in IPC has a single endpoint, with an associated dedicated Dialogue Server. These Dialogue Servers act as intermediaries in

coordinating dialogue between endpoints. An example of the Session Layer architecture depicting an established session between a user process and a system utility process over a two host network is shown in Figure 4.3. The exchange of messages is mediated by the two Dialogue Servers.

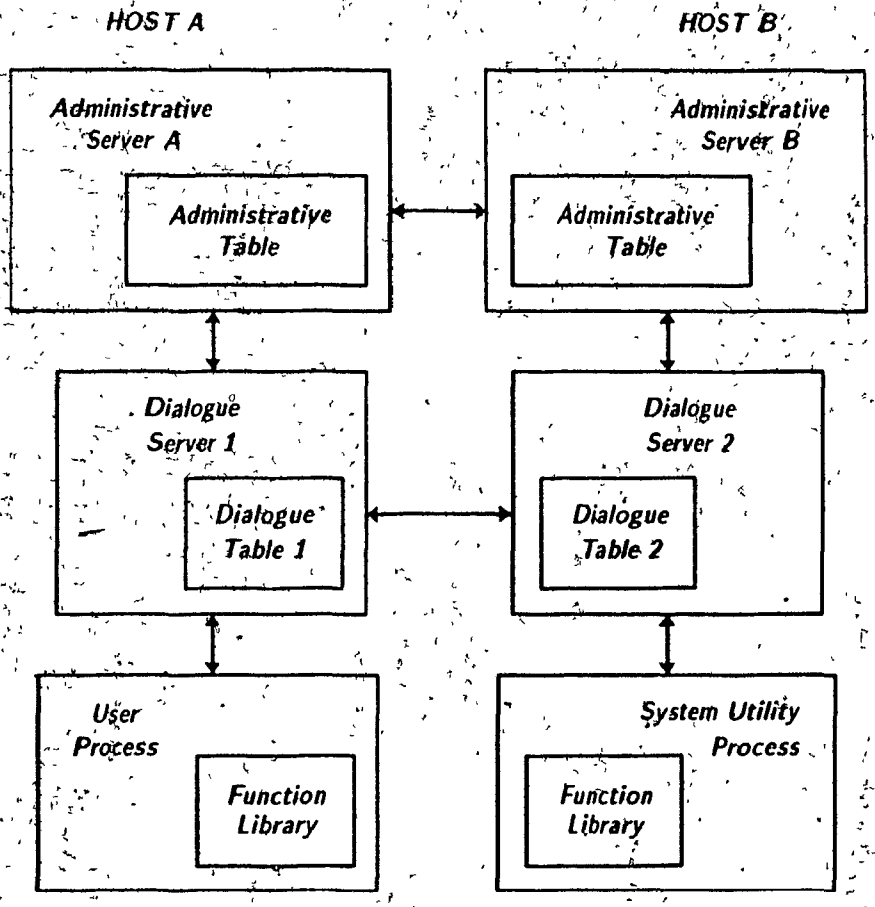


Figure 4.3 Session Layer architecture over a two host network

Each server maintains a Dialogue Table, shown in Figure 4.4, which describes the links established to the Dialogue Servers of other endpoints. Five data fields are required to describe a table entry. The first two fields, destination logical name and socket descriptor, map the name identifying the destination endpoint of a link onto its socket descriptor at the source endpoint. This information is used by the server to identify the link over which a message is to be sent or received. The last three fields, message pending, signal enabled, and signal number are used by the Dialogue Server to manage the reception of messages.

Message pending tells the server that it has read and interpreted the message at that endpoint to be data which will eventually be passed on to the user process. The signal enabled and signal number fields tell the server if the user process should be interrupted upon the receipt of a message and if so, which interrupt signal should be used.

Destination Name	Socket Descriptor	Message Pending	Signal Enabled	Signal Number
vision	7	0	1	28
lighting	6	0	0	29
XY stage	8	1	1	30

Figure 4.4 An example of a Dialogue Table

The Dialogue Server communicates with three different entities, the Administrative Server, the associated user process, and one or more other Dialogue Servers. Messages received by either Dialogue or Administratives Servers are always in the format of a command followed by any relevant data. Received messages must be read and parsed to determine the appropriate action and handling of any data that follows. For example, a message received by the Dialogue Server could contain status information from the local Administrative Server indicating that one of the network hosts will be going off line. The Dialogue Server would then inform the user process of the pending termination and perform an orderly shutdown of any affected communication links.

4.4.3 Session Layer Function Library

The Session Layer Function Library provides the user with an interface to the Session Layer Servers. The library is linked to the user program and executes in the user's process space. Functions are provided to create and delete communication endpoints, establish and terminate communication links between endpoints, and exchange messages over the links. The functions provide a useful high level network interface by building upon the IPC system calls provided by UNIX 4.2BSD. The user is not precluded from

circumventing the Session Layer and mixing lower level IPC function calls with Session Layer primitives. This might be necessary in instances where the speed of the Session Layer service is insufficient.

Some of the Session Layer functions must adhere to protocols to ensure that the various servers' tables residing on different hosts are properly updated and deadlock situations do not occur. These protocols are a consequence of the general nature of the Session Layer and are part of the penality that must be paid when such a general purpose architecture is adopted. In applications such as this, master/slave paradigms are often employed. Depending on the situation, these protocols may be symmetric or asymmetric. In a symmetric protocol, either side may be the master while in an asymmetric protocol, one side is always the master. Irrespective of the symmetry of the master/slave model, there is always an asymmetry in establishing the initial connection between the two processes. Under UNIX 4.2BSD, TELNET is an example of a symmetric protocol which is used for remote terminal emulation. FTP is an example of an asymmetric protocol that is used for file transfers [Sun 85].

4.4.4 The Session Layer Functions

The Session Layer functions can be divided into three classes: initialization routines, management routines, and termination routines. The Session Layer functions and their associated parameters are summarized in Figure 4.5. A sample application of the Session Layer in which three processes communicate over two host computers is presented in Section 4.4.5.

4.4.4.1 Initialization Routines

- **Init_session**

Init_session initializes a set of logical names in the Administrative Server Table. The Administrative Server is then responsible for ensuring that the logical names entered

Session Layer Functions	
Function Name	Parameters
init_session	list of logical names
create_endpoint	logical name of endpoint, maximum number of connections to endpoint, type of endpoint: reserved or user
establish_link	list of destination endpoints to link to, list of interrupt service routines
dispatch_msg	list of destination endpoints to send to, message buffer name
read_msg	logical name of source endpoint, message buffer name
delete_endpoint	—
terminate_link	list of destination logical names
exit_session	—

Figure 4.5 The Session Layer functions

in its table are unique across the network. The duplication of logical names as a result of concurrent actions by distinct servers must also be avoided. The following protocol achieves this by establishing a master-slave relationship between the Administrative Servers.

Prior to updating its table, the Administrative Server broadcasts a message to the other Servers requesting permission to perform an update. The broadcast is interpreted by the remote Servers as a temporary block on table modifications. The Administrative Server does not proceed with the update until all other Administrative Servers have acknowledged the request. Provided that the logical names specified by the user are unique, partial entries specifying the logical name and the identifications of both the user and the host are appended to the Administrative Table. The entries are later completed through with the *create_endpoint* call. The modifications are then broadcast to the other servers, the remote tables are updated and the update block is lifted.

Each server has a predefined priority which is used to resolve deadlocks. This

priority is known by all other hosts in the network. Should two or more servers simultaneously broadcast a request for permission to perform an update, the servers with the lower priorities will delay their requests. Administrative Servers receiving *table update* and *init_session* requests, while a block on updating is in effect, buffer the requests until the block has been lifted.

The protocol is most efficient when several entries are appended to the Administrative Table with a single *init_session* call. Once a server becomes the master, it can make all of its additions and the Administrative Table update protocol is invoked once. Endpoints not identified through *init_session* are initialized through *create_endpoint* and the update protocol must be performed each time *create_endpoint* is called; this increases the Session Layer's initialization time.

- **Create_endpoint**

This routine creates a single Session Layer endpoint. The user specifies the logical name of the endpoint, the maximum number of connections that can be accepted at the socket from other endpoints, and the type of endpoint, either *reserved* or *user*.

Users may specify permanent Administrative Table entries by declaring an endpoint type of *reserved*, otherwise it is of type *user* and may be maintained throughout the life of the process that called its creation. Reserved logical names are intended to be associated with work-cell utility processes such as controllable stages, sensors, or lighting.

When a process calls *create_endpoint*, a connection is is established to the Administrative Server via a well known socket port and the logical name of the endpoint is passed to the Administrative Server. The server in turn, verifies whether the user has been granted the use of the requested logical name through a previously made *init_session* call. This condition is signalled by a partial entry in the Administrative Table, referencing the logical name and the identification of its owner. If a partial entry is not present, it is

added through the *init_session* table update protocol. Each *process* participating in IPC may create only a single endpoint.

Once the uniqueness of the logical name has been verified, the Administrative Server creates the requested endpoint and enters all the pertinent information in the Administrative Table. It then creates a Dialogue Server which is dedicated to the newly created endpoint and possesses communication links to both the Administrative Server and the user process. The Dialogue Server listens at the newly created endpoint for connection requests from other Dialogue Servers.

- **Establish_link**

This routine establishes communication channels to other processes. Links are connected between the Dialogue Servers associated with the end processes, enabling interprocess communication via the servers.

The user provides a list of logical names of destination endpoints associated with other Dialogue Servers. These endpoints must be described in the Administrative Table prior to invoking this function. The Dialogue Server establishes the required links and appends their descriptions to the Dialogue Table.

Also passed as a parameter when the function is called is a list of interrupt service routines, one corresponding to each destination endpoint. A software interrupt invokes the appropriate routine when a message is received at the endpoint.

Before establishing a link, the Dialogue Server verifies, by examining its Dialogue Table, whether the link already exists. Should it exist, a duplicate link is not established. Otherwise, an active socket is created and connected to the destination through its associated port, which is retrieved from the Administrative Table by referencing its logical name. For each link established, the Dialogue Table is updated to include the socket descriptor of the new active socket and the logical name of the destination endpoint. The server then

transmits the logical name of the source endpoint over the newly created link in order that the destination Dialogue Server may update its table.

If programs are written by different users, it should not be necessary to assume any kind of master/slave relationship requiring that one particular program be responsible for establishing the communication links to the other, as would be the case if NIL or the TCP functions were employed. This is a significant feature of the Session Layer as it allows programs to be developed independently, without reguard to a particular IPC initialization scheme. The Session Layer allows either or both communicating programs to be the master. If one program is the master the other will automatically assume the role of the slave, thereby ensuring compatibility. The exceptions to this are processes that are bound to reserved endpoints. They are always slave processes, meant to be utilities that lie dormant, listening for connection requests from processes that call *establish link*.

Special precautions must be taken to prevent the concurrent actions of two Dialogue Servers from resulting in the duplication of links. When establishing a link between two endpoints, the Dialogue Servers ensure that accidental duplication of the link will be undone. Consider the situation where two Dialogue Servers each attempt to initiate a link to the other at the same time. Neither would find the required connection described in their respective Dialogue Tables and both would proceed to establish the link. The Dialogue Server accepting the connection is required to verify that two similar entries to its table have not been made If duplicate entries are present, then one link must be removed. To avoid removing both links, a link is removed if the ASCII equivalent of the logical name of the accepting endpoint is less than that of the initiating endpoint.

4.4.4.2 Management Routines

- **Dispatch_msg**

This routine sends a message, passed from the calling process in a character string, over one or more links to the destination(s) specified in a list of logical names.

- **Read_msg**

This function call returns the next message received over the link from a specified source. When the Dialogue Server receives a message at a socket, it signals its user process with a software interrupt. A different interrupt is associated with each link. The user process can therefore identify the link over which a message has been received. When the user process receives the interrupt, its interrupt service routine is executed. The contents of the service routine is at the discretion of the user but it generally includes a call to read_msg. Alternatively, UNIX allows the user to disable interrupts, in which case any received signals are ignored.

Messages sent or received over any link are appended to its endpoint's associated send or receive FIFO buffer, which is approximately 2 kilobytes in length, until transmitted or read. Because of TCP's inherent reliability, messages are not lost if the endpoint's buffer becomes full; further transmissions to that socket are blocked until sufficient space becomes available.

4.4.4.3 Termination Routines

- **Delete_endpoint**

This routine initializes a procedure that deletes an entry from the Administration Table, terminates all communication links to the specified endpoint, and terminates the associated Dialogue Server process.

Prior to closing a connected socket, the Dialogue Server notifies each destination endpoint's Dialogue Server of the impending termination so that the destination endpoint may be closed and the link's entry may be removed from all the Dialogue Tables. The Dialogue Server instructs its Administrative Server to delete the endpoint entry from its table and to inform other Administrative Servers of the table update. The socket is then closed and the associated Dialogue Server process is terminated.

- **Terminate_link**

This function call terminates communication links and removes their descriptions from the Dialogue Table. The links are identified by the logical names of their destinations. The Dialogue Server notifies the destinations' Dialogue Servers of its intent to terminate the links so that they can update their Dialogue Tables. The sockets supporting the specified links are then closed. UNIX enforces a limit on the number of files that may be open to a user at any one time. The number of processes per user and the number of open files per process are set to 25 and 20 respectively by default in a standard UNIX 4.2BSD configuration. Endpoints look like open files to the operating system and in applications where the number of endpoints and open files approaches the user's limit, it is conceivable that infrequently used links would be opened and closed when they are needed.

- **Exit_session**

This routine ends the session by closing any open sockets in an orderly way and terminating the user process.

4.4.5 A Sample Session Layer Program

The following example illustrates an implementation of some of the Session Layer's functions. Two short programs are presented whose objective is to establish the simple but commonly used architecture depicted in Figure 4.6. and to cycle a message through each process.

The first program is composed of a single process, which is identified as process A. The second program is made up of a two processes: a parent process and a child process, identified as processes B and C, respectively.

Prior to executing these programs, the Session Layer must have first been invoked on one host. A boot-strap loading procedure then invokes the Session Layer

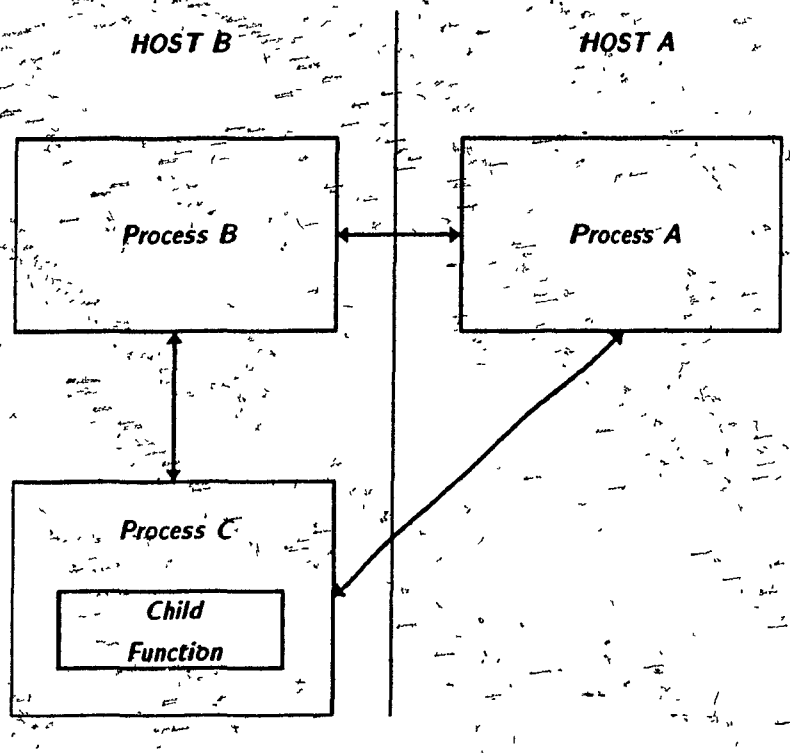


Figure 4.6 Architecture of the example's communication structure

program on the remote host. Initialization of the session is achieved through a remote execution call within the program; process A remotely invokes process B and establishes a communication link to it, through calls to init_session, create_endpoint, and establish_link. The child process, C, is created by forking process B and is linked to both its parent process and to process A, through the function establish_link. Using dispatch_msg and read_msg, a message is cycled from process A to B to C, and finally returned to A and displayed at the user's terminal. Comments have been included to assist in understanding the implementation. The logical names of the endpoints are "A", "B", and "C". The endpoints are referenced simply by their logical names.

Figure 4.7 Session Layer Application Program 1 for Host A – Process A.

```
#include <stdio.h>
#include "sltypes.h"      /* A Session Layer configuration file      */
```

```
extern int do_nothing(); /* A dummy interrupt handler provided        */
                         /* by the session layer                      */
int      done;           /* A global flag                             */
char msg[BUFFER_SIZE];   /* Array declaration for the message         */

get_msg_C (signo)        /* Interrupt service routine invoked when a  */
int        signo;        /* message arrives from Process C's endpoint.*/
{ read_msg('C', msg);    /* The function reads the message into msg   */
  done=1; }              /* and sets the global variable done         */

main()
{ int    get_msg_C();
  int_fn      fn[3];
  fn[0]=do_nothing;      /* Interrupt handlers for messages received  */
  fn[1]=get_msg_C;       /* at A from endpoints B and C respectively  */

  if (fork() == 0)       /* Remotely invoke process B on host B       */
  { execlp('rsh', 'rsh', 'host_B', 'process_B', (char *) 0); }
  init_session('A');     /* Pass the admin. server the endpoint list  */
  create_endpoint('A',2);/* Create endpoint A, max of 2 links allowed */
  establish_link
    ('B,C',fn);          /* Establish links to endpoints B and C      */
  dispatch_msg
    ('B', 'A');          /* Send the messages 'A' to the endpoint B.  */
  while (!done);         /* wait loop. Done is set to 1 by get_msg_C  */
  strcat (msg,' A');     /* Append the process name A to the message  */
  printf ('\n%s\n',msg); /* Display the final transmission: A B C A   */
  terminate_link('B,C'); /* Terminate links to A, this is optional    */
  exit_session(0);}      /* closes any open file descriptors and exit */
```

Figure 4.8 Session Layer Application Program 2 for Host B – Processes B, C

```
#include <stdio.h>
#include 'sltypes.h'      /* A Session Layer configuration file        */

extern int do_nothing(); /* A dummy interrupt handler provided        */
                         /* by the session layer                      */
int        done;         /* A global flag                             */
char msg[BUFFER_SIZE];   /* Array declaration for the message         */

get_msg_A (signo)        /* Interrupt service routine invoked when a  */
int        signo;        /* message arrives from Process A's endpoint */
{ read_msg ( 'A', msg ); /* The function reads the message into msg   */
  done=1; }              /* and sets the global variable done         */

get_msg_B (signo)        /* Interrupt service routine invoked when a  */
int        signo;        /* message arrives from Process B's endpoint.*/
{ read_msg ( 'B', msg ); /* The function reads the message into msg   */
  done=1; }              /* and sets the global variable done         */

main()
{
int     get_msg_B(), get_msg_C();
int_fn  fn[3];

init_session('B,C');     /* Initialize session with endpoints B and C */
                         /* Beginning of process C                    */
if (fork() == 0)         /* Create the child process C                */
```

```
{ create_endpoint
      ('C',2);          /* Create endpoint C, max of 2 links allowed  */
  fn[0]=do_nothing;     /* Interrupt handlers for messages received   */
  fn[1]=get_msg_B;      /* at C from endpoints A and B respectively   */
  establish_link
      ('A,B',fn);       /* establish links to A and B from C          */
  while (!done);        /* Wait loop until global done flag is set    */
  strcat (msg,' C');    /* Append ' C' to the message being cycled    */
  dispatch_msg
      ('A',msg);        /* Dispatch the message to process A          */
  exit_session(0);      /* closes any open file descriptors and exit  */
                        /* End of process C                           */
}

create_endpoint('B',2); /* create endpoint B, allow 2 links to it     */
fn[0]=get_msg_A;        /* Interrupt handlers for messages received   */
fn[1]=do_nothing;       /* at B from endpoints A and C respectively   */
establish_link
     ('A,C',fn);        /* establish links to A and C from endpoint B */
while (!done);          /* Wait loop until global done flag is set    */
strcat (msg,' B');      /* Append ' B' to the message being cycled    */
dispatch_msg('C',msg);  /* Dispatch the message to process C          */
exit_session(0);}       /* closes any open file descriptors and exit  */
```

4.4.6 Additional Features of the ISO Session Layer Model

With each refinement of the Basic Reference Model, the Session Layer has become more complicated. Session Layer concepts have evolved from a concept [Zimmermann 80] to a well defined protocol [Emmons and Chandler 83]. In part, this reflects the growing

maturity of protocol research as well as a better understanding of the services needed by the Application and Presentation Layers.

Having made our objectives clear, it is evident that we are not attempting to fully implement the OSI Session Layer Model. Instead, the design is tailored to satisfy the needs of our robotic environment. At this time, we would like to briefly mention some of the features of the ISO's Basic Reference Model that are **not** included in this implementation.

- *Quarantine* function that allows an entity to send and receive messages in groups, instead of individually. This is intended primarily for transaction-oriented activities such as those associated with database management. The current design allows the user to append several messages and transmit them all to the same destination. Future implementations could allow for grouping messages that are bound for different destinations.

- *Expedited Data Exchange* function that circumvents the flow control imposed on normal messages. This would typically be used for small, high priority messages such as 'abort'. This is certainly important, and will likely be added in a later revision. This service would require the support of an expedite feature at the Transport Layer.

- *Synchronization* function that allows a user entity to specify whether or not an acknowledgement should be provided by the server after each successful transmission. Because TCP guarantees message delivery, there is no need for this feature, except perhaps as a way to synchronize two end processes. In effect, the Dialogue Servers indirectly provide this synchronization by signalling the entity whenever a message arrives. The user can therefore implement synchronization through the proper use of the Session Layer function calls.

- *Resynchronization* function for transparent crash recovery, via 'rollback' to a 'checkpoint'. This is useful when sending large amounts of data from a single entity. The

dialogue server would periodically mark a position in the data stream and re-transmit from that mark. This would represent a much larger server overhead, as extra buffering would be required within the Session Layer. Since LAN Transport Layer service is very reliable, the overhead incurred in implementing this feature is judged to be unnecessay at the present time.

- *Dialogue Control* function to permit either full duplex or half duplex communication over the links. Since TCP supports full duplex communication, it is unnecessary to offer these alternatives.

- *Handshake on Termination* function which ensures that both communicating entities have no more messages to exchange and all data in transit is delivered prior to termination of the connection. This feature is more appropriate in 'long-haul' networks where messages move slowly. In our LAN environment, the propagation delay is relatively small. Handshaking is left for the user to implement.

- *Access Control* function which restricts communication access. The UNIX operating system inherently grants us a limited form of access control by virtue of the *owner*, *group*, and *world* permission attributes associated with files. We feel that this is sufficient at the present time. Future implementations may allow access to an endpoint based on user identification numbers.

4.5 Network Interface Library (NIL) Design

4.5.1 Overview

The interprocess communication network interface library (NIL) is a high level library of C functions which facilitates communication between programs executing on one or more hosts under the UNIX 4.2BSD operating system. The library functions simplify the implementation of IPC applications by providing an easy interface to the system network as

well as building blocks necessary to implement some of the more commonly used communication architectures. The basic services provided allow the creation of socket endpoints, the establishment of links between the sockets; the sending and receiving of messages over the links, and the termination of the links and sockets.

In terms of its degree of sophistication, NIL lies somewhere between the Session Layer and the low level TCP interface functions provided by UNIX. The functions contained in the NIL library are built upon the TCP functions. Programs implemented with either NIL or the TCP functions would be expected to execute at approximately the same speed since the NIL functions aggregate the TCP and other necessary UNIX functions into a form which is convenient for our applications.

In comparison to the Session Layer, NIL based communication is faster because it doesn't have its IPC mediated by servers. Links are established directly between user processes within the same or across different host boundaries. Messages therefore are transported over a single link instead of three as in the case of the Session Layer. The NIL functions parallel those of the Session Layer in their application; if a user understands how to use the Session Layer, he will be able to easily understand the NIL functions.

A disadvantage of using NIL over the Session Layer is its relative lack of power. The user has to do more work, writing longer programs to obtain a comparable implementation. The functions are also harder to use because some understanding of the TCP network functions is necessary.

The most important disadvantage is that the modular, standardized environment provided by the Session Layer is not present. An important concept of the Session Layer is to provide a high level, standardized environment that would facilitate integration of work-cell processes written by different users. Eventually a library of work-cell utility programs or modules would be accumulated that could be used by any member of the laboratory in their experiments. NIL does not provide any form of standardization. Because of this, the IPC architectures must be specified at the time that the communicating programs

are designed. NIL is best suited to simple applications where a small number (one or two) of fast communication links are involved and the architecture of all participating programs is well defined. Without standardization, each individual must maintain his own set of work-cell utilities, which is counterproductive to the laboratory as a whole.

A session can simultaneously incorporate communication structures from either the Session Layer, NIL, or TCP; one type of interface does not preclude the others. This can be used to great advantage by a knowledgeable programmer when the work-cell timing is critical.

4.5.2 The Network Interface Library Functions

As in the case of the Session Layer, the NIL functions can also be divided into three classes: initialization routines, management routines, and termination routines. The NIL functions and their associated parameters are presented in Figures 4.9, 4.13, and 4.14. A set of sample NIL programs, which implements the same example presented earlier that was implemented with the Session Layer, is presented in Section 4.5.3.

4.5.2.1 NIL Initialization Routines

The NIL initialization routines are summarized in Figure 4.9.

- **Session_init**
- **Session_server_init**

These routines work in conjunction with each other to establish an interprocess communication session between two processes. The processes must adhere to a master/slave relationship with the master calling session_init, which invokes the remote slave program. The slave, in turn, executes session_server_init.

- **Passive_socket**

Network Interface Library Initialization Functions	
Function Name	**Parameter(s)**
session_init	remote host name, remote program name, local passive socket descriptor, local TCP port number, connected socket, remote TCP port number
session_server_init	remote host name, local TCP port number, connected socket, remote TCP port number
passive_socket	local passive socket descriptor, local TCP port number
socket_connect	remote host name, remote TCP port number
socket_accept	local passive socket descriptor
create_child	child process identification number, connected socket descriptor, child function name
create_child_pair	child process 1 identifier, child process 2 identifier, child 1 connected socket descriptor, child 2 connected socket descriptor, child 1 function, child 2 function
create_child_remote	child process identifier, connected socket descriptor, child function, remote host name, remote TCP port number

Figure 4.9 Network Interface Library initialization functions

The function passive_socket establishes a communication endpoint which may accept connections from other endpoints, that is, a socket which has been bound to a TCP port. Following the call, the newly created passive socket listens for connection requests from active sockets. The TCP port number, which is assigned by the operating system, and the socket descriptor, are returned to the user. The socket descriptor must be referenced by the user when accepting connections to the endpoint, while the TCP port number must

be supplied to any process planning to establish a connection to the endpoint.

- **Socket_connect**

The function socket_connect creates a socket and connects it to the passive socket host which has been bound to the specified TCP port on the identified host. The function returns the descriptor of the connected socket, which must be referenced when sending or receiving messages.

- **Socket_accept**

This routine socket_accept is used to accept a connection at a specified socket. The accepting socket must be passive and listening for connection requests. When a connection is accepted, a copy of the original passive socket is created and connected; the original socket remains open and may continue to listen for and accept additional connections. The descriptor of the connected copy is returned to the user, to be referenced when sending or receiving messages over the link.

- **Create_child**

This routine forks[1] and establishes a communication link between the parent and child. The child process executes a specified function, passing as a parameter the socket descriptor of the link to the parent, in order that messages can be exchanged. Create_child returns the process identification of the child process as well as the socket descriptor, to be referenced by the parent when communicating with the child. The communication

---

[1] A UNIX fork splits a program into two concurrently executing copies. The original program is called the parent and the copy is called the child. Each process forks its execution path, with the parent proceeding in one direction and the child in the other, that is, the child executes one part of the original program and the parent executes the rest.

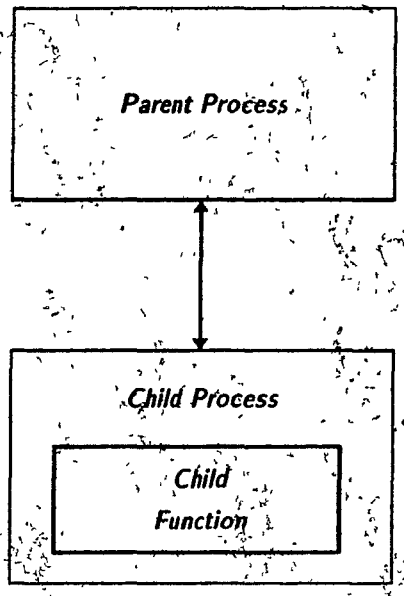


Figure 4.10 NIL communication architecture for create_child

architecture resulting from this call is shown in Figure 4.10. Communication links are represented by double arrow lines.

- **Create_pair**

This function forks a pair of child processes, with socket connections established between the parent and each child and between the two children. Each child process executes a specified function, passing as parameters the descriptors of its socket connections to the parent and to the other child. Create_pair returns the process identification of the child processes as well as the socket descriptors which must be referenced by the parent when communicating with each child. The communication architecture resulting from this call is shown in Figure 4.11.

- **Create_child_remote**

A child process is forked, with socket connections established between the parent and child, and between the child and a remote port. The child process executes a specified function, passing as parameters the descriptors of its two socket connections. Prior

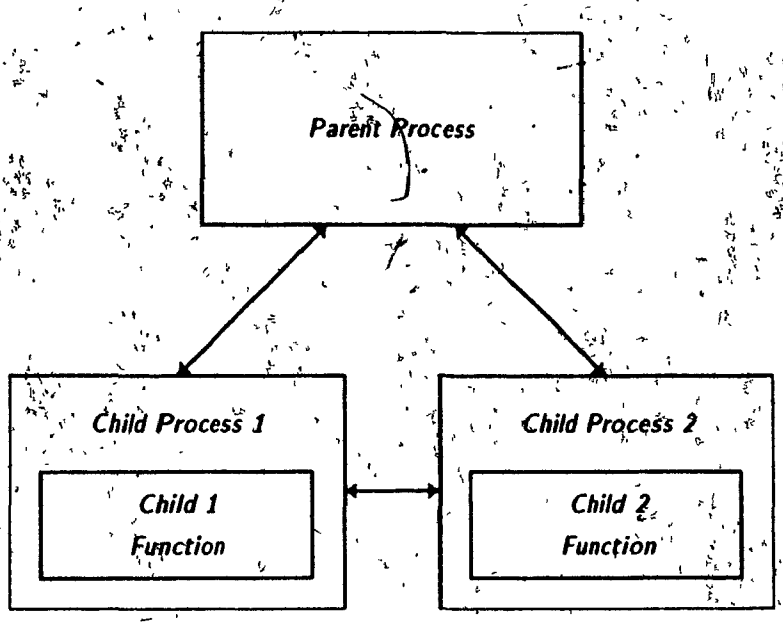


Figure 4.11 NIL communication architecture for create_pair

to calling this function, the parent process must have executed session_init; the remote process must have executed session_server_init and socket_accept. The communication architecture resulting from this call is shown in Figure 4.12.

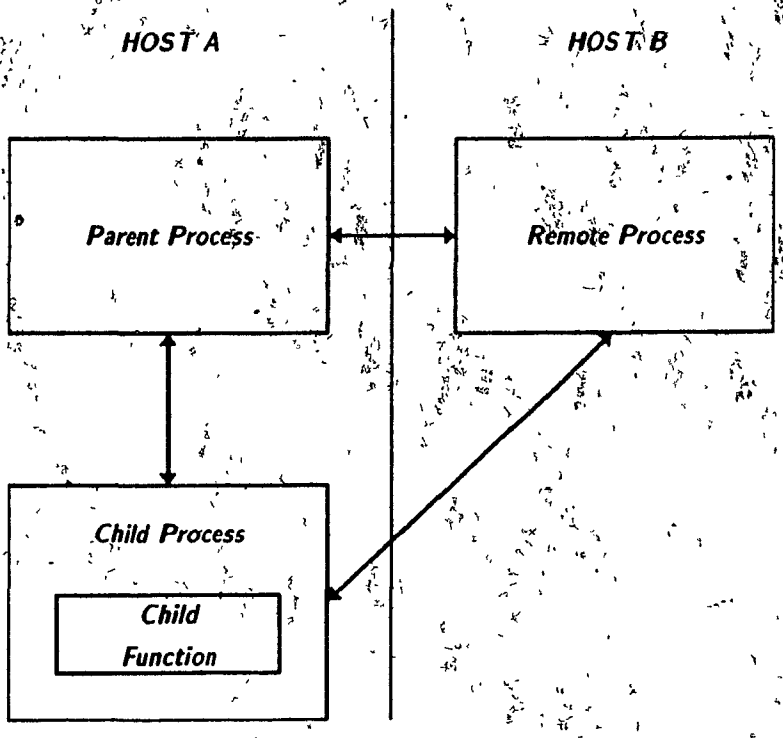


Figure 4.12 NIL communication architecture for create_child_remote

Network Interface Library Management Functions	
Function Name	Parameter(s)
check_msg	socket descriptor, time out period
read_msg	socket descriptor, message buffer, time out period
read_int	socket descriptor, data, time out period
read_char	socket descriptor, data, time out period
send_msg	socket descriptor, message buffer
send_int	socket descriptor, data
send_char	socket descriptor, data

Figure 4.13 Network Interface Library management functions

4.5.2.2 NIL Management Routines

The NIL management routines are summarized in Figure 4.13.

- **Check_msg**

This routine interrogates the specified socket for the presence of a message. In the absence of a message, it busy-waits either until the message arrival or for the duration of the specified timeout.

- **Read_msg**
- **Read_int**
- **Read_char**

These three routines return the next character string, integer or single character variable which has arrived at the specified socket. In the absence of a message, they busy-wait either until the message arrival or for the duration of the specified timeout.

- Send_msg
- Send_int
- Send_char

These three routines transmit a character string, integer, or single character variable over the specified links.

4.5.2.3 NIL Termination Routine

Network Interface Library Termination Function	
close	socket descriptor, file descriptor

Figure 4.14 Network Interface Library termination function

The NIL termination routine is shown in Figure 4.14.

- Close

This routine is a standard UNIX function which is used to close an open socket or file descriptor. It is considered a good programming practice to close any sockets or files, which were opened by the user, prior to exiting a program.

4.5.3 A Network Sample Interface Library Program

The following example illustrates an implementation of some of the NIL functions. Two programs, whose objective and communication architecture are identical to the Session Layer example presented in Section 4.4.5, have been implemented. The first program is again composed of a single process, identified as Process A; the second program is made up of a two processes: a parent process and its child, identified as processes B and C, respectively.

Initialization of the session is achieved through a master/slave relationship between the two programs: process A remotely invokes process B and initiates a communication link to it, through a call to session_init. Process B calls session_server_init to complete the establishment of the communication link. Process C is created by forking process B and is linked to both its parent process and process A, through the function create_one_remote. Using send_msg and recv_msg, a message is cycled from process A to B to C, and finally returned to A and displayed at the user's terminal.

As in the previous example, comments have been included to assist in understanding the implementation.

Figure 4.15 NIL Application Program 1 – Process A

```
#include 'NILconfig.h'          /* NIL host configuration header file   */
#include 'NILdefs.h'            /* NIL constants and definition         */

main()
{ char msg[BUFFER_SIZE];        /* data buffer for message transfer     */
  int       tcp_port_A,         /* the TCP port of A's endpoint         */
            passive_soc_A,      /* socket descriptor of the endpoint    */
            tcp_port_B,         /* TCP port of process B's  endpoint    */
            soc_B,              /* descriptor of link to process B      */
            soc_C;              /* descriptor of link to process C      */

  session_server_init           /* Invoke the remote process 'process_B'*/
       ( 'host_B',              /* on host 'host_B', and initiate a     */
         'process_B',           /* communication link                   */
         &passive_soc_A,        /* socket desc of passive socket        */
         &tcp_port_A,           /* TCP port number of passive socket    */
         &soc_B,                /* socket desc of local connected endpt */
```

```
                &tcp_port_B );      /* TCP port of endpoint B              */

    soc_C = socket_accept           /* Accept a connection request from    */
      ( passive_soc_A );            /* process C                           */

    strcpy (msg,'A');               /* Send a message containing the process*/
    send_msg (soc_B,msg);           /* name, A, to process B               */

    while (recv_msg                 /* Read a message from the link soc_C  */
      ( soc_C,msg,10) == 0);        /* into the buffer msg with a timeout  */
                                    /* of 10 sec                           */
    strcat (msg,' A');              /* Append process name A to the message*/
    printf ('\n%s\n', msg);         /* Display the final message: A B C A  */
    close(passive_soc_A);           /* close open socket descriptors       */
    close(soc_B);
    close(soc_C);
    exit(0);                        /* and exit                            */
}
```

Figure 4.16 NIL Application Program 2 – Processes B and C

```
#include 'NILconfig.h'          /* NIL configuration header file      */
#include 'NILdefs.h'            /* NIL constants and definitions      */

main(argc, argv)
int argc;
char **argv;
{ char msg[BUFFER_SIZE];        /* data buffer for message transfer   */
```

```
int       tcp_port_B,          /* the TCP port of B's endpoint      */
      passive_soc_B,           /* socket descriptor of the endpoint */
      tcp_port_A,              /* TCP port of endpoint of process A */
      soc_A,                   /* descriptor of link to process A   */
      fnc_child_C(),           /* function executed by child process*/
      soc_child_C,             /* descriptor of link to child       */
      child_C_id;              /* process identification of child   */

session_server_init            /* establish the communication session */
      ( argv[1],               /* with the master process A. The host */
        argv[2],               /* name and TCP port number of the     */
        &passive_soc_B,        /* endpoint owned by process A are     */
        &tcp_port_B,           /* specified by passed parameters      */
        &soc_A,                /* argv[1] and argv[2] respectively.   */
        &tcp_port_A );

create_one_remote              /* fork a child process, process C, to */
      ( &child_C_id,           /* be linked to both the parent and the */
        &soc_child_C,          /* remote process A. The child process */
        fnc_child_C,           /* will execute fnc_child_C.           */
        argv[1],
        argv[2] );

while (recv_msg                /* Read a message from the link soc_A  */
 ( soc_A,msg,10) == 0);        /* into the buffer msg with a 10 second */
                               /* timeout period.                     */
strcat (msg," B");             /* Append the process name B to msg    */

send_msg (soc_child_C,msg);    /* Send the modified message to the    */
```

```
                                /* child process C                    */
  close(passive_soc_B);         /* close open socket descriptors      */
  close(soc_A); close(soc_child_C);
  exit(0);                      /* and exit                           */
}

/*   This routine is executed by process C, the child of process B    */
/*   after the fork is called. Descriptors of socket links to the     */
/*   parent, process B, and to a remote process, A, are passed in     */
/*   soc_parent_B and soc_remote_A, respectively.                     */

fnc_child_C (soc_parent_B, soc_remote_A)
int          soc_parent_B, soc_remote_A;
{ char msg[BUFFER_SIZE];        /* buffer for message transfer        */

  while (recv_msg               /* Read a message from the link       */
  (soc_parent_B,msg,10)==0);    /* soc_parent_B into buffer msg       */
  strcat(msgmsg,' C');          /* Append the process name 'C'        */
                                /* to the message                     */
  send_msg(soc_remote_A,msg);   /* Send the modified message to       */
                                /* remote process A                   */
  close(soc_parent_B);          /* Close open sockets                 */
  close(soc_remote_A);
  exit(0);                      /* and exit                           */
}
```

4.6 Comparision of the Session Layer and NIL

The Session Layer and NIL programs presented in Sections 4.4.5 and 4.5.3

demonstrate that interprocess communication can be achieved in a simple and concise way with either approach. An important feature of these interfaces is their similarity with respect to their application. The user does not have to contend with different interfaces, which makes using them easier. The sample programs are relatively short in comparison to a TCP implementation of the same structure; a TCP implementation would be in excess of one thousand lines of program code and would be composed of a majority of the NIL source code. The relatively short length of the NIL and Session Layer programs clearly demonstrates the power that these interfaces offer to the user. Additional comparisons are presented in a performance analysis in Chapter 5.

4.7 Error Detection and Recovery

The network interface provides three levels of error detection and recovery. The processing is bottom up with the lowest level managed by TCP, the intermediate by the Session Layer and NIL functions, and the top level by the user program.

The TCP stream protocol provides a reliable endpoint-to-endpoint means of communication in a network environment. Transmission errors do not affect the correct delivery of data. TCP achieves this by recovering data that is lost, damaged, duplicated or delivered out of order. In the event that TCP cannot recover from an error condition, an appropriate message is passed to the Session Layer or NIL function.

Both libraries adhere to the UNIX standard for error reporting. In the event of an error, a function will return -1 and the user can opt to display the library function name in which the error occurred, a system error message, and a function error message. The function error message identifies the low level UNIX routine in which the error occurred while the system error message specifies the exact nature of the problem within the UNIX routine. For example, if an error occurred within the NIL function *socket_connect*, the function error message could be *"Cannot establish connection"*, implying that the error occurred in the UNIX routine *connect*; while the associated system error message might be

"Connection timed out", indicating that the remote endpoint did not accept the connection with the specified timeout period. It is the user's responsibility to decide and implement the next course of action. In the event of a catastrophic error, such as the loss of a host, both the Session Layer and NIL functions will automatically issue both system and function error messages and terminate the programs.

The Session Layer is more capable than the NIL library in dealing with error situations on its own. The Dialogue Server processes and attempts to recover from errors signalled by TCP. The Server passes TCP error messages such as communication timeouts and blocked transmissions to the user process, which is responsible for coping with these errors. The Servers also trap and process errors introduced by the user program such as invalid specification of links and endpoints. For more severe error situations, such as corrupted links or the crash of a host, full recovery is not possible; the session can not be restored to an image of what it was before the error situation occurred. In these cases, the Server performs a partial recovery which involves an orderly shutdown of the affected links and passes an appropriate error message to the user process. Under such severe conditions, it may not be possible to continue with the session. The partial recovery, ensures that processes on the other hosts participating in the session are not inundated with error messages or blocked in some way.

To illustrate some of the Session Layer's error detection and recovery procedures, consider the following examples.

Assume that a user process which is participating in a session unexpectedly aborts, resulting in the need to terminate the session. The Dialogue Server dedicated to that process would detect the severed communication link and would terminate the associated Session Layer endpoint. This involves terminating all of the links between the local and remote Dialogue Servers, terminating the link between the local Administrative and Dialogue Server, and finally shutting down the local Dialogue Server. The remote processes participating in the session, will detect that their counterpart is no longer active and they may in turn also terminate.

As a second example, assume that a host crashes. If a Dialogue Server attempts to establish a link to a endpoint on the affected host, or attempts to send or receive over a disconnected link, an error is returned from TCP. The Dialogue Server informs the Administrative Server that a link or endpoint is disabled. The Administrative Server verifies whether the remote host is operational. If not, an appropriate message is broadcast to the other hosts, which in turn, removes entries of endpoints which resided on the affected host from their Administrative Tables. Dialogue Servers are notified to remove from their tables all links to endpoints on the affected host.

4.8 Summary

In this chapter, we have presented the design strategy, architecture, and programming interface of a system for interprocess communication within our distributed robotics environment. The environment is based on host computer systems operating under the UNIX 4.2BSD operating system communicating over an Ethernet local area network.

The communication environment is designed to be flexible, to meet the needs of a constantly changing research environment; modular, to support robot work-cells that are composed of distinct elements; and easy to use, so that programmers do not have to devote time towards understanding the detailed mechanisms of interprocess communication.

The network interface is designed to provide a standardized way of achieving IPC while relieving the user from the low level aspects of network communication. Three interfaces are available, each providing a different level of sophistication. Each level provides mechanisms for establishing, managing, and terminating communication sessions, but in a different way. The higher the level, the easier it is to implement and manage interprocess communication in an efficient manner. The lower levels offer increased speed and flexibility

The highest level interface is based on a partial implementation of the ISO's Session Layer service specifications. The implementation is based on a hierarchical framework composed of an Administrative Server which is responsible for maintaining communication

endpoints and a Dialogue server which is responsible for managing the communication links. Of the three interfaces, this is the only level that offers a standardized way of achieving IPC.

The intermediate level interface is provided by the Network Interface Library. This interface is less sophisticated than the Session Layer in that it is more difficult to use and its functions are less powerful.

The lowest level interface is provided by UNIX 4.2BSD in the form of its TCP function library. This level offers the most flexibility and speed but these attributes are offset by the difficulty encountered in using the TCP functions. This level is primarily of interest to system level programmers.

Two sample programs illustrating both the Session Layer and Network Interface Library functions were presented to demonstrate the ease with which interprocess communication can be effected.

Error handling capabilities of both the Session Layer and NIL interfaces was presented. The Session Layer is far superior in its ability to deal with error situations, while the NIL functions require, to a greater extent, that the user decide what the course of action will be for normal and abnormal conditions and how it will be implemented when an error occurs.

Chapter 5 Performance Analysis

5.1 Introduction

Having presented the architecture and design details of the communication environment, we present a performance analysis of the Session Layer and Network Interface Library implementations.

5.2 Performance in a Distributed System

There are many factors which influence the performance of a distributed program. We will examine some of the factors particular to our environment to determine how they affect communication.

5.2.1 Network Tuning

Network tuning is the process of adjusting Transport Layer buffer sizes and flow control parameters. It can be used to optimize network throughput based on the requirements of the applications that the network supports. In our environment, network traffic consists primarily of file transfers and remote terminal applications. This is in contrast to the robot work-cell message traffic which is relatively low in volume, consisting of messages which are generally less than 32 characters in length. The network is presently tuned for

larger data transfers; this is therefore at the expense of the work-cell's communications needs. Although tuning affects the overall throughput rate, the same conditions are applied equally to both Session Layer and NIL programs since they are both based on TCP.

It is beyond the scope of this thesis to address the network performance below the level of the Session Layer and Network Interface Library. A performance evaluation of TCP, which includes the Transport, Network, Data Link, and Physical Layers, in a network configuration similar to our own, composed of VAX and SUN hosts operating under UNIX 4.2BSD and connected by an Ethernet is presented by Chanson [Chanson et al. 84]. In very general terms, the performance of the Transport Layer implementation depends on the following factors.

- The control structure which, in part, deals with packet format, retransmission and timeout protocols, data fragmentation, and packet routing.
- The flow control mechanisms which are based on the available buffer space at the source and destination endpoints.
- The implementation of the protocol. In the case of 4.2BSD, the protocol is implemented in the UNIX kernel which is the lowest level of the operating system and affords the greatest speed.
- The operating system overhead. This refers to the time consumed in process swapping, paging, system calls, and any other aspects pertaining to marshalling the environment of the processes.

5.2.2 Network Contention

Ethernet employs a carrier sense protocol for accessing the network. If a host has data to transport, it first interrogates the network for the presence of a carrier, that

is, a packet transmission present on the network. If the channel is busy, it waits a random amount of time and then retries; otherwise, the data is transmitted. Network contention arises through the concurrent action of more than one host attempting to transmit a packet on the network. Ethernet does not employ a priority or queuing mechanism to resolve contention. If a collision occurs, contention is resolved by having the hosts wait a random amount of time before retrying.

The performance of network protocols in the presence of contention is always decreased. Network contention can be partly evaluated by examining the collision rate of packets on the network. This rate is related to the number of packet retransmissions and thus affects the performance. To some extent, network balancing can minimize contention by optimizing packet sizes and timeouts.

In our environment, the average collision rate is very low, approximately 100 parts per million, and is not considered to contribute any significant effects.

5.2.3 Local versus Remote Communication

If interprocess message passing is confined to a single host, the physical network is never accessed and contention can not arise. Because remote message passing requires network access, it is somewhat slower. Since in our environment network contention is very low, the effects of local versus remote communication are negligible.

5.2.4 Processor Load

Processor load, or CPU load, plays a major role in network performance. TCP was designed to provide robust communication over lossy, long haul networks and therefore has extensive packetization and error detection and correction features. This requires a tremendous amount of CPU resources; so much so that some argue that TCP is too powerful for LAN applications. An alternative, less robust protocol has been proposed by Chanson [Chanson et al. 85].

5.2.5 Message Length

The length of messages is certainly a contributing factor to network and system performance. Messages designated to be transported are first stored in a TCP transmit buffer which is normally 2048 bytes in length, then packetized, usually in fragments of 128 bytes for TCP's stream protocol, and finally put on the network. The number of packets and ultimately the CPU load are proportional to the message length.

5.2.6 Program Overhead

Program overhead which entails calling routines to receive, parse, and send messages contributes to the end to end bandwidth. Efficient architectures and programming, both in implementing the network interface functions and on the part of the user who integrates these functions into their applications, is important.

5.3 Performance of the Implementation

The Session Layer performance was evaluated by comparing the times required to cycle a message, using both the Session Layer and NIL services, to and from a remote process.

The routines required to effect data transfers using the low level TCP functions amount to the *send_msg* and *recv_msg* functions of NIL. Therefore, from the perspective of this performance evaluation, the transmission times of both NIL and the TCP implementation can be considered to be identical. Henceforth the figures presented for the NIL implementation apply equally well to both NIL and TCP implementations

The time required to effect a message transfer depends on many factors, including the length of the message, the system loads, and the network balancing parameters which affect the performance of the Transport Layer.

Our objective was to evaluate the Session Layer *relative* to the Network Interface Library under the conditions of our present TCP installation. The actual times required to transport messages over the network are only relevant to our network configuration and tuning; relative comparisons are therefore made. NIL was chosen as the basis for performance comparison because it is the fastest TCP based mechanism available for transporting messages. We were not concerned with measuring the effects of factors specific to the TCP protocol. Thus, without loss of generality, we assumed that the time required to effect message transfers could be represented as an overhead time, due to TCP and the network interface functions, plus an increment which can be characterized as a function of message length. Further, we consider that both these parameters could themselves be dependent on the system load level and the type of host computer: VAX-11/750 (VAX), MicroVAX II (mVAX), or SUN System 2 (SUN). In evaluating the session layer, it was assumed that data generated under almost identical conditions would not have been subjected to significant variations in overhead time; any error introduced by this simplification could be absorbed into the overall error of the model without serious consequence. In other words, the small fluctuations in system load encountered while performing the experiments were not considered to be critical and therefore were not accounted for by the model.

In evaluating the Session Layer, we wanted the results to reflect typical conditions encountered in the laboratory. Since the packet collision rate is essentially zero, averaging 100 parts per million, the effect of varying the *network load* was not deemed necessary. Thus measurements of load will be based strictly on *CPU load*, which for the purposes of this study, is defined as the number of jobs in the operating systems run queue averaged over the last one minute period.

5.4 Experimental Design

Programs which measure the time to cycle messages between two hosts using both the Session Layer and the NIL services were developed. A block diagram of the experimental setup is presented in Figure 5.1.

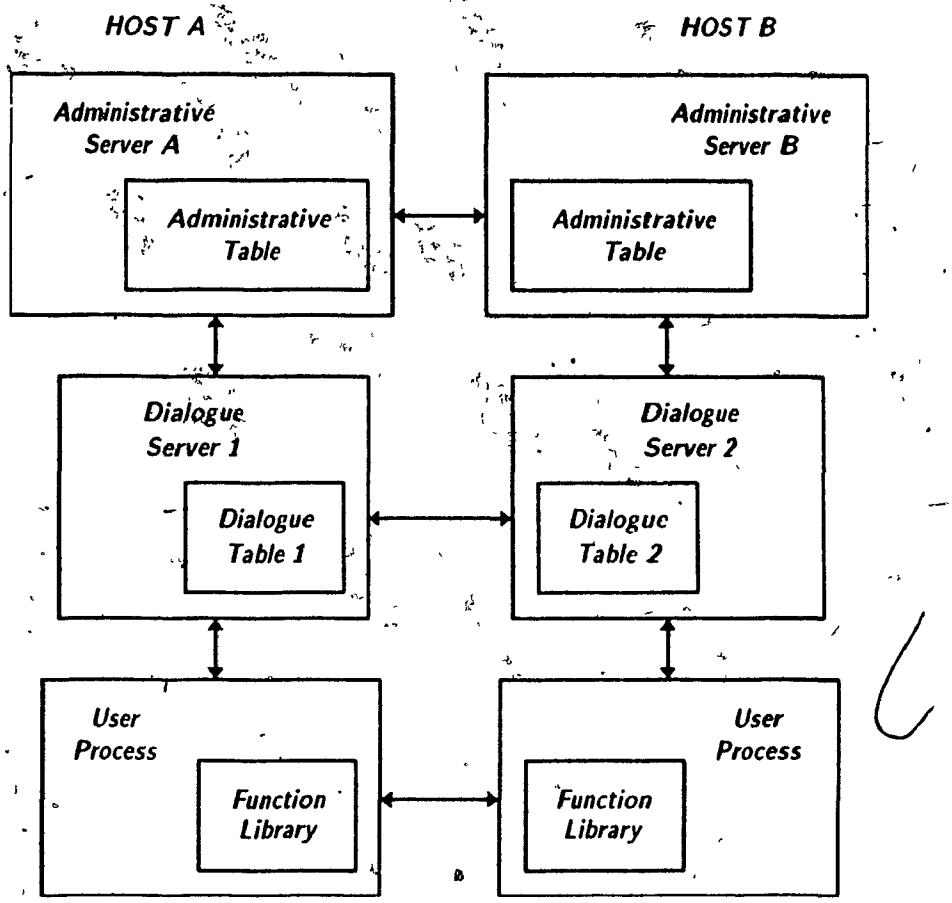


Figure 5.1 Configuration implementing both Session Layer and NIL links

Using the Session Layer facility, a message was transmitted from the local endpoint to the remote endpoint, and then returned. The elapsed time from the initial *dispatch_msg* function call to the return of the final *read_msg* call was measured. The message circulation was then immediately duplicated, this time over the direct socket connection, using the NIL *send_msg* and *recv_msg* functions. The time of the message transfer was again determined.

This procedure was performed in triplicate for messages of lengths ranging from 1 to 2048 characters, with an emphasis on lengths at the lower extreme of the range. An upper limit of 2048 characters was chosen because it corresponds to the transmit and receive message buffer sizes allocated to TCP when the system was configured.

To examine the influence of system load and computer host types, two sep-

arate studies were performed. The first study considered the affect of system load on Session Layer performance. The programs were executed on the same host configuration, consisting of two VAX 11/750 computers, at three different CPU load levels, classified as 'low', 'medium', and 'high'. For the purpose of this study, low, medium, and high loads corresponded to an average of 1, 5, and 9 jobs respectively, in the run queue over the last one minute period.

The second study evaluated the performance of the Session Layer over various combinations of different host computer types. The programs were executed on six different host-pair configurations: VAX/VAX, VAX/mVAX, VAX/SUN, mVAX/mVAX, mVAX/SUN, SUN/SUN. All programs were executed under conditions of medium system load in an attempt to parallel the operating conditions that are normally encountered in the laboratory when running robot experiments.

The raw data generated by these studies is not presented in this thesis as it is too voluminous. The data tabulated in the subsequent sections of this chapter represents results of our analyses.

5.5 Analysis and Results

The objective of the data analysis was to compare the times required to transmit messages, under identical conditions, using the Session Layer and NIL services. From this comparison, a relative measure of the Session Layer's performance was derived. Two separate analyses were performed, based on the studies discussed above. The first considers variations in system load; while the second examines the effects of different types of host computer.

All data was analysed using the SAS General Linear Models Procedure Package [SAS 82]. The statistical concepts for the analysis were based on material from Searle [Searle 71] and Draper [Draper and Smith 66].

5.5.1 Variation of System Load

It was assumed that the transmission time could be described by a relation involving the message length, the system load level and the communication implementation – Session Layer or NIL. Other factors which could influence transmission time were assumed to remain constant. As will be more apparent later, the consequence of an error in this assumption is that our estimation of the dependence of transmission time on system load may be in error. However, since our objective is to obtain a *relative* comparison of the Session Layer to NIL, this consequence should not be critical to our analysis. Further, it was assumed that the covariance model which is described by equation 5.1 could be applied.

$$Y_{ij} = \alpha_i + \tau_{ij} + (\beta_i + \rho_{ij}) \cdot X \tag{5.1}$$

where

i denotes the consistent system load on all machines: low, medium, or high, i = 0, 1, 2;

j denotes the communication implementation: Session Layer or NIL, j = 0, 1;

Y measures the time to circulate a message under load condition i using communication implementation j;

X represents the message length;

α_i represents the component of the time, which is independent of the message length, required to circulate a message using NIL, under load condition i;

β_i measures the dependence on message length of the time to circulate a message using NIL under load condition i;

τ_{i0} represents the overhead time of the Session Layer under load condition i, that is, the additional time, independent of message length, which is required to circulate a message under load condition i using the Session Layer rather than NIL;

$\tau_{i1} = 0$;

ρ_{i0} represents the difference in the dependence on message length of the time to circulate messages using the Session Layer rather than NIL, under load condition i;

$\rho_{i1} = 0$.

Four variations of equation 5.1 were considered:

$$T_{ij} = \alpha_i + \tau_{ij} + (\beta_i + \rho_{ij}) \cdot L \tag{5.2}$$

$$T_{ij} = \alpha_i + \tau_{ij} + (\beta_i + \rho_{ij}) \cdot ln(L) \tag{5.3}$$

$$ln(T_{ij}) = \alpha_i + \tau_{ij} + (\beta_i + \rho_{ij}) \cdot L \tag{5.4}$$

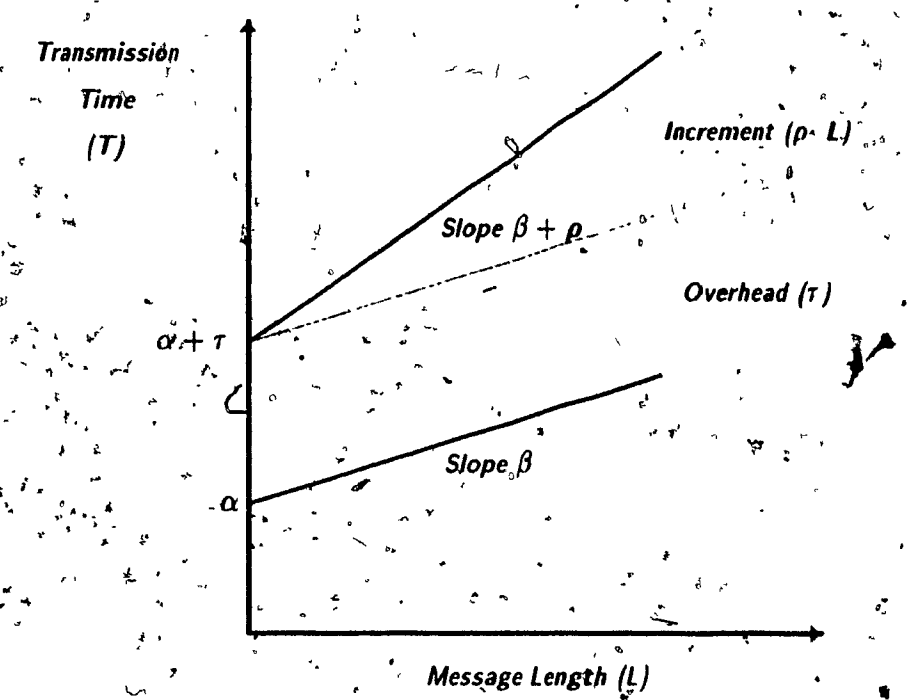
$$ln(T_{ij}) = \alpha_i + \tau_{ij} + (\beta_i + \rho_{ij}) \cdot ln(L) \tag{5.5}$$

where

T denotes the time to circulate a message (seconds);

L is the message length (characters);

$ln(\cdot)$ represents the natural logarithm.



NIL: $T = \alpha + \beta \cdot L$

Session Layer: $T = \alpha + \tau + (\beta + \rho) \cdot L$

Figure 5.2 Estimation of Session Layer and NIL message transmission times

Each model was fitted to the data using the method of least squares. The coefficient of correlation, which measures the amount of variation of the data which is described by the model, was computed for each case. Based on the coefficient of correlation, the most appropriate model was determined to be the linear relation, 5.2. A graphical representation of this equation is presented in Figure 5.2.

Simplification of the linear model was considered; the reduction of the model to one which would assume that effects of the Session Layer were independent of system load, that is, $\tau_{ij} = \tau_j$ and $\rho_{ij} = \rho_j$ for all i, was examined. F statistics were computed to evaluate the significance of the variations of the parameters τ and ρ between system loads. Both F statistics were determined to be highly significant, implying that the data demonstrated a difference according system load of the Session Layer's effect on both the overhead time and the dependence of time on message length. Therefore, the model was not reduced.

The estimated parameters of equation 5.2 are tabulated in Figure 5.3 The additional time to circulate messages using the Session Layer rather than NIL was described by an overhead time which is independent of the message length, and a time which is directly proportional to the message length. The overhead, which is represented by τ, was determined to increase with increasing system load. At low system load, the overhead was estimated to be 1.62 seconds, while at high load, the overhead was estimated to be 1.93 seconds. The proportionality constant characterizing the effect of message length on the Session Layer's performance, represented by ρ, was also determined to increase with system load. At low load, the proportionality constant was estimated to be 0.0022 sec/char, indicating that the additional time required by the Session Layer to transmit messages increases by 0.28 seconds for each increase of message length by 128 characters. At high system load, the additional transmission time required by the Session Layer to transmit a message was estimated to increase by 1.05 seconds for each 128 character increase in message length.

CPU Load	α sec	τ sec	β sec/char	ρ sec/char
low	0.01	1.62	0.0021	0.0022
medium	0.02	1.75	0.0049	0.0052
high	0.04	1.93	0.0067	0.0082

Figure 5.3 Parameter estimates of equation 5.2

In Figure 5.4, estimates of message transmission times based on the derived regression equation are presented for each system load level, for messages of lengths 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and 2048 characters.

5.5.2 Variation Between Host Computers

The analysis of data generated by the second study, which considered variations in host computer types, maintained at medium system load, parallelled the analysis

Load	Low		Medium		High	
Mode	NIL	SL	NIL	SL	NIL	SL
2	0.02	1.65	0.03	1.79	0.05	2.00
4	0.03	1.66	0.04	1.81	0.07	2.03
8	0.04	1.67	0.06	1.85	0.09	2.09
16	0.05	1.71	0.10	1.93	0.15	2.21
32	0.09	1.78	0.17	2.09	0.26	2.45
64	0.15	1.92	0.33	2.42	0.47	2.92
128	0.29	2.19	0.64	3.06	0.90	3.88
256	0.56	2.75	1.26	4.36	1.76	5.79
512	1.10	3.86	2.51	6.94	3.49	9.60
1024	2.19	6.08	4.99	12.11	6.94	17.23
2048	4.36	10.53	9.97	22.46	13.84	32.49

Figure 5.4 Estimates of message transmission times (seconds).

described above. It was assumed that the transmission time could be described by a relation involving the message length, the computer types and the communication implementation – Session Layer or NIL. Other factors which could influence transmission time were assumed to remain constant. Since our objective was to obtain a relative comparison of the Session Layer to NIL, this assumption was not judged to be critical to our analysis. It was also assumed that the covariance model defined by equation 5.1, with $i=0,1,2,\ldots,5$ denoting the computer type, could be applied.

The four variations of the model described by equation 5.1 were again considered. Each model was fitted to the data using the method of least squares and the coefficient of correlation computed. The most appropriate model was selected, based on the coefficient of correlation, to again be the linear relation represented by equation 5.2.

Simplification of the linear model was considered; the reduction of the model to one which would assume that effects of the Session Layer were independent of computer

type was examined. F statistics were computed to evaluate the significance of the variations of the parameters τ and ρ between various pairs of hosts. Both F statistics were determined to be highly significant, implying that the data demonstrated a difference according to host computer type of the effect of the Session Layer on both τ, the overhead component of the message circulation time, and ρ, the time component which varies with message length. The model was therefore not reduced.

Host	α	τ	β	ρ
	sec	sec	sec/char	sec/char
VAX/mVAX	0.02	1.52	0.0017	0.0046
VAX/VAX	0.01	1.47	0.0024	0.0043
SUN/VAX	0.02	1.56	0.0018	0.0024
mVAX/SUN	0.01	1.42	0.0010	0.0016
SUN/SUN	0.01	1.39	0.0011	0.0020
mVAX/mVAX	0.01	1.44	0.0009	0.0014

Figure 5.5 Parameter estimates for transmission times

The estimated parameters of equation 5.2 are tabulated in Figure 5.5. The overhead, which is described by τ, was determined to range from 1.39 seconds for the SUN/SUN configuration to 1.56 seconds for the SUN/VAX configuration, a variation of 0.17 seconds. The proportionality constant characterizing the effect of message length on the Session Layer performance, described by ρ, was also determined to vary according to host configuration. The proportionality constant was estimated to range from 0.0014 sec/char for the mVAX/mVAX to 0.0046 sec/char for the VAX/mVAX configuration. In other words, the difference between transmission times of the Session Layer and NIL increased from the overhead time by an amount ranging from 0.18 to 0.59 seconds for each additional 128 characters.

In Figures 5.6 and 5.7, estimates of message transmission times for NIL and Session Layer implementations are presented for each host computer type, for messages of lengths 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 characters.

NIL Transmission Time (sec)						
Char	VAX/VAX	mVAX/mVAX	SUN/SUN	VAX/mVAX	mVAX/SUN	SUN/VAX
2	0.01	0.01	0.01	0.02	0.01	0.03
4	0.01	0.01	0.01	0.02	0.01	0.03
8	0.01	0.02	0.01	0.03	0.02	0.04
16	0.03	0.02	0.02	0.04	0.02	0.05
32	0.07	0.04	0.04	0.07	0.04	0.08
64	0.15	0.07	0.08	0.13	0.07	0.14
128	0.30	0.12	0.15	0.24	0.13	0.25
256	0.60	0.24	0.30	0.46	0.26	0.48
512	1.21	0.46	0.59	0.90	0.50	0.93
1024	2.42	0.91	1.18	1.78	1.00	1.84
2048	4.85	1.81	2.36	3.54	2.00	3.67

Figure 5.6 Estimates of NIL message transmission times

Session Layer Transmission Time (sec)						
Char	VAX/VAX	mVAX/mVAX	SUN/SUN	VAX/mVAX	mVAX/SUN	SUN/VAX
2	1.48	1.46	1.40	1.55	1.43	1.59
4	1.49	1.46	1.40	1.56	1.43	1.60
8	1.52	1.47	1.42	1.58	1.44	1.61
16	1.57	1.49	1.44	1.64	1.46	1.65
32	1.68	1.53	1.49	1.74	1.50	1.71
64	1.89	1.60	1.59	1.94	1.59	1.85
128	2.32	1.74	1.79	2.34	1.75	2.12
256	3.17	2.03	2.19	3.15	2.07	2.66
512	4.88	2.61	2.98	4.77	2.72	3.73
1024	8.29	3.77	4.58	8.00	4.01	5.89
2048	15.11	6.08	7.76	14.46	6.61	10.20

Figure 5.7 Estimates of Session Layer message transmission times

5.6 Discussion

The data analysis determined that the performance of the Session Layer depends

on the system load, the host computer type and the message length. Further, the analysis suggested that the relation between the message circulation time and the message length was linear.

The difference in performance between the Session Layer and direct TCP, as measured by NIL, was expressed as the sum of an overhead time which is independent of the message length, and an increment which is directly proportional to the message length, as was defined by equation 5.2. The overhead of the Session Layer was estimated to range from 1.39 to 1.93 seconds, depending on the system load and host configuration. The increment was observed to vary from as low as 0.18 to as high as 1.05 seconds per 128 character block transmitted in a message. That is, in addition to the overhead which varied between 1.39 and 1.93 seconds, each block of 128 characters transmitted will add from 0.18 to 1.05 seconds to the transmission time.

The variations in transmission times between host configurations did not imply that any particular host type will consistently enhance or impede the Session Layer's performance. The estimates of Session Layer overhead and increment times for each host computer combination indicated that the Session Layer transmission time is highest when a VAX host is involved. However, it was observed that the VAX configurations also exhibited the highest NIL transmission times. Thus, although data transmission is apparently slowest when a VAX host is involved, the relative performance of Session Layer and NIL based application programs is not diminished.

The times required to circulate messages between two processes using the Session Layer and NIL services are tabulated in Figure 5.4 as functions of message length for low, medium and high system loads, for the VAX/VAX host configuration. In Figures 5.6 and 5.7, similar results are presented for host configurations of VAX/VAX, mVAX/mVAX, SUN/SUN, VAX/mVAX, mVAX/SUN, SUN/VAX, based on medium system loads. Three observations are immediate from Figures 5.4, 5.6, and 5.7: the Session Layer results are consistently higher than the NIL times, the transmission times increase as the message length increases, and the transmission times increase as the system load increases.

In our applications, the majority of messages passed are expected to be short, not exceeding 32 characters. In order to better understand the performance characteristics of the Session Layer, a bound on the difference in performance of the Session Layer and NIL for messages of this length was estimated. The difference in message transmission times between the Session Layer and NIL for 32 character messages were determined from Figures 5.4, 5.6, and 5.7. For a VAX/VAX host configuration, Figures 5.6 and 5.7 estimated the additional time required by the Session Layer to be 1.61 seconds, while Figure 5.4 estimated the time to be as high as 2.19 seconds in the presence of a high load, a variance of 36%. This variance applies directly to the pair of machines, VAX/VAX, on which the measurements were made. Although the effect of load on the Session Layer's performance cannot be assumed to be constant over all machines, it would be reasonable to expect that comparable loads would produce similar effects. Applying the 36% variance to the estimated difference in Session Layer and NIL transmission times for other host configurations, it was predicted that the additional time expended by the Session Layer should not exceed 2.3 seconds at high system load for any host combination. Thus, the additional time required for most messages transmitted via the Session Layer is not expected to exceed 2.3 seconds, regardless of system load and host computer configuration.

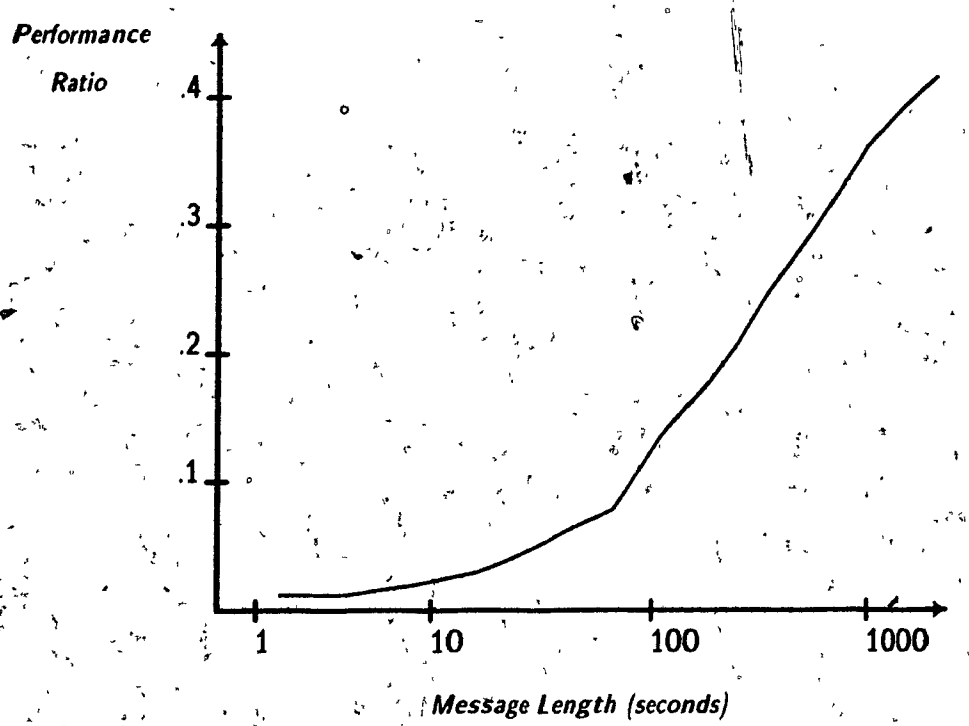
It is interesting to notice from Figures 5.4, 5.6, and 5.7 that the effects of message length would be more evident in NIL than in Session Layer applications. As the message length increased from 2 characters to 2048 characters, the time to circulate a message using the Session Layer did not increase by more than 17 fold for any level of system load or any host computer combination; while the time to circulate a message using NIL functions increased by more than 100 fold for all load levels and host configurations. In Figure 5.8, the ratio of the times to transmit a message using NIL to those for the Session Layer, determined from Figures 5.6 and 5.7, are tabulated for each host computer configuration, for messages of lengths 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 and 2048 characters.

NIL / Session Layer Transmission Times						
Char	VAX/VAX	mVAX/mVAX	SUN/SUN	VAX/mVAX	mVAX/SUN	SUN/VAX
2	.007	.007	.007	.013	.007	.019
4	.007	007	.007	.013	.007	.019
8	.007	.014	.007	.019	.014	.025
16	.019	.013	.014	.024	.014	.030
32	.042	.026	.027	.040	.027	.047
64	.079	.044	.050	.067	.044	.076
128	.129	.069	.084	.102	.074	.118
256	.189	.119	.137	.146	.126	.180
512	.248	.176	.198	.189	.184	.249
1024	.292	.241	.258	.223	.249	.312
2048	.321	.298	.304	.245	.303	.360

Figure 5.8 Ratio of NIL to Session Layer performances

The discrepancy between the effect of increased message length on the Session Layer and NIL performances is clearly indicated by the consistent increase in the performance ratio with increasing message length, demonstrated by all host configurations. This apparent discrepancy can be attributed to the fact that the overhead of the Session Layer, $\alpha + \tau$ is much larger than the overhead of NIL, α.

The performance ratio, tabulated in Figure 5.8, is illustrated graphically in Figure 5.9 for the VAX/VAX host configuration. A logarithmic scale is utilized for the message length axis to highlight the ratio at short message lengths. The ratio initially remains low, reflecting the relatively high overhead of the Session Layer over NIL. As the message length increases, the performance ratio increases, but not in such a way that an optimal message length for either the Session Layer or NIL can not be identified. The data revealed that as the message length approaches the size of the TCP transmit and receive buffer, 2048 bytes, the message transmission speed was almost solely dependent on the systems' efficiency in managing the buffers.



$$\text{Performance Ratio} = \frac{\text{NIL Transmission Time}}{\text{Session Layer Transmission Time}}$$

Figure 5.9 Performance ratio for VAX/VAX host pair

5.7 Summary

In this chapter, we presented a performance analysis of the Session Layer and Network Interface Library implementations. The analysis was structured to compare the relative performance of the Session Layer to that of NIL and consequently TCP based implementations. NIL was used as a performance standard because it is the fastest means of transporting a message over our network.

There are many factors which influence the performance of a distributed program. The factors examined included Transport Layer tuning, network contention, local versus remote communication, processor load, message length, and program overhead.

The most obvious result was that the Session Layer was much slower than NIL. This was anticipated because of the nature of the Session Layer's architecture which

depends on communication mediated by servers as opposed to NIL's which specifies direct links between end processes. The Session Layer was shown to exhibit a relatively significant message transmission time which is primarily due to a large overhead, that is, the time necessary to transmit a message of length zero. It was shown that the time to transmit messages via the Session Layer is directly proportional to CPU load and message length The analysis also suggested a dependence on host configuration, but no individual host could be identified as either enhancing or impeding the performance. The data indicated that high Session Layer transmission times were accompanied by high NIL transmission times, indicating that data transmission in general, and not the Session Layer's relative performance, could be impeded by any particular host. This phenomenon could be studied in the future.

The primary objective of the Session layer is to facilitate access to the network and standardize the method of doing so. Although performance is not a primary concern, we must consider whether or not the observed performance is acceptable for our applications.

In our opinion, the performance of the Session Layer and NIL, either separately or together is satisfactory. Under worst case conditions, which occur very rarely, we expect that the Session Layer's end-to-end message transmission time will not exceed 2.3 seconds for a 32 character message. The present bottlenecks in our work-cell applications are image processing and to a lesser extent, robot tasks which are usually of an order of magnitude and two or three times that of the Session Layer respectively Even if their speeds approached or were faster than that of the Session Layer, because of our distributed design philosophy the worst that could happen would be a very short pause in the work-cell activities. In our applications, task loads are partitioned in such a way as not to require high speed transactions, as would be needed for servo control or some types of image processing. These tasks are consolidated on the same host to avoid excessive network transactions. If a situation should ever arise that requires more efficient message passing, the user can always integrate NIL functions into their programs to provide better performance.

Chapter 6 Conclusion

In this thesis we have presented an interprocess communication environment that can serve as a tool for implementing distributed robot work-cell applications. The environment consists of two useful communication architectures which vary in complexity and performance, and offer a consistent network interface.

The primary communication architecture presented, a Session Layer implementation, offers a standard format for integrating interprocess communication primitives into sophisticated application programs; it also ensures compatibility so that programs written by different users can be integrated in work-cell applications. The secondary communication architecture, the Network Interface Library, offers very efficient end-to-end process communication.

An analysis of the implementations' performance shows that we have achieved acceptable end-to-end communication results without resorting to operating system modifications or custom hardware.

6.1 Future Research

In this section we suggest areas for future research which are natural extensions of the work presented in this thesis.

6.1.1 Operating System Resident Daemons

The current version of the Session Layer was implemented for feasibility purposes. A more refined version would have the Session Layer executing as an operating system daemon process on all hosts. The present configuration employs server processes, across only two hosts, that the user initiates prior to running his session programs.

A recommended implementation would involve designing the servers as "stateless machines" that could operate under conditions such as when a system crashes and reboots. A "stateless daemon" is one which does not have to remember from one transaction to the next any details of its state such as to what it is connected, or which task it has just performed. The major advantage of this approach is robustness. If a system fails, the Session Layer would be capable of taking the necessary action on its own, without operator intervention, to resynchronize itself with the other hosts and any system utility processes associated with the work-cell.

6.1.2 Reducing Overhead

By far the most demanding Session Layer constraint was preserving the integrity of the operating system. The most efficient way to reduce the overhead of the Session Layer would be to implement the message passing and signalling mechanisms in the operating system's kernel. This is in part, how TCP is able to be as effective as it is. Additional low level signalling mechanisms could also allow the efficient implementation of some type of message priority scheme.

The notion of the intermediary for message passing certainly is responsible for complicating the communication process, not to mention the obvious overhead that it creates. A low level message passing scheme would allow the elimination of the intermediary and therefore allow direct links to be established between end processes while still maintaining the flexibility and services that the servers provide.

Finially, overhead could also be reduced by implementing a transmission protocol other than TCP which is less robust and therefore faster.

6.1.3 Reducing System Processes

The present scheme permits a number of dormant server processes which are associated with dormant utility processes on any host. The dormant processes listen for connection requests and then activate themselves when they are needed. Under this scheme, at times, the operating can be over-burdened with an unnecessarily high number of processes doing nothing but listening for connection requests.

The Xerox Courier protocol can alleviate this problem [Xerox 85]. These processes could be eliminated if a single master server was responsible for listening for *all* requests, identifying which services are required, executing the appropriate server process, splicing the newly executed server process to the process that initially requested the connection, and then finally voiding its part in the transaction.

References

[Alford and Belyeu 84] Alford, C., Belyeu, S., "A Computer Control Structure for Coordination of Two Robot Arms." *Proc. American Control Conf.*, 1984.

[Andrews 82] Andrews, G., "The Distributed Programming Language SR - Mechanisms, Design, and Implementation." *Software Practice and Experience*, Vol. 12, 1982.

[Andrews and Schneider 83] Andrews, G., Schneider, F., "Concepts and Notations for Concurrent Programming." ACM Computing Surveys, Vol. 15, no. 1, March 1983.

[Baird et al. 84] Baird, H., Wells, E., Britton, D., "Coordination Software for Robotic Workcells." Proc. IEEE Int. Conf. on Robotics and Automation, 1984.

[Baker and Riccardi 85] Baker, T., Riccardi, G., "Ada Tasking: From Semantics to Efficient Implementation." IEEE Software, March, 1985.

[Barthes and Zavidovique 81] Barthes, J., Zavidovique, B., "How Much Intelligence should we expect from a Vision Processor in a Multi-processor Robot System ?." Proc. 1st Int. Conf. on Robot Vision and Sensory Controls, 1981.

[Bejczy and Lee 84] Bejczy, A., Lee, S., "Generalized Bilateral Control of Robot Arms." Proc. American Control Conf., 1984.

[Birrell and Nelson 83] Birrell, A., Nelson, B., "Implementing Remote Procedure Calls." Proc. 9th ACM Symp. on Operating Systems Principles, 1983.

[Bonner and Shin 82] Bonner, S., Shin, K.: "A Comparative Study of Robot Languages." IEEE Computer, December 1982.

[Bonin et al. 83] Bonin, J., Elloy, J., Haurat, A., Molinaro, P., Thomas, M., "The Tools for Synchronizaton of the LMAC System for the Cooperation of Industrial Robots." Proc. Int. Conf. on Advanced Software for Robotics, 1983.

[Brinch-Hansen 75] Brinch-Hansen, P., "The Programming Language Concurrent Pascal." IEEE Trans. on Software Eng., June 1975.

[Bruno et al. 84] Bruno, G., Demartini, C., Valenzano, A., "Communication and Programming Issues in Robotics Manufacturing Cells." Proc. IEEE Int. Conf. on Robotics and Automation, 1984.

[Carayannis 82] Carayannis, G., "Controlling a PUMA 260 Robot from a VAX Computer." Technical Report TR-83-3, Computer Vision and Robotics Lab., McGill University, April 1983.

[Chanson et al. 84] Chanson, S., Ravindran, K., Atkins, S., "Performance Evaluation of the ARPANET Transmission Control Protocol in a Local Area Network Environment," Technical Report 85–6, Department of Computer Science, The University of British Columbia, Vancouver, B.C., 1984.

[Chanson et al. 85] Chanson, S., Ravindran, K., Atkins, S., "LNTP – An Efficient Transport Protocol for Local Area Networks," Technical Report 85–4, Department of Computer Science, The University of British Columbia, Vancouver, B.C., 1985.

[Chen et al. 83] Chen, J., Loh, H., Chiang, K., Say, K., "A Software Package for Microrobot and Coordination between Robots with Voice Command," Proc IEEE Annual Conf. on Industrial Electronics (IECON), 1983.

[Clark 82] Clark, D., "Modularity and Efficiency in Protocol Implementation," Technical Report, Computer Systems and Communication Group, MIT Laboratory for Computer Science, 1982

[Devarakonda et al. 85] Devarakonda, M., McGrath, R., Campbell, R., and Kubitz, W., "Networking a Large Number of Workstations Using UNIX United," Proceedings of IEEE Workstations Conference, November 1985.

[Emmons and Chandler 83] Emmons, W., Chandler, A., "OSI Session Layer: services and protocols," Proc. IEEE, Vol 71, no. 12, December 1983.

[Draper et al. 66] Draper, N., Smith, H., *Applied Regression Analysis*, John Wiley and Sons, Inc., New York, N.Y., 1966.

[Dwivedi 83] Dwivedi, S., "Design of a Low Cost Remote Control Unit for a Robot," Proc. IEEE Southeastcon, 1983.

[Elfes and Talukdar 83] Elfes, A., Talukdar, S., "A Distributed Control System for the CMU Rover," Proc. 1983 IJCAI Conf., 1983.

[Elgazzar et al. 84] Elgazzar, S., Green, D., O'Hara, D., "A Vision-based Robot System using a Multiprocessor Controller," NRCC no. 23485, June 1984.

[Faro and Messina 83] Faro, A., Messina, G., "Robot Internetworking," IEEE Trans. on Industrial Electronics, November 1984.

[Faro and Messina 84] Faro, A., Messina, G., "Error Management in Robot Environment," Proc. IEEE TENCON (Trends in Electronics) Conf., 1984.

[Faro et al. 85] Faro, A., Mirabella, O., Vita, L., "A Multimicrocomputer-based Structure for Computer Networking," IEEE Micro, April 1985.

[Fathi et al. 83] Fathi, E., Krieger, M., "Multiple Microprocessor Systems: What, Why, and When," IEEE Micro, March 1983.

[Fisher and Weatherly 86] Fisher, D., Weatherly, R.: "Issues in the Design of a Distributed Operating System for Ada," IEEE Computer, May 1986.

[Garetti et al. 82] Garetti, P., Laface, P., Rivoira, S., "MODUSK: A Modular Distributed Operating System Kernel for Real-Time Process Control," Microprocessing and Microprogramming, Vol 9, 1982.

[Gauthier et al. 85] Gauthier, D., Carayannis, G., Freedman, P., Malowany, A., "A Session Layer Design for the CVaRL Local Area Network," Technical Report TR-85-7R, Computer Vision and Robotics Lab., McGill University, February 1985.

[Gentleman 83] Gentleman, W., "Using the HARMONY Operating System," NRCC no. 23030, December 1983.

[Goldwasser 84] Goldwasser, S., "Computer Architecture for Grasping," Proc. IEEE Int. Conf. on Robotics and Automation, 1984.

[Goldwasser and Bajcsy 83] Goldwasser, S., Bajcsy, R., "A Distributed Active Sensory Processor System," Proc. 3rd Scandinavian Conf. on Image Analysis, 1983.

[Gonzalez and Safabakhsh 82] Gonzalez, R., Safabakhsh, R., "Computer Vision Techniques for Industrial Applications and Robot Control," IEEE Computer, Vol. 15, no 12, December 1982.

[Green 83] Green, D., "CHORUS: A Multiprocessor Architecture for Real-Time Control Applications," NRCC no. 23031, December 1983.

[Gruver et al. 83] Gruver, W., Soroka, B., Craig, J., Turner, T., "Evaluation of Commercially Available Robot Programming Languages," Proc 13th Int. Symp. on Industrial Robots, 1983.

[Hanlon and Weston 82] Hanlon, P., Weston, R., "Use of Local Area Networks within Manufacturing Systems," Microprocessors and Microsystems, October 1982.

[Harmon 83] Harmon, S., "Co-ordination between Control and Knowledge-based Systems for Autonomous Vehicle Guidance," Proc. IEEE Trends and Applications Conf., 1983.

[Harmon and Gage 80] Harmon, S., Gage, D., "Protocols for Robot Communications. Transport and Content Layers," Proc. IEEE Int. Conf on Cybernetics and Society, 1980.

[Harmon et al. 84] Harmon, S., Gage, D., Aviles, W., Bianchini, G., "Coordination of Intelligent Subsystems in Complex Robots," Proc. IEEE 1st Conf. on Artificial Intelligence Applications, 1984.

[Hayward and Paul 83] Hayward, V., Paul R. P., "Robot Manipulator Control Using

the 'C' language under UNIX." Proc. IEEE Workshop on Languages for Automation, November 1983.

[Hoberecht 80] Hoberecht,V., "SNA Function Management," IEEE Trans. Communications, Vol. 28, no. 4, April 1980.

[Holland 83] Holland, J., "Local Area Networks: A Critical Part of Factory Automation," Proc. Numerical Control Society's Technical Conf. 1983.

[Holmgren 85] Holmgren, S., "The Untapped Potential of Remote Procedure Calls," UNIX Review, May 1985.

[ISO 7498] "OSI Basic Reference Model," ISO TC97/SC16, N.7498, 1982, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8072] "Information Processing Systems - OSI - Transport Protocol Specification," N.8073, 1985, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8073] "Information Processing Systems - OSI - Transport Service Definition," N.8073, 1985, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8326] "Information Processing Systems - OSI - Session Service Definition," N.8326, 1984, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8327] "Information Processing Systems - OSI - Basic Connection Oriented Session Protocol Specification," N.8327, 1984, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8473] "Data Comunications Protocol for Providing Connectionless Mode Internet Network Service," N.8473, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8571] "Information Processing Systems - OSI - File Transfer, Access, and Management," N.8571/1/2/3/4, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8649] "Information Processing Systems - OSI - Common Application Service elements Service Definition," N.8649, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[ISO 8802] "IEEE 802.1/2/3/4/5 LAN Standard Documents," N.8802, ISO Central Secretatiat, 1 rue de Varembe, 1211 Geneva 20, Switzerland.

[Iversen 85] Iversen, W., "Factory Automation Advances Apace," ElectronicsWeek, Feb 18, 1985.

[Jarvis 84] Jarvis, J., "Robotics," IEEE Computer, Vol. 17, no. 10, October 1984.

[Joseph 74] Joseph, E., "Distributed Function Computer Systems: Innovative Trends," Digest of Papers, IEEE COMPCON 1974, Spring, 1974.

[Kaminski 86] Kaminski, M., "Protocols for Communicating in the Factory," IEEE Spectrum, April 1986.

[Kärkkäinen 83] Kärkkäinen, P., "A Sensor Information Preprocessing System for Manipulators based on Distributed Microcomputers," Proc. Conf. on Advanced Software in Robotics, 1983.

[Kärkkäinen and Manninen 83] Kärkkäinen, P., Manninen, M., "A Hierarchical Distributed Information Processing System for Forest Manipulation," Proc. IFAC Real Time Digital Control Applications Conf., 1983.

[Leopold 84] Leopold, G., "Factory Nets Follow a MAP," ElectronicsWeek, Dec. 17, 1984.

[Lloyd 85] Lloyd, J., "Implementation of a Robot Development Environment," M. Eng Thesis, McGill University, Montreal, Quebec, 1985.

[Michaud et al. 85] Michaud, C., Malowany, A., Levine, M "Multi-robot Assembly of Electronic Circuits," Proc. Graphics Interface 85, 1985.

[Ma and Krishnamurti 84] Ma, Y.-W., Krishnamurti, R., "REPLICA – A Reconfigurable Partionable Highly Parallel Computer Architecture for Active Multi-sensory Perception of 3-Dimensional Objects," Proc. IEEE Int. Conf. on Robots and Automation, 1984.

[MacWilliams et al. 84] MacWilliams, P., Wolochow P., Zasloff R., "Microcontroller Serial Bus Yields Distributed Multispeed Control," Electronics, Feb. 9, 1984.

[Maimon 85] Maimon, O., "A Multi-robot Control Experimental System with Random Parts Arrival," Proc. IEEE Int. Conf on Robots and Automation, 1985

[Maimon and Nof 83] Maimon, O., Nof, S., "Activity Controller for a Multiple Robot Assembly Cell," Proc. Winter Annual Meeting of the ASME, 1983.

[Makhlin 82] Makhlin, A., "Vision Controlled Assembly by a Multiple Manipulator Robot," Proc. 2nd Int. Conf on Robot Vision and Sensory Controls, 1982.

[Mao and Yeh 80] Mao, T., Yeh, R., "Communication Port: A Language concept for Concurrent Programming," IEEE Transactions on Software Engineering, March, 1980.

[MAP 85] "Manufacturing Automation Protocol, Version 2.1," Manufacturing Engineering and Development, General Motors Technical Center, Manufacturing Building,

A/MD-39, 30300 Mound Road, Warren, MI, 48090-9040, 1985.

[McQuillan and Walden 77] McQuillan, J., Walden, D., "The ARPA Network Design Decisions," Computer Networks, Vol.1, August 1977.

[Metcalfe and Boggs 76] Metcalfe,R., Boggs, D.,"ETHERNET: Distributed Packet Switching for Local Computer Networks," Comm. ACM, Vol.19, July 1976.

[Michaud 85] Michaud, C., "MROUTINES.C: Using the Microbo Robot with Style," Technical Report TR-85-3, Computer Vision and Robotics Lab., McGill University, January 1985.

[Milne 83] Milne, B., "Board Testing: The Future is Software, Networking," Electronic Design, Nov. 24, 1983

[Nagel 83] Nagel, R. "Robots: not yet smart enough," IEEE Spectrum, Vol. 20, no. 5, May 1983.

[Neumann 83] Neumann, J., "OSI Transport and Session Layer Services and Protocols," Proc. IEEE Infocom Conf., 1983.

[Pearson and Green 84] Pearson, J., Green, D., "Three Channel Infrared Wireless Link," Proc. IEEE Southeastcon, 1984.

[Plessmann 83] Plessmann, K., "A Multi-microcomputer-based Robot Control System," Proc. IFAC Real Time Digital Control Applications Conf., 1983.

[SAS 82] SAS Institute Inc., *SAS User's Guide: Statistics, 1982 Edition*, SAS Institute, Inc., Cary, North Carolina, 1982.

[Searle 71] Searle, S., *Linear Models*, John Wiley and Sons, Inc., New York, N. Y., 1971.

[Shatz 84] Shatz, S., "Communication Mechanisms for Programming Distributed Systems," IEEE Computer, June 1984.

[Shimano et al. 84] Shimano, B., Geschke, C., Spalding, C., "VAL II: A New Robot Control System for Automatic Manufacturing," Proc. IEEE Int. Conf. on Robotics and Automation, 1984.

[Shin and Epstein 85] Shin, K., Epstein, M., "Communication Primitives for a Distributed Multi-robot System," Proc. IEEE Int. Conf. on Robotics and Automation, 1985.

[Silverman 84] Silverman, J., "Communications in a Distributed Computer Testbed," IEEE Int. Conf. on Distributed Computing Systems, 1984.

[SRI 82] Transmission Control Procedure/Internet Protocol, Internet Workbook, Network Information Center, SRI International, March 1982.

[Stankovic 82] Stankovic, J., "Software Communication Mechanisms. Procedure Calls versus Messages," IEEE Computer, Vol. 15, no. 4, April 1982

[Stauffer 85] Stauffer, R., "Three-robot Workcells Help Assemble Solid-state Ignition Modules," Robotics Today, February 1985.

[Storoshchuk et al. 83] Storoshchuk, O., Szabados, B., "Industrial Local Area Network," Proc. IEEE Int. Electrical and Electronics Conf., 1983.

[Sun 85] Sun Microsystems Inc., "Networking on the Sun Workstation – Inter-Process Comunication Primer," Part No. 800-1177-01, Release 2, May 1985.

[Tannenbaum 81] Tannenbaum, A., *Computer Networks*, Prentice-Hall, 1981.

[Taylor et al. 82] Taylor, R., Summers, P., Meyer, J., "AML: A Manufacturing Language," Int. Journal of Robotics Research, Fall 1982.

[Tuthill 85] Tuthill, B., "IPC Facilities in 4.2BSD," Unix Review, April 1985.

[UCB 83] 4.2BSD System Manual, Computer Systems Research Group, University of California - Berkeley, July 1983. ACM Symp. on Operating Systems Principles, 1983.

[Unimation 86] Unimation Inc., "Robot Controller Under development," IEEE Computer, Vol. 19, no. 6, June, 1986.

[Voelcker 86] Voelcker, J., "Helping Computers Communicate," IEEE Spectrum, April 1986.

[Volz et al. 84] Volz, R., Mudge, T., Gal, D., "Using ADA as a Programming Language for Robot-based Manufacturing Cells," IEEE Trans. on System, Man, and Cybernetics, November/December 1984.

[Wainwright and Moss 85] Wainwright, R., Moss, R., "A Microcomputer-based Model Robot System with Pulse-width Modulation Control," IEEE Micro, February 1985

[Willis and Sanderson 84] Willis, J., Sanderson, A., "RAPIDbus: Design of an Extensible Multiprocessor Structure," Technical Report no. 84-13 from The Robotics Institute of Carnegie-Mellon University, 1984.

[Wirth 77] Wirth, N., "Modula - A Language for Modular Multiprogramming," Software Practice and Experience, January 1977.

[Xerox 85] *Xerox Network Systems Architecture – General Information Manual*, XNSG 068504, Xerox Corporation, 1985.

[Zimmermann 80] Zimmermann, H., "OSI Reference Model - The ISO Model for Open Systems Interconnection," IEEE Trans. Comm., April 1980.