To

From

Date JUN 14 1988

memo

University of Waterloo

CS-88-11

gave 30 copies

to Terry

Stepien

Printing Requisition / Graphic Services

19935

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number, to extension 3451.**

TITLE OR DESCRIPTION
CS-88-11 A Communication System for Local Area Networks

DATE REQUISITIONED: May 31/88
DATE REQUIRED: ASAP
ACCOUNT NO.: 1 2 6 6 0 4 0 4 1

REQUISITIONER – PRINT: D. Cowan
PHONE: 4467
SIGNING AUTHORITY: Sue DeAngelis / D. Cowan

MAILING INFO – NAME: Sue DeAngelis
DEPT.: C.S.
BLDG. & ROOM NO.: DC 2314
[X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 45
NUMBER OF COPIES: 100

TYPE OF PAPER STOCK: [ ] BOND [ ] NCR ____ PT. [ ] COVER [ ] BRISTOL [ ] SUPPLIED [ ] Alpac Ivory 140M

PAPER SIZE: [ ] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] 10x14 Glossocat 10 pt Rolland Tint

PAPER COLOUR: [ ] WHITE [X] ____ INK: [X] BLACK [ ] ____

PRINTING: [ ] 1 SIDE ____ PGS. [X] 2 SIDES ____ PGS.
NUMBERING: FROM ____ TO ____

BINDING/FINISHING: [X] COLLATING [ ] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING: 7x10 saddle stitched
CUTTING SIZE:

Special Instructions

Beaver Cover

Both cover and inside in black ink please

Thank you

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

			D 0 1
			D 0 1
			D 0 1

TYPESETTING — QUANTITY

P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1

PROOF

P R F				
P R F				
P R F				

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1

PMT

P M T				C 0 1
P M T				C 0 1
P M T				C 0 1

PLATES

P L T				P 0 1
P L T				P 0 1
P L T				P 0 1

STOCK

				0 0 1
				0 0 1
				0 0 1
				0 0 1

BINDERY

R N G				B 0 1
R N G				B 0 1
R N G				B 0 1
M I S 0 0 0 0 0				B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

GRAPHIC SERV. OCT. 85 482-2

May 30/88
3:15

Sue,

It turns out that the thesis I gave you this morning (Terry Stepien's) is to be in duck rept form. (Beaver cover). Sorry if this inconveniences you.

Thanks Mary R.

Ps. you already gave him the report #.

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

A Communication System for Local Area Networks

Terry M. Stepien

Research Report
CS-88-11

May, 1988

A Communication System
for
Local Area Networks

Terry M. Stepien

Department of Computer Science

University of Waterloo

Waterloo, Ontario, 1988

ABSTRACT

Even with powerful independent workstations there is a need for connectivity to high speed local area networks. These networks are required to allow the sharing of information and expensive peripherals.

This paper describes the design and implementation of a layered communication system for local area networks which is designed to provide an efficient modular approach to data communications. The process and message passing model is used to implement the layers in the communication system.

The communication system is designed to provide the transport mechanism for various applications. One such application, called Waterloo JANET, which manages the resources of a local area network, has been implemented to demonstrate the viability of the communication system.

The communication system is implemented for the IBM Personal Computer family of computers running the PC/DOS operating system version 3.3. The IBM PC Network Baseband adapter and the IBM PC Cluster adapter form the basis of the network hardware. Although the communication system was implemented on a specific hardware and software family, the approach should easily generalize.

ACKNOWLEDGEMENTS

I wish to thank my supervisor, Prof. D.D. Cowan, for his advice, constructive criticisms and tireless readings of this paper.

Eric Mackie was a source of considerable assistance both as director of the Computer Systems Group and for his valuable comments.

All members of the Computer Systems Group deserve thanks for their co-operation and assistance, especially Peter Bumbulis and Rob Veitch.

Special thanks to my wife Nancy for her motivation and encouragement during the writing of this paper.

Finally, I wish to thank the Waterloo Foundation for the Advancement of Computing for providing fellowship support.

CONTENTS

FIGURES

INTRODUCTION

Microcomputer workstations interconnected by local area networks (LANs) are often becoming the systems of choice to satisfy the ever increasing computing needs of user communities. These powerful workstations are being used to solve many of the problems once solved on larger mainframe systems.

The distributed nature of the microcomputer workstation environment presents problems in sharing and with the administration of these systems. A well designed and implemented communication system is required to provide the framework for the network applications to address these problems. The design of such a system must deal with the many difficulties associated with remote communications. As stated by Svobodova [Svob85], "One of the major problems in designing distributed systems is how to deal with the uncertainties caused by component failure and imperfect communications."

In this paper we demonstrate that a communication system for local area networks can be efficiently implemented in layers. We also show that the process and message passing model is a suitable implementation method for the layers of a local area network. As well we investigate the issues associated with making a communication system for local area networks which can be used by a number of applications. One such application, the Waterloo JANET local area network package, is used to demonstrate the feasibility of the communication system.

The Waterloo JANET system was designed to interconnect a number of microcomputer workstations with file, print and communication servers. The servers provide the user with application software, files and shared printing devices over a number of interconnected local area networks.

In order to demonstrate the design of this communication system the following topics are examined:

- communication models for distributed processing,
- a simple protocol for local area network based communications,
- a modular approach for communications,
- the Waterloo JANET local area network.

Motivation

The original motivation for building a communication system for local area networks came from the need to interconnect microcomputer workstations in an educational setting to allow sharing of application software, files, and printing capacity.

The ability to share files and devices is one of the primary reasons for installing a network. An extensive discussion of sharing and other rationale for local area networks in an educational environment is provided in [Cowa87a]. An extensive discussion of the design of file servers in a network setting and the type of services they should provide is presented in [Svob84]. The functions in the local area network must be provided to a level consistent with the facilities provided in a mainframe environment. Thus in designing a distributed environment, such as a local area network, we must provide the same level of file sharing as is provided in a mainframe or centralized environment.

An existing operating system, such as PC/DOS, is a base for the communication system, since it already supports an extensive number of applications, an important attribute in an educational environment. Existing standalone applications must be compatible with the network environment; for example,

applications such as word processors, language compilers, spreadsheets and data bases must execute transparently on the network. In other words, application programs should not require any modifications to be compatible with the network environment.

The existing operating system is extended to allow a modular approach to accommodate the network communication system. Modularity is required for two reasons. First, new network based applications should be able to connect to the communication system without the need to duplicate existing function. Second, the ability to add or change existing communication layers dynamically during development of new communication protocols is a desireable feature to facilitate experimentation.

The design of the communication system could follow either of the following two approaches. The first approach is called "the integrated approach" [Cheu86]. In this design the entities in the communication system are integrated with the application to form a vertically layered solution. The advantages of this approach are usually improved performance and reduced size for the implementation. The disadvantages of the integrated approach are the reduced modularity and the increased complexity of considering all communication issues in one layer.

In the second approach, called "the layered approach" [Svob86a], communication functions are provided by different layers. The facilities in an upper layer use only those primitives defined in the lower layer. Two possible approaches to layering are based on the use of general versus special purpose protocols in the communication system. The implementation of general purpose protocols in a workstation involves each layer in the communication system implementing the standard protocols for data exchange at that layer. The advantage of this approach is that the completed communication system would be completely compatible with networks from other manufacturers which adhered to the same standards. The disadvantage is that the existing standard protocols are designed to handle the general communications problems which arise in wide area networks and do not take advantage of the features in underlying local area networks. Hence, performance of the system is traded for portability.

The "special purpose protocol approach" is also a layered approach, but special purpose protocols designed to take advantage of the architecture of the LAN are used in place of the general purpose protocols. The advantage of this approach is that it maintains the modularity of the layered approach plus it has much of the enhanced performance and reduced size of the integrated approach. For example, special purpose protocols can be designed to take advantage of transmission characteristics of a local area network which would be ignored in the general purpose protocol. Also, since the special purpose protocols are customized for the local area network hardware, some of the unused features of general purpose protocols may be omitted from the implementation to reduce the size and complexity of the implementation. The disadvantage of this approach is that workstations on the network cannot be added directly to existing networks which use the general purpose protocols. However, the effect of this disadvantage is minimized if we follow an approach similar to the one described by Cheriton [Cheri84a]. A gateway station may be used to translate the special purpose protocols of a particular local area network to the general purpose protocols available on other systems. In this paper we describe the implementation of a PC/DOS based communication system which follows the layered approach with special purpose protocols.

Overview of Accomplishments

In this paper we describe the design and implementation of a communication system for interconnected local area networks, which demonstrates the feasibility of a layered approach based on the OSI, Internet and IEEE 802 Standards.

Further, each layer in the communication system is made highly modular through the use of cooperating sequential processes which communicate by passing messages. The communicating sequential

processes have been implemented using a set of message passing primitives and a real-time kernel which have been added to an existing operating system, namely PC/DOS.

The transport layer on the network uses special purpose protocols and "light weight" connections to take advantage of the medium-access-control architecture of most local area network structures.

An application, called Waterloo JANET, which manages the resources of a local area network has also been implemented to demonstrate the viability of our approach.

Finally, the performance of the communication system and the Waterloo JANET application have been evaluated and found to justify our design approach.

Organization

This chapter explains the motivation for the paper. Chapter 2 describes communication systems for local area networks and an overview of the existing models for such systems. Chapter 3 describes a model for interprocess communication, the implementation, and the performance of such a model. Chapter 4 describes a process-based layered communication system, the implementation, and the performance of such a system. Chapter 5 describes Waterloo JANET, an application which uses the communication system. Chapter 6 summarizes our work and gives some directions for further investigation. The appendices are divided into two parts. Appendix A through Appendix D are the structure definitions for the communication system. Appendix A contains the structure definitions for the real-time message passing kernel. Appendix B provides the definitions for the queue administrator. Appendix C provides the definitions for the packet assembly/disassembly process. Appendix D contains the Waterloo JANET packet definitions. Appendices E and F contain a coding framework for a sample network port and the queue administrator.

COMMUNICATION MODELS FOR DISTRIBUTED PROCESSING

Introduction

The description of the functions required to perform communications between two machines are often considered too complex to be contained in a single unit. This leads to the realization that the communication functions may be conveniently described by a set of layers [Zimm80]. Each layer performs a certain subset of the functions required to communicate with another system.

A widely accepted model for describing this type of structure is the seven layer reference model for Open System Interconnection from the International Organization for Standardization. The layered approach for communications and the functions performed at each layer are described in this chapter.

An alternative model for communications has resulted from the extensive research and practical experience of the ARPANET project. This model proposes the use of a four layer hierarchy of functions. The hierarchical approach is defined and the functions performed at each of the layers are explained in this chapter. The differences between the layered and the hierarchical model are indicated.

Both the OSI layered model and the hierarchical model were developed to describe communication systems for wide area networks. A subset of the OSI model designed for local area network communications was developed by the IEEE 802 committee. The 802 LAN reference model follows the philosophy of the OSI model, but the IEEE 802 committee were concerned with only the lowest two layers of the

OSI model The 802 LAN reference model was designed to take advantage of unique communication characteristics in local area network environments.

All these models are included in this chapter since our communication system uses concepts from each of them.

The OSI Layered Approach

Zimmerman [Zimm80] described principles used to arrive at a seven layer model for communications. A common analogy is an **onion skin,** each subsequent layer conceals the contents of the inner layer. Between each layer there is a protocol. Most communication systems have made some use of a layering structure. Each layer performs a certain function in the transformation. However only the first layer has a physical connection. The other layers have software connections to each other. In an effort to standardize the function at each layer, the International Organization for Standardization (ISO) defined a seven layer model. The standard which describes the specific functions to be performed at each layer is called the Model of Open System Interconnection (OSI). This model is not universally accepted; however, it is a useful model when discussing network design. The layers of the model are described in the following section.

The OSI Reference Model consists of seven layers:

1. physical layer
2. data link layer
3. network layer
4. transport layer
5. session layer
6. presentation layer
7. application layer

Each layer k on one machine communicates with layer k-1 on the same machine using the layer k interface between layer k and layer k-1. The communication of data between two machines A and B can be pictured as being between layer k on machine A and layer k on machine B. A data unit originating at the application layer on host A is passed down through the layers. Each layer adds control information in the form of a header to the data unit. When the data unit arrives at machine B, each header is removed as the data unit is passed up through the layers until just the data unit originally sent arrives at the application layer on host B. Of course, some layers may also subdivide the data unit into smaller fragments and other layers may reassemble the fragments into data units as part of their function. In other words, communication is achieved by having corresponding "peer" entities in the same layer on two different systems communicate via a protocol. Figure 1 illustrates the OSI seven layer model.
Following is a description of the functions performed by each layer in the OSI model.

Physical: The physical layer provides a connection between data link entities for transmission of data. It performs the encoding of the data for transmission and regulates access to the physical network.

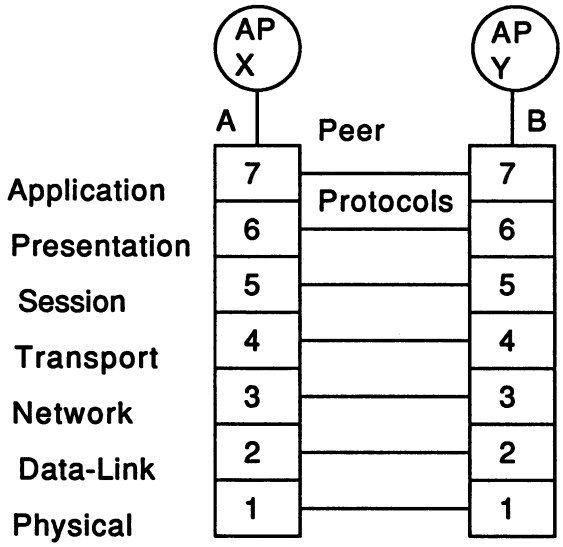

Figure 1: Layered Model

Data Link: The data link layer provides error-free communication across the physical link between adjacent nodes. Fixed length frames are transmitted along with a checksum and addressing information. When the data link layer on an adjacent machine receives a valid frame it passes an acknowledgement back to the sender.

Network: The primary function of the network layer is routing. Packets from the transport layer must be directed to the appropriate destination using header information in the packet. The services provided by the network layer are independent of the distance between the machines.

Transport: The transport layer provides reliable end-to-end transmission for arbitrary length messages. The size and complexity of this layer depends on the type of services provided by the network layer.

Session: The session layer provides process to process communication between machines. This involves the establishment, management and termination of sessions or virtual connections.

Presentation: The presentation layer provides for transformation of the data such as compression, encryption and code conversion.

Application: The application layer includes all application systems that may want to communicate with another machine on the network. Examples include terminal servers, printer servers and file servers.

The Hierarchical Approach

One set of protocols which defines the layers of a hierarchical model is called the "Internet protocol suite" [Hedr87]. These protocols were developed as a result of research and experience gained in the ARPANET project.

The Internet protocols are based on a three-part view of communication. The three parts are processes, hosts, and networks. Processes are the entities which execute on hosts and desire to communicate with processes on other hosts. To facilitate this communication the two hosts are attached to a network.

The transfer of information between processes can be divided into three parts. First the information is directed to the correct network. Second, the data is directed to the correct host on the network. Finally, the message is directed to the correct process on the host. Each component of the demultiplexing may be handled independently. The network is only required to route information between hosts.

The four layer architecture which results from the hierarchical structure has:

1. a network access layer
2. an internet layer
3. a host-host layer
4. a process/application layer

An entity at layer k may use the services of another entity at the same layer or directly use the services of lower layers. Like the OSI model each layer adds header information onto the data unit. When the data unit augmented with headers arrives at the destination host, the headers are removed as the data unit is passed up through the layers. In the hierarchical model data units may also be subdivided and reassembled as they pass through some of the layers. The following is a description of the functions performed by each layer in the Internet model.

Network Access Layer: The network access layer contains the protocols required to access the communication network. The goal of the network access layer is to route data between hosts on the same network.

Internet Layer: The internet layer is responsible for routing data between hosts on multiple networks. This layer adds the concept of a gateway which connects network access layers for two networks. This allows the transfer of data to occur between networks.

Host-Host Layer: The host-host layer contains a set of protocols which provide reliable transfer of data between two processes on different host computers.

Process/Application Layer: The process/application layer contains all the protocols for resource sharing. Examples of applications at this layer include file transfer and remote login. Figure 2 shows the layers of the hierarchical model and the names of common protocols at each layer.

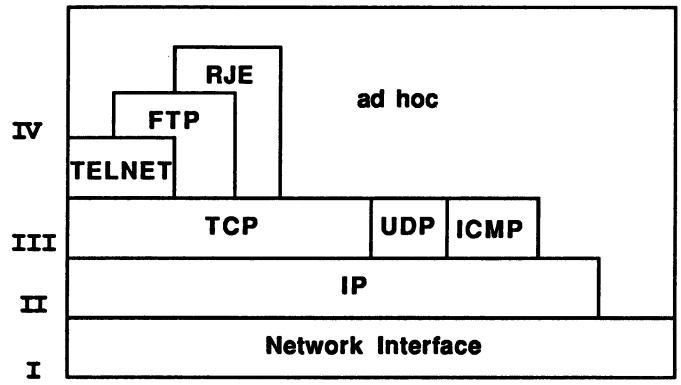

Figure 2: Hierarchical Model

Internet Versus OSI

Although the Internet hierarchical and the OSI layered model have the same final goal of providing communications between machines, they have three fundamental differences in approach. These differences are hierarchy versus layering, importance of internetworking and the utility of connectionless service.

Both the OSI and the Internet models follow the approach that communications are too complex for a single unit. The communication problem is decomposed into modules or entities. The OSI model restricts a layer k to use only those functions provided by layer k-1. The Internet model is not as restrictive. An entity in the Internet model may directly use the services of any entity lower in the hierarchy. The importance of this distinction is best illustrated by an example. Consider the communication system for a local area network, the broadcast packet facility of a LAN is a function normally not found in a wide area network. Hence, an interface would not be defined for this function and access to the broadcast function would be denied to the upper layers in the OSI model. The additional flexibility in the Internet model allows the protocol designer more freedom to develop efficient and cost-effective protocols.

The initial design of the Internet model included internetworking as an integral component. A design parameter of the model was to allow for a hierarchy of interconnected networks. The internet protocols (IP) were included to accommodate this goal. The concept of internetworking was not included in the initial definition of the OSI model.

Although the OSI reference model supports both connectionless and connection oriented services, many current implementations favour the use of connection services. The Internet model also supports

both connectionless and connection-oriented services. However, the connectionless service is an integral component of the internetworking support in the Internet model. Services such as name binding and urgent datagrams make valuable use of connectionless services.

The relationship between the layers in the OSI reference model and the layers in the Internet model are shown in Figure 3. Each of the layers in the Internet model integrates functions from a set of layers in the OSI model. By combining the layers the amount of overhead resulting from layer interfaces is reduced.

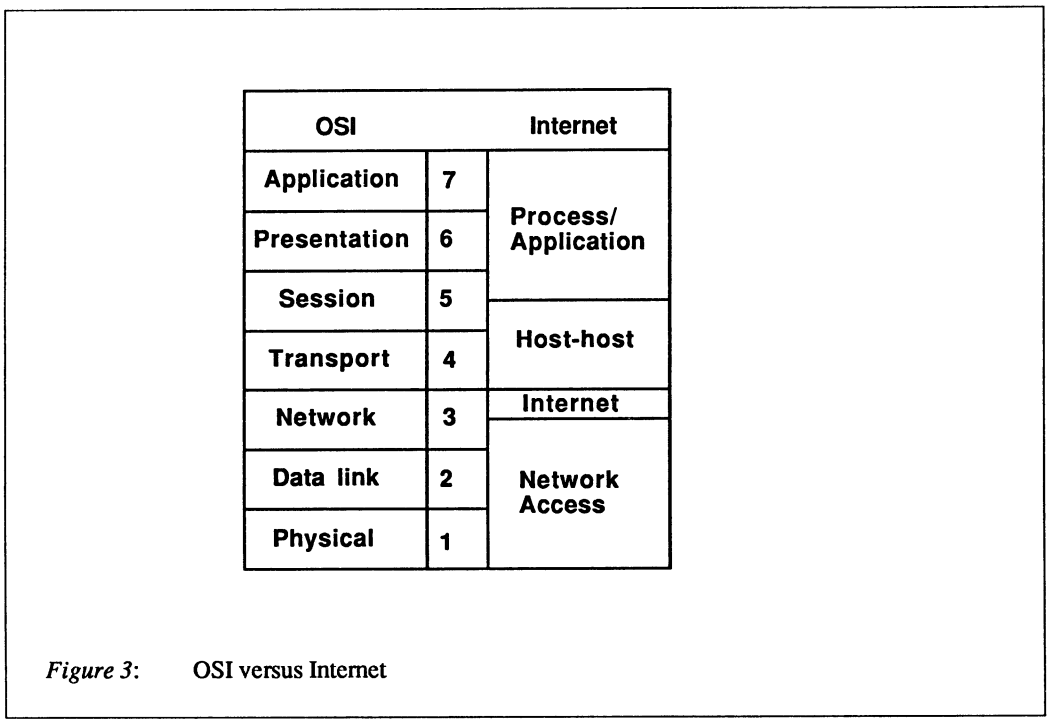

Figure 3: OSI versus Internet

LAN Approach

The IEEE 802 committee was formed to develop a set of protocols to define how devices could communicate over a local area network. A local area network is distinguished from the data networks described in the previous sections by the following three points:

- spans a small geographic area
- high data transfer rate
- low error rate

The 802 LAN model follows the philosophy of the OSI model but the IEEE 802 committee was concerned with only the lowest two layers. A subset of the OSI reference model with the following three layers was developed for local area networks:

1. physical
2. medium access control
3. logical link control

An application connects to a service access point (SAP), which is the interface to the logical link control layer, to communicate with a destination SAP on another machine. A data unit is passed from the logical link control layer to the medium access control and then to the physical layer for transmission. Each layer encapsulates the data unit with the appropriate header information. Figure 4 illustrates the 802 local area network reference model.

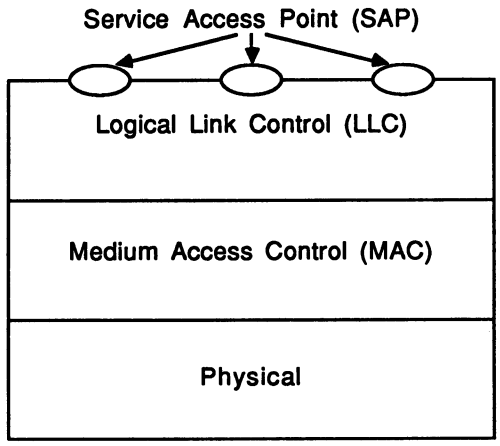

Figure 4: 802 LAN Model

The following is a description of the functions performed at each layer.

Physical Layer: The physical layer is concerned with the details of the transmission medium and electrical signaling.

Medium Access Control Layer: The medium access control layer handles the multiplexing and demultiplexing of the access to the shared transmission medium.

Logical Link Control Layer: The logical link control layer is concerned with the establishment, maintenance and termination of links between communicating service access points.

General Comments

The 802 LAN reference model describes a system which is adequate for communications on a single local area network. The layers above the logical link control are not defined. There is proposed support for internetworking in the 802 LAN model dealing with interconnection of homogeneous networks at the media access control layer. A goal of our communication system is to support multiple heterogeneous local area network media access control layers in one network. Hence, the 802 LAN reference is not sufficient for our communication system. However, if we include the internetworking and higher layers above the network access layer of the Internet model with the 802 LAN model we arrive at a model for our communication system. Our model differs from the standards; however, it borrows ideas on layering and hierarchy from the wide area network standards and LAN access from the local area network standard, in particular those standards from ISO, ARPANET and the ANSI 802 committee.

A MODEL FOR INTERPROCESS COMMUNICATION

Introduction

The components of each layer of the LAN communication system are co-operating sequential processes. In the implementation of the system we decide how co-operating processes should be used to provide both efficiency and portability. Since the target operating system did not have primitives for co-operating sequential processes we chose to provide such an extension for the PC/DOS operating system. We describe the syntax and semantics of the message passing primitives followed by a discussion of implementation and performance considerations. An overview of the extension is also described at the end of this chapter.

Process Model

The use of processes is central to the design of most modern operating systems. Many models for processes and interprocess communications have been suggested and implemented. Each model has its own definitions for a process and interprocess communications primitives. The model for our operating system extension consists of processes which communicate by passing messages and is similar to the one described by Gentleman [Gent81] and used in the V Kernel [Cheri84] and Thoth [Cheri79]. This model has been shown to be sufficient for many applications.

The process abstraction is implemented with a kernel containing a real-time scheduler which provides for inexpensive context switching. The kernel contains four primitives which implement interprocess communication on the same machine and which cause suspension of a process when a message is sent or received. In addition, process management primitives are provided for the creation and destruction of processes. The following sections provide descriptions in the language C of the primitives supported by the kernel and the data structures used.

Messages

Messages are composed of two components, the header of a message, and the contents of the message. The header contains two sub-component fields, the process id of the destination process and the size of the the message following the length field. A typical definition of the type message, using data types in the language C, is of the form:

```
typedef struct a_msg {
    a_pid       partner;
    unsigned    length;
    int         number[40];
} a_msg;
```

where partner is the name of the receiving process and is of type a_pid, and length is an unsigned integer. The contents of the message may be any legal data type including a structure. The definition of a_pid may be found in Appendix A.

Process Communication Primitives

The Send Primitive: The Send primitive sends a message from one process to another. All the information is contained in a variable which is defined to be of type a_msg. The name of the process receiving the message is contained in the partner component of the message header. The Send primitive has a pointer to the message as its single argument.

A typical use of the Send primitive is of the form:

```
status = Send( &message );
```

where &message is a pointer to a variable of type a_msg. Send returns the value zero if the Send fails, and non-zero if Send succeeds. Send would fail if there is no process with the process id specified in the message. Once the Send primitive is executed, the process sending the message is blocked until a Reply to the message is received.

The Receive Primitive: The Receive primitive receives a message sent from any other process. Receive does not know the identity of the sending process until the message arrives. A process executing the Receive primitive is blocked until a message arrives from a sending process.

The arriving message is copied to a message buffer in the receiving process and the name of the sending process is placed in the partner field of the message. This allows the receiving process to know the identity of the sending process.

A receiving process must specify the maximum size of the message it expects before the Receive is executed. The Receive primitive has two parameters, a pointer to the received message and a time-out value. Once the Receive primitive is executed, the process waiting to receive a message is blocked until a message actually arrives or the time-out value is exceeded. If a time-out value of zero is specified, then Receive blocks until a message is received.

A typical use of the Receive primitive is of the form:

```
message.length = msgsize( message );
signal = Receive( &message, delay );
```

where msgsize calculates the size of the structure represented by message. The variable signal, of type unsigned, would have the value zero if Receive was unblocked because of the arrival of a message, and non-zero if Receive was unblocked because the time-out value was exceeded.

The Reply Primitive: The Reply primitive responds to a message sent by a process. When the sending process receives a reply to its message, the sending process is unblocked. The Reply primitive is non-blocking. A simple Receive-Reply sequence is given in the following example:

```
msg.length = msgsize( msg );
Receive( &msg, delay );
status = Reply( &msg );
```

The unsigned variable status would have value zero if the Reply fails, and non-zero if Reply succeeds. Failure would occur if there is no process with the process identifier contained in the message, or if the process named in the message was not waiting for a reply.

Replies to messages do not have to be given in the order received, and messages in the reply may be different than the message that was originally received. The address of the sender must be in the partner field of the header and is automatically placed there when the message is received. This allows a receiving process to return a message to the sender.

The Signal Primitive: The Signal primitive signals a specific process that an event has transpired. The Signal primitive has two parameters, the process identifier of the process being signalled, and the value of the signal. The signal is an unsigned integer and each bit or combination of bits in the integer can be assigned a meaning. An example of a signal is:

```
Signal( process_id, 0x4000 );
```

Signals are non-blocking and they can be sent from any number of processes. Provided that each process sends a unique signal, signals are collected for the receiving process and the union of all signals since the last Receive are returned.

A signal will cause a process blocked on a receive to be unblocked. The process issuing the signal is not blocked by the signal primitive.

Process Management Primitives

The Priority of Processes: A process will block whenever it executes a Send or a Receive primitive to send or receive a message. A process will unblock if it received a message or signal while blocked on a Receive, or if it received a reply while blocked on a Send.

When a process is unblocked it is placed in a queue of processes waiting to be run. The next process to be run or dispatched is selected from the queue using a FIFO discipline. Processes are assigned priorities when they are created and processes in a higher priority queue are always dispatched first. A process will only surrender its use of the processor if the process becomes blocked, or if a process of a higher priority is waiting to be dispatched. Thus it is possible for a process which does not execute a Send or Receive primitive, to monopolize the processor for a long time and prevent all other processes at the same or lower priorities from running.

Most processes are spawned to run at NORMAL_PRIORITY. Processes can be spawned to run at higher priority by adding multiples of a UNIT_PRIORITY to the priority field in the Spawn procedure. NORMAL_PRIORITY, UNIT_PRIORITY and MAX_PRIORITY are defined in appendix A.

The Spawn Primitive: The Spawn primitive creates a process that can now run. The Spawn primitive has four parameters:

1. a pointer to a stack for the process;
2. the priority of the process;
3. the name of the process;
4. the start of the code for the process.

The statement:

```
pid = Spawn( GetStk(1024), NORMAL_PRIORITY, 3, Main );
```

creates a process with 1024 bytes of stack space, normal priority, a user assigned logical name of 3 and a starting address specified by the label Main.

A unique process identifier (pid) is returned as the result of the Spawn primitive. The process identifier is used to distinguish the process from other processes on the machine. A process may be assigned a logical name when it is created by the Spawn primitive. The logical name provides a compile time name binding which may be used to identify the process in any process communications. The logical name ensures that you will know how to communicate with a process. The logical names are most often used to identify system service processes which are always available on the workstation. The pid and logical name of a process can be used synonymously for communications with the process.

The Kill Primitive: The Kill primitive destroys a process and removes it from the system. The statement:

```
status = Kill( pid );
```

will destroy a process whose process identifier has the value of the variable pid. The unsigned variable status will have the value zero if Kill fails and non-zero if Kill succeeds. Failure would occur if there is no process with process identifier pid.

Implementation Considerations

In order to implement a process-based system we require a real-time kernel which provides fast context-switching. Each process has an associated process descriptor which maintains the current state for the process. A process may be in any of the following four states:

- Ready
- Awaiting Send
- Awaiting Receive
- Awaiting Reply

When a process is created it is placed in the Ready state. This indicates the process is prepared to be dispatched or run whenever the processor is available. Issuing a Send primitive will cause a process to block

and the state for the process to be changed to Awaiting Receive. The process will remain in this state until the receiving process issues a Receive primitive.

When the receiving process issues a Receive the previously sent message is received and the state of the sending process is modified to be Awaiting Reply. If there were no sends waiting when the Receive primitive was issued, the state of the receiving process would be set to Awaiting Send and the process would block until a message arrives or the specified time-out period expired. The sending process remains blocked until a Reply primitive is executed which changes the state of the blocked process to Ready.

Integration with PC/DOS: The kernel is implemented as a terminate and stay resident (TSR) program under the PC/DOS operating system. Since PC/DOS is only serially re-useable, a process must not give up control during an operating system call. Some preliminary research has indicated this problem may be resolved by creating a PC/DOS file system access process. However, we rely on friendly application processes to prevent this problem. The message passing primitives are implemented as a library of procedure calls which are shown in appendix A.

Timer Support: The Receive primitive provides access to the timer support mechanism in the kernel. When a Receive primitive is executed with a time-out value specified and there is no message waiting to be received, an alarm is set indicating the time when the Receive expires. The alarm value for the process is inserted into an ordered list of processes awaiting time-out. If the time-out period expires, a signal is sent from the timer to the process to indicate the event. The signal unblocks the waiting process, indicating a signal from the timer has been received. If the process receives a message prior to the time-out period expiring the alarm for the process is deactivated and the message is passed to the waiting process.

Performance: The process kernel is installed as a resident portion of the operating system on a workstation. As such, it must be both compact in size and efficient in operation. The hardware configuration used to measure the performance of the process kernel consisted of an IBM PC/AT microcomputer with a clock speed of 8MHz and an Intel 80286 microprocessor. The workstation had 512K bytes of memory and was running the PC/DOS version 3.3 operating system.

The memory requirements to run the kernel are quite small, it adds an additional 8K bytes to the resident portion of the operating system. Each user process requires a process descriptor to maintain state information. The size of these descriptors is reduced to 40 bytes by storing some of the register state information for the user process on the user process stack prior to performing a kernel request.

Performance measurements on the kernel were obtained by executing 20000 send-receive-reply sequences with a message size of 10 bytes. The overhead time associated with the loop code was subtracted from the total time. This resulted in 1709 send-receive-reply sequences being performed per second. Hence the time for one send-receive-reply is 0.58 milliseconds. A similar test determined the context switching time was 4334 context switches per second. The time for one context switch was 0.23 milliseconds.

A subroutine call and return sequence on the Intel 80286 requires 2 microseconds. The send-receive-reply sequence requires 580 microseconds. This implies the time required for message passing is approximately 300 times the normal subroutine invocation sequence. The use of processes must be carefully controlled to ensure that messages are not used as substitutes for subroutine invocations.

PROCESS-BASED COMMUNICATION SYSTEM IMPLEMENTATION

Introduction

In this chapter we concentrate on the implementation of a process-based layered communication system for a local area network. The distinguishing features of the communication system are efficient and reliable network communications, modular structure and the ability to support heterogeneous network environments. The communication system may be described by using the Internet model discussed in chapter 2. Each of the network access, internet, and host-host layers are implemented as a group of communicating processes. The network access layer contains all aspects of communications related to the exchange of data between workstations on the same network. The internet layer provides the routing and queue management services for communication between local area networks. The host-host transport service provides reliable communications for arbitrary sized data objects. This layer is responsible for the establishment and breaking of connections between processes. Included in this layer is the disassembly and assembly of data objects which exceed the capacity of the internet communication layer.

Addressing

Two types of addressing are used in the communication system. We use the term extended LAN [Back88] to refer to a local area network consisting of multiple component LAN's connected by gateway stations.

A flat name structure is one in which each entity has a name that is unique throughout the entire communication network. This type of structure is appropriate for entities on a single component network. This structure falters when communications may involve a variety of systems of different types from different vendors as each system tends to have different naming conventions.

A hierarchical name structure is one in which the names are layered, such that an entity has a name which is only required to be unique over one component of the network. Each name forms a three-tuple consisting of the fields domain, node, and port. The domain field identifies the local area network in the extended LAN to which we wish to send a message. The node field identifies a particular workstation within the specified domain. The port field identifies a service access point connected to an individual process on the workstation. An advantage of hierarchical structure is that it is easier to add names to the system since an entity must only be unique within a local system. Routing messages between local area networks is also simplified since it is easy to identify which domain contains the entity.

In an extended LAN, routing is required to direct messages between component systems. Static routing is a type of routing where each workstation maintains a predefined table of predefined routes which span the topology of the extended LAN. The information in the route table is used to direct packets to the proper domain.

Our communication system currently uses static routing. However, since the routes on any station may be modified locally at the station or remotely from another station, static routing does not appear to restrict our system. If a failure occurs, the network may be reconfigured quite rapidly.

Implementation of Layers

Each of the three layers in our communication system is composed of sets of processes which communicate by passing messages using the inter-process communication primitives discussed in chapter 3. The process structuring of the communication system is illustrated in Figure 5. Each of the application processes communicates through a port process or service access point to access the network. The port process encapsulates the message from the application with control information and sends it to a queue administrator process. The queue administrator has similar functions to those of the administrator defined by Gentleman [Gent79]. By definition, the administrator process must never block on any of the worker or client processes it services. To ensure the queue administrator does not block, Send primitives are only issued by port processes, thus the queue administrator process is always available to handle incoming requests.

The queue administrator process is responsible for the management of buffers for an arbitrary number of processes called ports. Ports are divided into two types: driver ports and application ports. A driver port implements an interface between the queue administrator and the network. The application port is an interface between the queue administrator and the application process using the communication system. Messages are forwarded by the queue administrator to the driver ports which transfer the messages to the network medium. The driver port implements the medium access interface, described in the IEEE 802.2 LAN standard, and provides the connection to the physical media. The next three sections describe the implementation details of the host-host, internet and the network access layers of the system.

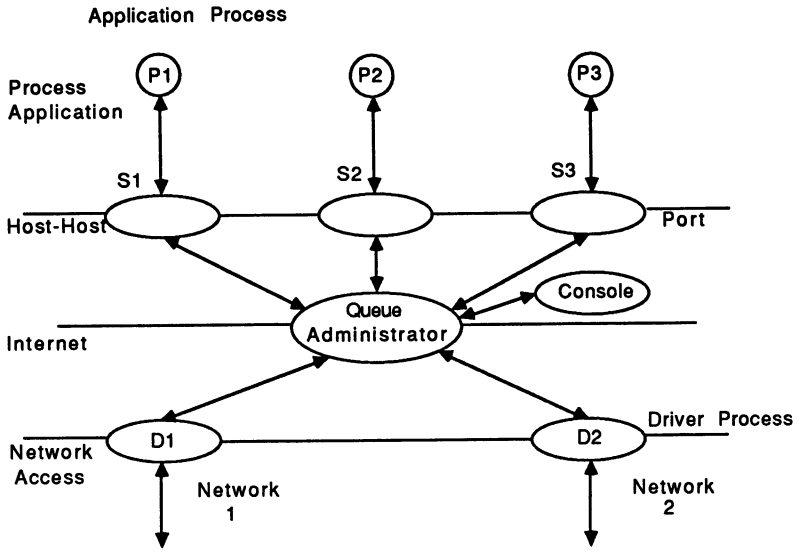

Figure 5: Communication System

The Host-Host Transport Layer

The purpose of the host-host transport layer is to ensure reliable information exchange between different nodes reachable via the underlying network. In a LAN-based communication system the internet layer usually implements a form of connectionless service. Hence, error detection, recovery and flow control must be provided by the host-host transport layer. We use a transport protocol similar to the "light weight" connections described by Cheriton [Cheri84].

The transport process must implement a protocol to describe the transmission language between machines. The nature of the protocol required is dependent on the underlying communications system. If the underlying communication system was slow and unreliable a robust protocol which could easily recover from communication errors would be required. In the case of a LAN-based communication system the underlying network displays the characteristics of a high data rate and a low error rate. In this type of environment a simpler protocol may be used.

The simplest of these protocols, the stop and wait protocol, involves the transmission of an acknowledgement (ACK) packet for every data packet. Zwaenenpoel [Zwae85] showed that the cost of using a stop and wait protocol on a local area network is substantial because of the overhead in generating and receiving ACK packets. He suggested using a blast protocol with retransmission. In the blast protocol, the number of ACK packets is reduced to one at the end of a successful transmission. If an error is detected, retransmission will be from the first packet.

The use of a blast protocol must be restricted to reduce the probability of flooding the receiver. For example, in a file-server based network it is common to have multiple stations using the server simultaneously. If a blast protocol is used by all stations it would place a heavy demand on buffer space in the server. Incoming data packets which arrive after the server has exhausted the buffer supply will be lost or ignored. Retransmission of these lost or ignored packets may cause the server to flood again. This type of situation could result in a very low effective throughput rate for the network.

The group blast protocol is a modified version of the blast protocol. The group blast protocol includes a flow control mechanism to reduce the probability of flooding with large data transfers. The group blast protocol divides large data transfers into a number of groups. Each group is transmitted using the blast protocol. Between the transmission of successive groups the sender and receiver synchronize to ensure the group was received correctly and that the receiver is prepared to receive the next group.

In our implementation of the host-host layer, data transmission is divided into two types: short messages and long messages. Short messages are sent using a blast protocol. The receiver is assumed to have the capacity to receive small packets. Currently small packets contain up to a maximum of 256 bytes. Each small packet is acknowledged by the server upon reception. Lost small packets are handled by a time-out and retransmission mechanism. The protocol for the transmission of a single small packet is shown in Figure 6. The definition of transport layer packets may be found in Appendix C.

Long messages are sent using a synchronized group blast protocol. Prior to transmitting a large message the workstation must synchronize with the server by sending a service request packet (SRQ). Message transfer is integrated with the service request packet to reduce the number of packet exchanges. If the server has buffers available it returns a clear to send (CTS) packet to the workstation indicating it is willing to receive the remainder of the group. Each group may hold a maximum of 4096 bytes of data. The final packet of the group is marked to indicate the end of the transmission. If the server received the group correctly, an acknowledgement packet is returned to the workstation. The protocol for the transmission of a single large message transmission is shown in Figure 7.

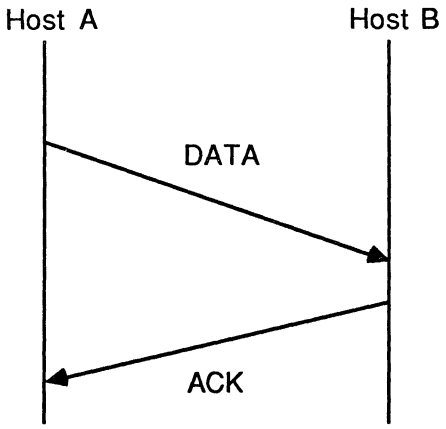

Figure 6: Single Packet Transfer Protocol

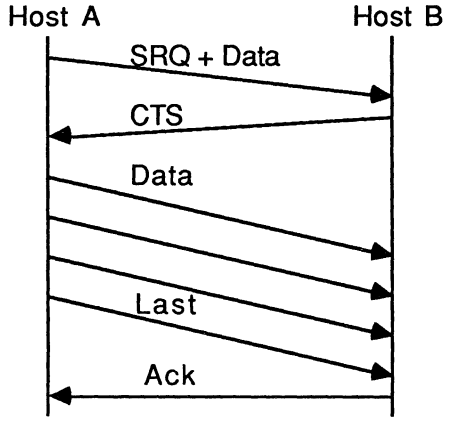

Figure 7: Group Packet Transfer Protocol

Internet Layer

The purpose of the internet layer is to route packets between networks on the extended LAN. We use static routing implemented in the internet layer to direct packets to the proper domain. Routing tables exist at each workstation which specify the driver port and node to use to get the packet one step closer to the destination. The packet is sent to the driver port specified in the routing table. The packet is then forwarded by the port to the gateway node specified in the routing table entry. A gateway node is a workstation which has connections to two or more component networks. The gateway node will determine where to send the packet next. If the packet destination is on the local component LAN the routing table entry has NULL in the node field. This will result in the packet being sent directly to the destination. Each routing table maintains information on how to direct a packet to any node on the extended LAN. The implementation assumes that the extended LAN is acyclic. There are no procedures to remove packets from the extended network after a maximum number of hops between component LAN's.

Figure 8 shows an extended LAN with two domains labeled D=1 and D=2. The domains or networks are represented in Figure 8 by circles circumscribed with squares. The circles labelled AP, DP and Sw are the application ports, the driver ports and the queue administrator respectively. The following example shows how the application on machine A on domain D=1 would communicate with application port P=4 on machine D on domain D=2.

Routing table entries are required in machines A, C and D to enable the communications to occur. Machine A would require a routing entry which specifies that to reach domain D=2 first send the data unit to driver port P=2 then forward it to node N=1. Machine D would require a routing entry for the reverse communications with machine A. It would specify that to reach domain D=1 first send the data unit to port P=2 then forward it to node N=3 on machine C. Machine C is a gateway machine since it exists on more than one domain. On domain D=1, machine C is known as node N=1. On domain D=2, machine C is known as node N=3. Machine C requires two routing table entries. The first entry specifies that to reach nodes on domain D=1, data units should by sent to driver port P=2. Since driver port P=2 is on domain D=1 no additional forwarding of the data unit is required. The second routing entry specifies that to reach nodes on domain D=2, data units should be sent to port P=4. Once again no further forwarding of the data unit is required.

The data unit transfer is performed between machine A and machine C in the following fashion:

1. The application on machine A sends a message to application port P=1 indicating the message is to be sent to domain 2, node 4, and port 4.

2. The application port formulates the message into the format required for a queue administrator data unit. The packet is sent to the queue administrator.

3. By using the information in the routing table the queue administrator determines the data unit should be sent to port P=2 and then forwarded to node N=1. The data unit is transferred to the driver port P=2.

4. The data unit is sent over the network to the gateway node N=1. When received the driver port relays the data unit to the queue administrator on machine C.

5. The routing tables are used to determine the data unit should be sent to driver port P=4 which is on the destination domain for the data unit. The driver port sends the data unit to the destination node N=4 on machine D.

6. The driver port on machine D receives the data unit and transfers it to the queue administrator. The queue administrator determines that the data unit is on the correct domain and port before it sends the data unit to the application port P=4.

Data units are routed first to the proper domain and then to the correct machine on that domain. The address of the final destination for the data unit is maintained in the destination node field during the routing process. The port field is used to select the correct port on the destination machine.

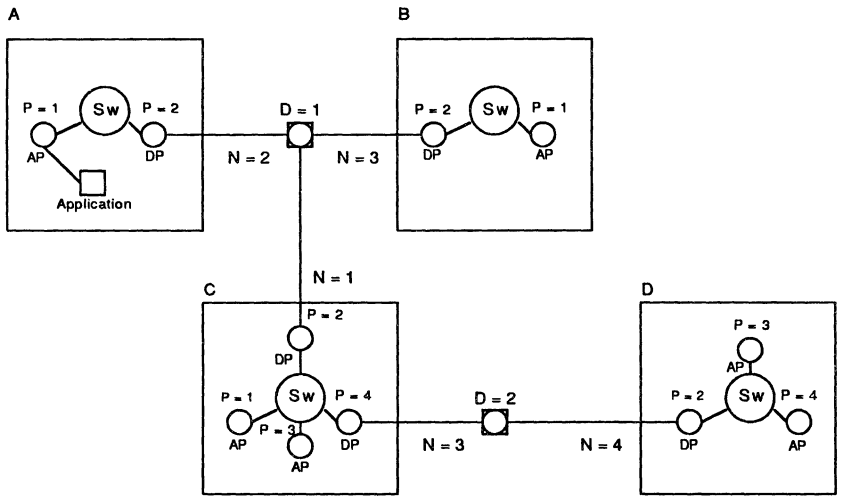

Figure 8: Network Addressing

Network Access Layer

The purpose of the network access layer is to provide connectionless service between nodes on a single network. The network access layer is implemented as a driver port process which is connected to the queue administrator process. Sample code for a driver port is provided in Appendix E. The driver port is responsible for transmitting and receiving packets on the local area network. Packets received from the queue administrator process are transmitted to their next destination. Packets received from the network are sent to the queue administrator process for distribution. The packet format for network packets is shown in Figure 9.

The driver port prepares packets for transmission by using the network header information to fill in the necessary data link fields of the packet. When packets are received, the driver port performs any necessary translation to put the packet into the standard internal format shown in Figure 9.

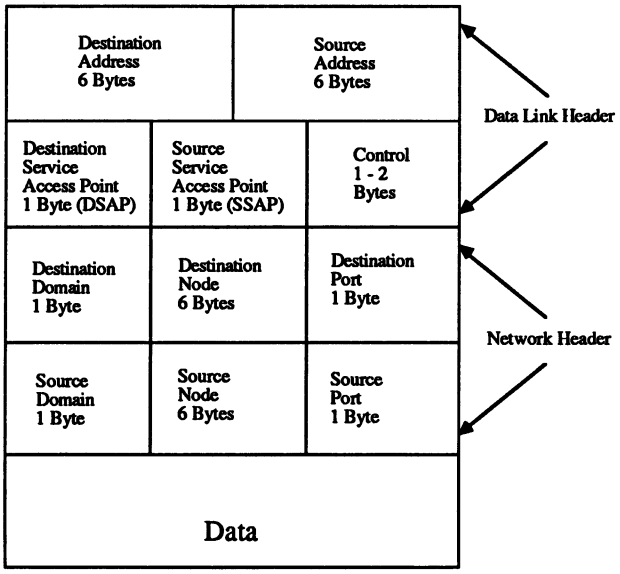

Figure 9: Packet Format

The network adapters are configured to respond to both the individual node address and the network broadcast address. Thus incoming broadcast packets are received in addition to packets addressed to the individual node. Broadcast packets follow the packet format shown in Figure 9. The broadcast packet is sent to the queue administrator on each machine for delivery to the application port specified in the broadcast packet. Packets with the destination address set to be the broadcast address are transmitted by the driver port to all stations on the local domain.

Most local area network adapter cards support some form of remote program load facility. This facility allows workstations to be automatically loaded from the network when they are started. The packets generated by the remote program load facility usually differ in format from the standard packet format used in the communication system. The format of the packets cannot be easily modified because of the ROM implementation of the procedures. A transformation by the driver port changes these packets into the standard format and sets the address field to be a prescribed port which services remote boot requests.

Currently two driver ports have been implemented for the communication system. Driver ports for the IBM PC Cluster and the IBM PC Network Baseband adapter have been implemented. The widely varying data link interfaces presented by these two adapters did not present any serious complications. Other network adapters such as Token Ring and Ethernet have been examined and driver ports for these media appear feasible. This leads us to believe that driver ports for other network adapters can be added easily to the system.

Buffer Management

The performance of a layered communication system is directly affected by the number of message copy operations required between the layers. This fact is amplified in local area networks since the transmission speed of the network usually exceeds the speed of the processor in the workstation accessing the network. Buffer management in our communication system is controlled by the queue administrator.

The queue administrator process is responsible for the management of buffers for an arbitrary number of port processes. Ports are divided into two types: driver ports and application ports. A driver port implements the media access interface to a network. The application port implements the interface between the queue administrator process and the application process using the communication system.

A port process is connected to the queue administrator dynamically by sending a SET_PORT request to the known address for the Console port. The Console port is a utility port which can establish and remove ports from the local queue administrator process. In addition the Console port controls the setting and modification of the static routes in the routing tables on the workstation. Since the Console port is attached to the queue administrator, the functions provided by the Console port may also be accessed remotely from any workstation on the extended network. A port descriptor is established which describes the current state of the port. Each port process connected to the queue administrator process is assigned a pair of packet queues which are initially empty. These packet queues operate in a first-in-first-out (FIFO) manner and represent the transmission and reception queues for the port. Once the port is connected to the queue administrator process it can receive packets from other ports on the same machine.

A port obtains packets of data from the queue administrator process by sending a message requesting a packet from the receive queue for the port. If a packet exists in the receive queue the packet field in the message is set to the address of the packet and returned with a reply. Otherwise the message is returned with the packet field set to NULL. Figure 10 illustrates the packet data structure returned from the queue administrator. The complete definition of the packet data structure may be found in Appendix B.

When the packet has been emptied it is returned to the queue administrator. If another packet is waiting to be received it is returned automatically. By passing pointers we avoid extra message passing while processing a sequence of packets. When there are no further packets to receive, the switch message is returned with the packet field set to NULL. The next time a packet is received for this port the queue administrator signals the port that there is a packet available for processing. This signalling prevents the port from having to poll to determine the state of the receive queue.

If the port is required to send data in addition to receiving data, buffer space for this operation must be allocated by the port process. The number of packets allocated determines the transmission window size for the port. The allocated packets are given to the queue administrator and placed in the transmission queue for that port. Figure 11 shows a sample configuration. The queue administrator process has three ports connected. Each port has one available packet remaining in their respective transmission queues and two packets in flight. Transmission packets are managed by the queue administrator and returned to the port when packets are required for data transmission. When a packet is transmitted to the queue administrator the addressing information contained in the buffer within the packet is used to determine the proper driver port to use to reach the destination port. The packet is placed in the receive queue for the proper driver port. The driver port transmits the packet on to the network and then returns the packet to the queue administrator process so it can be returned to the original owner port of the packet. The returned packet is placed on the end of the transmission queue of the originating port. Sample code for the queue administrator is provided in Appendix F.

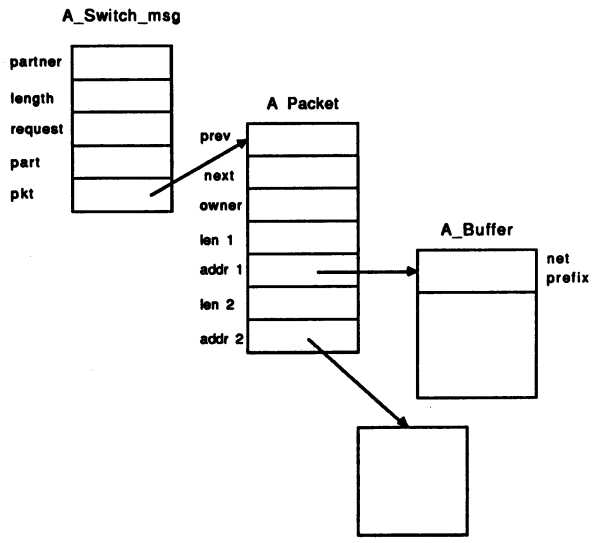

Figure 10: Packet Data Structure

Performance

The communication system was designed to provide good system performance, not just good kernel performance. Good system performance requires that the higher level operations performed by applications must map efficiently onto the primitives provided by the communications system. The performance of the communication system is measured in terms of the attainable data throughput rate. In this section we measure the performance of our communications system at both the internet and the host-host layers.

The hardware configuration used consisted of two IBM PC/AT microcomputers. Both workstations had clock speeds of 8MHz and used the Intel 80286 microprocessor. Each workstation was equipped with 512K bytes of memory and an IBM PC Network Baseband adapter. The network adapter had a data transmission speed of 2.0Mb and used the CSMA/CD access protocol. The network was lightly loaded when the testing was performed.

The performance of the internet layer was measured using an application port designed to transmit or receive data through the internet layer. The program is designed to measure the time required to transmit the specified number of packets on the network. The packet size transmitted may be set to either small or large packets. The transmission of 2000 packets of size 120 bytes required 9 seconds. This translates to a rate of 222 packets/sec and a transmission throughput of 0.21Mb/sec. The transmission of 2000 packets of size 1220 bytes required 17 seconds. This translates to a rate of 117 packets/sec and a transmission throughput of 1.2Mb/sec which represent a 60 percent utilization of the transmission medium.

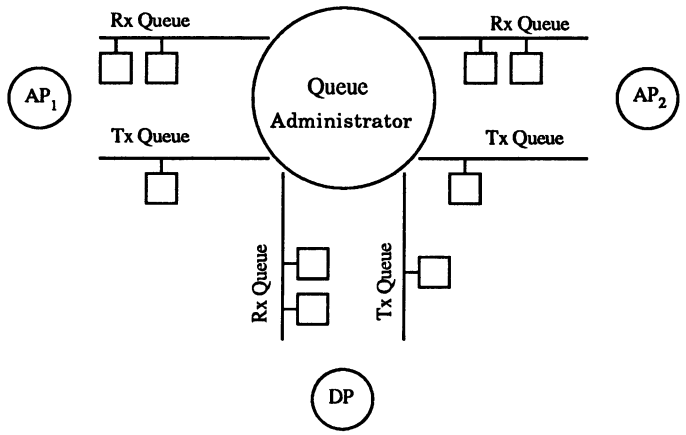

Figure 11: Buffer Management

The throughput performance of the communication system is directly related to the size and number of packets transmitted. By experimentation, we have determined the optimal packet size to be 1024 bytes. At this size we achieved good performances and the demand on buffer space was not excessive. The performance of the host-host transport layer was measured by a measurement program designed to transmit data through the communication system to an application port on the destination node. The test involved the transmission of 500 messages of 4000 bytes each. Each message transmitted would require a group of four large packets be sent with acknowledgement by the host-host transport layer. The measured speed was 0.591Mb/sec.

General Comments

The communication system provides a general purpose mechanism for application processes to use the network. Application processes can use either the reliable transport service provided by the host-host layer or they can interface directly to the internet layer.

Heterogeneous network environments can be accommodated by having a gateway machine with two driver ports. Then by setting up the appropriate routing table entries, a node on the network can communicate with any other node on the extended network.

The processing overhead related to buffer copying has been minimized by passing packets by reference between the layers. Synchronization problems are avoided by using blocking send and receive primitives.

AN APPLICATION OF THE COMMUNICATION SYSTEM

Introduction

Even with powerful independent workstations there is a need for connectivity to high speed local area networks. These networks are required to allow the sharing of information and expensive peripherals. In this chapter we concentrate on the implementation of the Waterloo JANET local area network system which uses the communication system described in chapter 4.

Waterloo JANET

Waterloo JANET was originally designed to provide shared access to printer and disk resources on a central server. The configuration in Figure 12 shows a typical system. Waterloo JANET was developed to provide efficient low cost computing resources for instructional purposes. Waterloo JANET was first installed in 1981 and there are currently over 25 such LANs in various academic areas on the campus of the University of Waterloo.

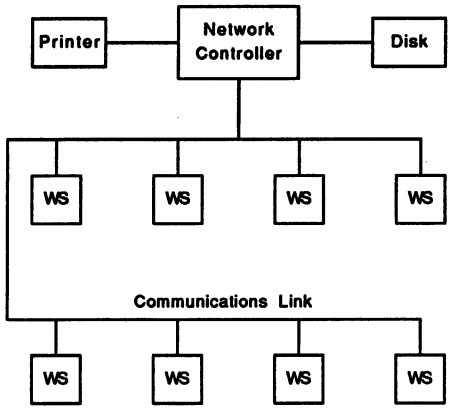

Figure 12: Waterloo JANET Configuration

The original Waterloo JANET implementation used the IEEE-488 bus [Fish80] for a physical layer. The structure of the communication system in this implementation followed an integrated approach and co-existed with the single thread PC/DOS operating system. Workstations were either IBM Personal

Computers or PC Compatibles. Waterloo JANET was designed with the philosophy that the amount of extra knowledge required to use the network should be minimal.

When a user first encounters a microcomputer workstation connected to Waterloo JANET, he sees a logo and a line requesting a userid. When the userid is typed, a password is requested. Once this authorization and authentication procedure is complete, the Waterloo JANET workstation appears to the user as a normal stand-alone microcomputer with six fixed-capacity network disks. These network disks behave as if they were diskette drives; for example, on an IBM PC the network disks are labelled A,B,C,D,E and F and each can store files in a tree-structured file system. Each network disk can be a different size ranging from 20Kbytes to 2.0Mbytes, the size depends upon some initial allocation. Each network disk is actually a portion of the hard disk which is attached to the disk-server.

Security is extremely important when file space is shared among many users in a computer system designed for educational applications. Access to files is controlled by the userid and password required when a user signs on to the system. Also the amount of sharing that occurs among the different user-groups can be controlled by mechanisms which allow limits to be placed on this type of activity.

Some extra commands have been added to PC/DOS to allow the disk-server and print-server to be manipulated from a workstation. These commands, which are described in the following paragraphs, are: access, pswd, detach, limit, logoff, printit, purge, and qprn.

The access command allows the user to establish a logical connection between one of the drive letters on the workstation and a portion of the hard disk on the server. Access to a specific network disk may also require a password. The former network disk attached to that label is released. The pswd command allows the user to change the password of a network disk. The detach command is the counterpart of the access command, it breaks the mapping between a local drive letter and the server.

The print-server is controlled by a number of commands. Output from the workstation is accumulated or spooled on the server until it is released by the printit command from the workstation. When the file is released for printing it can be directed to one of the printers attached to the print-server. The number of characters of printing which can be accumulated by the print-server for any workstation is limited. The limit command can be used to change this value. Spooled output which is not required can be deleted via the purge command before it is printed. The qprn command allows a user to determine the status of the printer. A summary of the output queue for the printer is reported.

The logoff command disconnects a user from the Waterloo JANET server and from the network disks that were accessed.

Waterloo JANET Implementation

A server is a system that provides services for a previously unknown set of clients. A common example of a server occurs in most local area networks where there is a file server and a number of workstations which access the file server. The Waterloo JANET server may be described in terms of this client/server model. The server is responsible for scheduling the access to the shared resources it controls.

A client is an entity that uses the services provided by servers. The client is designed to communicate with the server over the connecting local area network.

Since each client and server usually reside on a separate processor, the modular approach to the software design requires special attention. The division of function between the client and the server must take into account the remote connection between the entities. Many other research projects have investi-

gated the client/server approach including project Athena at MIT [Balk85]. We use the client/server model to describe our Waterloo JANET implementation. Figure 13 is a block diagram showing the architecture of a Waterloo JANET system using the communication system. An important implementation consideration was that the communication system co-exist with PC/DOS such that existing PC/DOS applications would still operate correctly.

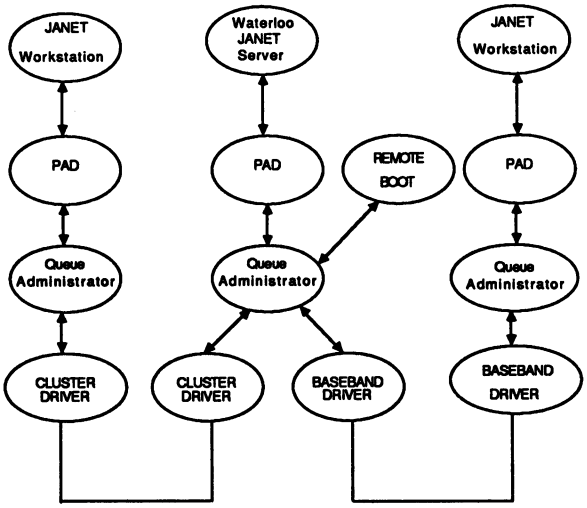

Figure 13: Layering of System

The illustrated configuration has the Waterloo JANET server program installed on a gateway workstation. The gateway station connects a domain of workstations on IBM PC Cluster adapters with a domain of workstations on IBM PC Network Baseband adapters. The Waterloo JANET software is composed of the following three applications:

- the Waterloo JANET Server
- the Waterloo JANET Workstation
- the Remote Boot Port

The following sections describe the functions of the three components of Waterloo JANET.

Waterloo JANET Server

The interface between Waterloo JANET server and the communication system is the host-host transport service labelled PAD in Figure 13 on page 27. This layer provides the reliable communication of arbitrary size application packets required in the Waterloo JANET application.

In addition to servicing requests for data transfer and printer access, the Waterloo JANET server provides authentication, authorization and network maintenance services. The format of requests for service by the Waterloo JANET server are shown in Appendix D. Service requests received from the workstations are processed by the server and the results are returned to the requesting workstation.

The structure of the Waterloo JANET server requires that messages be copied to and from the transport layer of the communication system. To minimize the overhead associated with the copy we have added a simple presentation layer data compression algorithm. A library routine compresses the data while it is being copied between the buffers. The compression routine eliminates sequences of three or more identical characters. The sequences are replaced in the buffer by an escape character, a length, and the repeated character. Since the chosen escape character may appear in the buffer, occurrences of the escape character are replaced by an escape character, a length and the escape character. The compression routine has been designed to remove the possibility of the compressed buffer being larger than the original. If we are asked to compress a buffer where the resulting buffer is larger than the original, the compress routine aborts the attempted compression and performs the function of a normal copy routine. Measurements of the operation of the compression routine indicate we achieve a 15 to 20 percent reduction on average in the packet size after compression.

Waterloo JANET Workstation

The Waterloo JANET workstation process, along with the communication system, is loaded from diskette at the workstation. The workstation process replaces the diskette and printer interrupt routines and connects to the transport layer of the communication system. To determine the location of the Waterloo JANET server program, a broadcast packet is issued to all stations on the network. Any station which has a Waterloo JANET server running will respond and provide the workstation with an address to be used for any future communications.

Remote Boot Port

The remote boot port services boot requests from workstations which do not have diskette drives. These stations send a broadcast message on the network to locate a port from which to load. The remote boot port transfers a copy of the Waterloo JANET workstation program to the requesting stations. When the Waterloo JANET workstation code is received, it is installed as a replacement for the diskette device driver. Any future diskette requests from the workstation are directed to the Waterloo JANET server for processing. The function of the workstation is now similar to the locally loaded workstation previously described.

The remote boot port is an application port which communicates directly with the queue administrator process. The remote boot protocol reliability is provided by the network adapter card.

Performance of System

The Waterloo JANET application was designed to provide disk access for the workstations which is comparable in performance to local diskette drives. The performance of the system is measured by timing a file copy from the network and from diskette. The file copied in each case was 107698 bytes in length.

The hardware configuration used consisted of two IBM PC/AT microcomputers. Both workstations had clock speeds of 8MHz and used the Intel 80286 microprocessor. Each workstation was equipped with 512K bytes of memory and an IBM PC Network Baseband adapter. The network adapter had a data transmission speed of 2.0Mb and used the CSMA/CD access protocol. The network was lightly loaded when the testing was performed.

The first copy from the network required 3.8 seconds, subsequent copies from the network required an average of 2.5 seconds. The first copy was longer because the file had to be loaded from the disk on the server. The second and subsequent copy operations read the file directly from the disk cache on the server. The average time required for the copy from diskette was 8.7 seconds. We see that the network performance meets and exceeds the expressed goal for the Waterloo JANET system. For comparison, we measured the time required to copy the test file from the hard disk on the PC/AT. The copy from the hard disk required 1.1 seconds on average. The speed of the copy from the local hard disk is a lower bound on the performance of the network server since the network server has the same type of hard disk.

Other measurements of the Waterloo JANET system were performed on a network of 16 IBM PS/2 model 25 workstations and IBM PS/2 model 60 server. The simultaneous load of a 160K program file on the 16 workstation took 8 seconds. These measurements indicate the communication system performs well under load. Our measurements indicate the speed of the hard disk on the server is a limiting factor on network performance before network transmission capacity is exceeded.

CONCLUSIONS

Summary

In this paper, we have investigated the design and implementation of a process-based layered communication system for a local area network. Special purpose "light weight" protocols designed to take advantage of the high data and low error rates of the local area network are used in place of standard general purpose data communication protocols.

Many models have been proposed for the structure of communication systems. The widely accepted OSI model proposes the communication function be divided into seven layers. Each layer performs a portion of the entire communication operation. A subset of the OSI model designed to deal with communication systems for local area networks has been proposed by the IEEE 802.2 committee. This three layer model accommodates the special communication requirements of local area networks. The IEEE 802.2 LAN model does not provide protocols for interconnected local area networks. The Internet model suggests a four layer structure where the four layers form a hierarchy and an application may use functions of any lower layer. Our communication system uses features from both the hierarchical and IEEE 802.2 models.

Each of the layers of the communication system is implemented as a set of co-operating sequential processes. A real-time message passing kernel was implemented to provide the process and message passing environment required by the communication system.

The communication system was designed to provide applications with efficient communications over the network.

The Waterloo JANET local area network application, which uses the communication system, has been in use at the University of Waterloo since September 1987. Undergraduate student computing requirements for many courses have been provided by the system. In addition, there are five other sites outside the University of Waterloo successfully using the system.

Conclusions

Measurements of the system indicate that it is possible to design and implement a local area network according to the design principles exemplified by the layered and hierarchical approaches. Further modularity can be achieved through the use of co-operating sequential processes implemented under an existing operating system supplemented by a real-time kernel.

Future Investigation

The current network access layer interfaces to the IEEE 802.2 medium access control layer on the IBM PC Network Baseband adapter. Plans are in place to extend this interface to the 802.2 logical link control level, a common interface used by many existing network application packages. Hence, existing packages not specifically designed for our communication system will be able to access the network through the standard interface.

Other network applications programs, such as distributed data-bases and terminal servers, are being developed at the time of writing. These applications connect to the communication system as applications ports. The network requirements of these applications may well be different from the requirements for Waterloo JANET. Additional monitoring of the communication system will be required to determine if minor modifications are required for these applications.

Finally, the protocols and packet formats used in the internet and host-host layers of the communication system should be changed to proper subsets of the Internet standard protocols. The growing number of networks using the Internet protocols suggests that gateway access to these networks will soon be desired from local area networks. By using subsets of the standard protocols the complexity of the gateway program will be reduced.

Appendix A
DOS/KX: MULTI PROCESS KERNEL OPERATIONS

```
/*
        kernel
*/
#define KERNEL_SIGNATURE        "Kernel v1.3"

/*
    Priorities
*/
typedef unsigned a_priority;
#define MAX_PRIORITY            ((a_priority) 65535)
#define NORMAL_PRIORITY         ((a_priority) 32768)
#define UNIT_PRIORITY           ((a_priority) 4096)

/*
    Names
*/
typedef unsigned a_pid;
#define ME                      ((a_pid) 32767) /* Also used for no-name. */
#define NET_SWITCH              1               /* Reserved name for Switch. */
#define NUT_KPRN                2               /* Reserved name for kprintf.*/
#define NUT_FIRST_FREE          10              /* First non-reserved name. */

/*
    Messages
*/
typedef struct a_message {
    a_pid       partner;
    unsigned    length;         /* # of bytes of data that follow */
} a_message;
#define msgdata( m )            (((char *) &(m)) + sizeof(a_message))
#define msgsize( m )            (sizeof(m)-sizeof(a_message))
```

```
/*
    Signals
*/
#define SIG_STATS               0x8000          /* Sent by kstat(). */
#define SIG_TIMER               1
#define SEC_1000                18206
#define SEC_1                   (SEC_1000/1000)

/*
    Interrupt #'s
*/
#define IRQ_TIMER               1
#define IRQ_BASEBAND            2               /* and 3 */
#define IRQ_CLUSTER             4               /* and 5, 6, 7 */

extern unsigned (far *kcall)();
#define K_DISPATCH              0
#define K_CATCH                 1
#define K_IGNORE                2
#define K_SEND                  3
#define K_RECEIVE               4
#define K_REPLY                 5
#define K_SIGNAL                6
#define K_SPAWN                 7
#define K_KILL                  8
#define K_STOP                  9
#define K_ME                    10
#define K_MAXPD                 11

extern char far * GetStk( unsigned ); /* Small code, no free. */
extern char * StackAlloc( unsigned ); /* Large code, alloc/free. */
extern void StackFree( char * );

extern a_pid Spawn( char far *, a_priority, a_pid, void (far *)() );
#define Kill( id )              (*kcall)( K_KILL, (a_pid) id )

#define Send( msg )             (*kcall)( K_SEND, (a_message far *) msg )
#define Receive( msg, delay )   (*kcall)( K_RECEIVE, (a_message far *) msg, (unsigned)
delay )
#define Reply( msg )            (*kcall)( K_REPLY, (a_message far *) msg )
#define Signal( id, signal )    (*kcall)( K_SIGNAL, (a_pid) id, (unsigned) signal )

#define Catch( intreq )         (*kcall)( K_CATCH, (unsigned) intreq )
#define Ignore( intreq )        (*kcall)( K_IGNORE, (unsigned) intreq )

#define Stop()                  (*kcall)( K_STOP )
#define Me()                    (*kcall)( K_ME )
#define Maxpd()                 (*kcall)( K_MAXPD )

extern unsigned long (far *ktime)( void );
#define KTime()                 (*ktime)()
```

Appendix B
NETWORK LAYER MESSAGE DEFINITIONS

```
#define SWITCH_SIGNATURE        "Switch v1.1"
#define SWITCH_TS               1287            /* (0) Dec 87 */
/*
   Priorities.
*/
#define SWITCH_PRIORITY         (MAX_PRIORITY-UNIT_PRIORITY)
#define PORT_PRIORITY           (SWITCH_PRIORITY-UNIT_PRIORITY)
/*
   Reserved ports.
*/
#define NET_DROP_PORT           0       /* Just assumed in code. */
#define NET_BCAST_PORT          1       /* All domain broadcast. */
#define NET_CONSOLE_PORT        2       /* Name & remote routes. */
#define NET_TEST_PORT           3       /* Receive wiring test messages. */
#define NET_SCREEN_VIEW         4       /* Receive screen broadcasts. */
#define NET_JANET               5       /* Reserved for JANET use. */
#define NET_FIRST_FREE          10      /* First non-reserved port number. */

/*
   an_address
*/
```

```
typedef unsigned char
   a_domain_id;

typedef struct {
   unsigned char a[6];
 } a_node_id;

typedef unsigned char
   a_port_id;

typedef struct an_address {
   a_domain_id domain;
   a_node_id node;
   a_port_id port;
 } an_address;

static a_node_id  NULL_NODE_ID = {0};           /* a constant ! */
static an_address NULL_ADDRESS = {0,0,0};       /* a constant ! */
static a_node_id  RPL_NODE_ID = { 0xC0, 0x00, 0x40, 0x00, 0x00, 0x00 };

/*
   a_net_prefix

     The application port must set dst to the destination.  The source domain
    and node is filled in if the src domain is 0.  The source port is filled
    in if the source port is 0.

     Switch sets the src and lan.dst field based on the switch route tables.

*/
#define ROUTE_MAX_LEN                   18      /* bytes */
#define RouteBit( pfxp )                ( pfxp->lan.src.a[0] & 0x80 )
#define RouteBitSet( pfxp )             ( pfxp->lan.src.a[0] |= 0x80 )
#define RouteLen( pfxp )                ( pfxp->lan.route[0] & 0x1f )
#define RouteLenSet( pfxp, len )        ( pfxp->lan.route[0] |= (len & 0x1f) )

typedef struct a_net_prefix {
   struct LAN802 {              /* 802.2 lan */
      a_node_id dst;
      a_node_id src;
      unsigned int length;
    } lan;
   an_address dst;
   an_address src;
} a_net_prefix;


/*
   a_buffer
*/
#define SWITCH_DATA_MIN (578 - sizeof(a_net_prefix) + sizeof(struct LAN802) - 1)

typedef struct a_buffer {
```

```
    a_net_prefix pfx;
    unsigned char data[SWITCH_DATA_MIN];
  } a_buffer;


/*
    a_packet

       A packet's first buffer (len1, addr1) must start with a_net_prefix.
    Len1 includes sizeof(a_net_prefix).  The optional second buffer (len2, addr2)
    must contain only data.  Driver ports receive data into a single buffer
    (len1, addr1) which starts with a_net_prefix.
*/
typedef struct a_packet {
    struct a_packet  far * prev; /* switch internal */
    struct a_packet  far * next; /* switch internal */
    void near * owner;           /* switch internal */
    unsigned int len1;           /* len of data in addr1 (incl pfx in first) */
    a_buffer far * addr1;        /* prefix & data */
    unsigned int len2;           /* len of data in addr2 */
    char far * addr2;            /* data */
  } a_packet;


/*
    a_switch_msg
*/
#define ALLOC      0x01
#define TX_QUEUE   0x02         /* RX_QUEUE if not TX_QUEUE */

typedef struct a_switch_msg {
  a_pid partner;
  unsigned int length;
  unsigned char request;
  a_port_id port;
  struct a_packet far * pkt;
} a_switch_msg;

#define TX_AVAIL        0x4000          /* 0x8000 is SIG_STATS */
#define RX_AVAIL        0x2000

/*
    alloc.c
*/
extern int AllocBuffers( a_port_id, unsigned );            /* port, n */
extern int AllocBuffersS( a_port_id, unsigned, unsigned ); /* port, n, size */

extern a_port_id SetPort( a_port_id, char * );
extern a_port_id SetDriv( char *, a_domain_id, a_node_id, unsigned );
extern int ResetPort( a_port_id );
extern a_domain_id SwMaxDomain( void );
extern unsigned SwPktLen( a_domain_id );
extern a_port_id SwFindPort( char * );
```

Appendix C
TRANSPORT LAYER MESSAGE DEFINITIONS

```
typedef struct a_pad_prefix {
   a_net_prefix  net;
   unsigned ts;
   enum PAD_ID { PAD_RTS, PAD_CTS, PAD_CONT, PAD_EOT } id;
   unsigned      bundle_serial;
   unsigned char packet_serial;
 } a_pad_prefix;

typedef struct a_pad_msg {
    a_message h;
    enum { PAD_SEND, PAD_RECV, PAD_NONE } request;
    a_pad_prefix * pfxp;
    unsigned pfxl;
    char far * b1;
    unsigned len1;
    char far * b2;
    unsigned len2;
} a_pad_msg;
```

Appendix D
WATERLOO JANET MESSAGE DEFINITIONS

```
typedef struct a_janet_prefix {
    int enum  Handler {
        USERINFO,          /* User defined packets      */
        DISKIO,            /* Disk i/o request          */
        PRINTER,           /* Spool request for spooler */
        ACCESS,            /* Access minidisk           */
        BOOT,              /* Boot workstation occurred */
        TIMER,             /* Workstation wants TOD     */
        CHPSWD,            /* Password processor        */
        LOGOFF,            /* Workstation logoff        */
        LOGON,             /* Workstation logon         */
        SPEEDCHK,          /* Measure speed of network  */
        DIRREQ             /* Directory manager req     */
        } id;
  unsigned char data[ 4096 ];
 } a_janet_prefix;
```

Appendix E
NETWORK DRIVER PORT

```
unsigned  signal;
a_request txmsg;
int       txactive;
a_request rxmsg;
int       rxactive;

initialize_port();
txmsg.partner = NET_SWITCH;
txmsg.length = msgsize( txmsg );
txmsg.request = TX_QUEUE|ALLOC;
txmsg.pkt = NULL;
txactive = FALSE;
rxmsg.partner = NET_SWITCH;
rxmsg.length = msgsize( rxmsg );
rxmsg.request = ALLOC;
rxmsg.pkt = NULL;
rxactive = FALSE;
catch();
while( 1 ) {
   signal = Receive( NULL, 0 );
   if( signal & NEW_TX )
      send( &txmsg );
   if( signal & NEW_RX )
      send( &rxmsg );
   if( signal & HW_DONE_TX ) {
      send( &txmsg );
      txactive = FALSE; }
   if( signal & HW_DONE_RX ) {
      send( &rxmsg );
      rxactive = FALSE;  }
   if( !txactive && txmsg.pkt != NULL ) {
      start_tx_io( txmsg.pkt );
      txactive := TRUE; }
   if( !rxactive && rxmsg.pkt != NULL ) {
      start_rx_io( rxmsg.pkt );
      rxactive := TRUE;  }
}
```

The functions start_tx_io and start_rx_io initiate transmission and reception of data from the network hardware, respectively. The hardware signals HW_DONE_TX when a message has been sent and HW_DONE_RX when a message has been received.

Appendix F
QUEUE ADMINISTRATOR

```
struct a_port_descriptor {
   a_packet    *rxqueue;
   a_packet    *txqueue;
   a_domain_id  domain;
   a_process_id driver;
}

code
   a_switch_msg      msg;
   a_port_descriptor *port;
   a_process_id      partner;
   int               empty;
   initialize_switch();
   while ( 1 ) {
      receive( &msg, 0 );
      port = port_of( msg.partner );
      if( msg.request & TX_QUEUE ) {
        if( msg.pkt != NULL ) {
          partner = msg.pkt->owner;
          empty = partner->rxqueue == NULL;
          enqueue( partner->rxqueue, msg.pkt );
          if( empty )
            signal( partner->driver, TX_AVAIL );
          msg.pkt = NULL;
        }
        if( msg.request & ALLOC && port->txqueue != NULL )
          msg.pkt = dequeue( port->txqueue );
      }

      else {
        if( msg.pkt != NULL ) {
          partner = route( msg.pkt );
          empty = partner->txqueue == NULL;
          enqueue( partner->txqueue, msg.pkt );
          if( empty )
            signal( partner->driver, RX_AVAIL );
          msg.pkt = NULL;
          }
        }
        if( msg.request & ALLOC && port->rxqueue != NULL )
          msg.pkt = dequeue( port->.rxqueue );
      }
      reply( &msg );
   }
```

The Port_of(id) function returns a pointer to the port descriptor associated with the process id. The Route(buffer) function returns a pointer to the port descriptor for the port we should send the message in 'buffer' out on.

BIBLIOGRAPHY

Back88 Backes, F, "Transparent Bridges for Interconnection of IEEE 802 LANs", IEEE Network, Vol 2 No. 1, January, 1988, 5-9.

Balk85 Balkovich, E., Lerman, S., Parmelee, R., "Computing in Higher Education: the ATHENA Experience", Communications of the ACM, Vol 28 No. 11, November, 1985, 1214-1224.

Black87 Black, J. P., Cheung, W. H., Lam, E. C., Lau, F. C. M., Manning, E. C., "Shoshin: Developing and Understanding Distributed System Software", Institute of Computer Research 87-04, University of Waterloo, Waterloo, Ontario, Canada.

Cheri79 Cheriton, D. R., Malcolm, M. A., Melen, L. S., Sager, S. R., "Thoth, a Portable Real Time Operating System", Communications of the ACM, Vol 22 No. 2, February, 1979, 105-115.

Cheri83a Cheriton, D. R., "Local Networking and Internetworking in the V-System", *Proc. 8th Data Commun. Symp.*, 9-16, 1983.

Cheri83b Cheriton, D. R., Zwaenepoel, W., "The Distributed V Kernel and its Performance for Diskless Workstation", *Proc. of the 9th ACM Symposium on Operating Systems Principles*, 129-140, 1983.

Cheri84 Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems", IEEE Software, April 1984, 19-42.

Cheu86 Cheung, W. H., "Integrated Versus Layered Approach for Efficient Remote Communication", Master's thesis, Dept. of Computer Science, University of Waterloo, Ontario, Canada, 1986.

Cowa87a Cowan, D. D., Fenton, S., L., Graham, J. W., Stepien, T. M., "Networks for Education at the University of Waterloo", Computer Science Research Report CS-87-49, University of Waterloo, Waterloo, Ontario, Canada.

Cowa87b Cowan, D. D., Stepien, T. M., Veitch, R. G., "A Network Operating System for Interconnected LANS with Heterogeneous Data-Link Layers", Computer Science Research Report CS-87-50, University of Waterloo, Waterloo, Ontario, Canada.

Dici79 Dicicco, V., Sunshine, C. A., Field, J. A., Manning, E. G.,: "Alternatives for the Interconnection of Public Packet Switching Data Networks", *Proc. Sixth Data Commun. Symp.*, 120-125, 1979.

Eise83 Eisenhard, B. T., "Corvus Omninet and its Position in a Hierarchical Network", *IEEE Digest of Papers - Spring'83 Compcon*, 1983.

Fish80 Fisher, E., Jenson, C. W., "PET and the IEEE 488 Bus (GPIB)", Osborne/McGraw-Hill, California, USA, 1980.

Gent81 Gentleman, W. M., "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", Software Practice and Experience, Vol. 11 August 1981, 435-466.

Hail85 Hailpern, B., Heller, A., Hoevel, L. W., Thefaine, Y. J., "ALAN: A (Circuit Switched) Local Area Network", IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA.

Hect86 "HECTOR: A High Speed Heterogeneous University Network. Personal Computer in University Teaching and Training. Project Overview", Project Overview, IBM Deutschland, June 1986.

Hedr87 Hedrick, C. L., "Introduction to the Internet Protocols", Computer Science Facilities Group, Rutgers University, New Jersey, July 1987. 1986, 184-201.

IBM84 IBM Corp, "IBM Personal Computer Hardware Reference Library Technical Reference Options and Adapters Volume 2", IBM Corp, Florida, USA, 1984.

IBM86 IBM Corp, "IBM Personal Computer Hardware Reference Library Technical Reference Token-Ring Network PC Adapter", IBM Corp, Florida, USA, 1986.

IBM87 IBM Corp, "IBM Personal Computer Hardware Reference Library Technical Reference PC Network Baseband Adapter", IBM Corp, Florida, USA, 1987.

IEEE85a IEEE Standard 802.2-1985, "Logical Link Control", IEEE, New York, USA, 1985.

IEEE85b IEEE Standard 802.3-1985, "Carrier Sense Multiple Access with Collision Detection", IEEE, New York, USA, 1985.

IEEE85c IEEE Standard 802.5-1985, "Token-Passing Ring Access Method and Physical Layer Specification", IEEE, New York, USA, 1985.

Kell85 Keller, H., Muller, H. R., "Engineering Aspects for Token-Ring Design", IBM Zurich Research Laboratory, Ruschlikon, Switzerland.

Limb82 Limb, J. O., Flores, C., "Description of Fasnet - A Unidirectional Local Area Communications Network", *Bell System Tech Journal.*, Vol 61, No 7, p1413-1440,1982

Morr85 Morris, J. H., Satyanarayanan, M., Canner, M. H., Howard, J. H., Rosenthal, D. S. H., Smith, F. D., "Andrew: A Distributed Personal Computing Environment", Communications of the ACM, Vol 29 No. 8 March 1986, 184-201.

Pope79 Popescu-Zeletin, R.,: "The Data Access and Transfer Support in a Local Heterogeneous Network (HMINET)", *Proc. Sixth Data Commun. Symp.*, 147-152, 1979.

Post80 Postel, J., "User Datagram Protocol", RFC 768, USC/Information Sciences Institute, August, 1980.

Post81 Postel, J., "Internet Protocol: DARPA Internet Program Protocol Specification", RFC 768, USC/Information Sciences Institute, August, 1980.

Pouz82 Pouzin, L et al. (1982). *The Cyclades Computer Network*, North-Holland Publishing Company, New York, USA.

Shoc79 Shoch, J. F., Stewart, L.,: "Interconnecting Local Networks via the Packet Radio Network", *Proc. Sixth Data Commun. Symp.*, 153-158, 1979.

Stal84 Stallings, W., "A Primer: Understanding Transport Protocols", Data Communications, November 1984, 201-215.

Stal85 Stallings, W., "Data and Computer Communications", Macmillan, New York, 1985.

Stal86 Stallings, W., "A Tutorial on the IEEE 802 Local Network Standard", Local Area and Multiple Access Networks, Computer Science Press, 1986.

Svob84 Svobodova, L., "File Servers for Network-Based Distributed Systems", *Computing Surveys* vol 16, no 4, December, 353-398, 1984.

Svob85 Svobodova, L., "Client/Server Model of Distributed Processing", *Kommunikation in verteilten Systemem 1.*, Springer-Verlag, 485-498, 1985.

Svob86a Svobodova, L., "Communication Support for Distributed Processing: Design and Implementation Issues", *Networking in Open Systems*, IBM Europe Institute, Oberlech, Austria, 1986.

Svob86b Svobodova, L., Drobnik, O.,: "OSI Communication Services in a Local Area Network", IBM Zurich Research Laboratory, Ruschlikon, Switzerland, 1986.

Tane81a Tanenbaum, A. B., *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1981.

Tane81b Tanenbaum, A. B., "Network Protocols", *Computing Surveys* vol 13, no 4, December, 453-489, 1981.

TOP10 Technical and Office Protocols: A Communications Network for Open System Interconnection, version 1.0, The Boeing Company, Network Services Group, Seattle, WA98124-0346, November, 1985.

Zimm80 Zimmermann, H., "OSI reference model - the ISO model of architecture for open systems interconnection", *IEEE Trans. Commun.* :April, 425-432, 1980.