AN IMPLEMENTATION OF X.25 OVER TCP/IP

A Thesis Presented in Partial Fulfilment
of the Requirements for the Degree of
Master of Technology in Computing Technology
at Massey University

Ajitkumar J. W. Rasiah

1991

ACKNOWLEDGMENTS

I am indebted to my supervisor, Dr. Peter Kay of the Department of Computer Science, Massey University, for his invaluable advice, assistance and helpful criticism throughout this study and the preparation of this manuscript. His patience and encouragement are greatly appreciated.

I also wish to thank:

Professor Robert M. Hodgson of the Department of Production Technology, Massey University, for providing the opportunity for this study.

Professor Mark D. Apperley of the Department of Computer Science, Massey University, and the Massey University Computer Centre for providing the facilities for this study.

I would also like to express my sincere thanks to my parents Dr. and Mrs. B. W. Rasiah for their patience, help and encouragement during the course of this study.

CONTENTS

LIST OF FIGURES

ABSTRACT

Computer Networks are being used increasingly around the world. More importantly, many of these networks are interconnected by gateways, allowing a user in one geographical location to send and receive messages from a host located elsewhere.

The International Standards Organisation (ISO) has proposed the Open Systems Interconnection (OSI) Reference Model as a basis for building computer networks and the protocols which are used on those networks. Developers of networks are encouraged to follow these guidelines so that their networks may have an 'open architecture'.

This thesis examines two networking protocols, CCITT's X.25 Recommendation and TCP/IP. It continuously refers back to the OSI Reference Model as it describes the design of these protocols. It then looks at an implementation of X.25 over TCP/IP, which will allow users on non-X.25 hosts to develop and run X.25 applications courtesy of a separate host that supports X.25.

CHAPTER 1

INTRODUCTION

1.1 The Rapid Growth of Computer Networks

Computer networking is an area of technology which has rapidly grown over the past decade. A few years ago networks were research tools used by a few specialist organizations. Over the period of a few years computer networks gained a reputation for usefulness and reliability and many more organizations recognized the necessity to invest in a public or private network of some kind for daily use with machines ranging from personal computers to supercomputers. A computer network offers a practical means of communication and eliminates the problem of isolation which many organizations have previously experienced. Organizations have come to rely on computer networking to an extent that they are almost helpless without one. World-wide electronic mail is used daily by millions of people. Over the past decade networks have evolved from being a research tool used by academics to an essential tool for users in business, government and research institutions.

As networking evolved, every computer manufacturer had a different networking architecture that was incompatible with the products of other manufacturers. This is now changing. Almost the entire computing industry has agreed on standardizing the architecture of networks. The spin-offs from the resulting standards are already apparent. The products of one manufacturer can communicate easily with the products of other manufacturers.

The advantages of using computer networks are immense. One of the most significant is resource sharing. This enables a community of users in one geographical location to share resources with another community somewhere else. Most universities and research institutions nowadays make their computing facilities available to remote users via resource sharing networks. This means that a lecturer or student at one university can employ the facilities at any other university on the network.

Some networks provide access to specialized facilities. For example, the New York Times has made available millions of abstract news items dating back to the early 1960's. These can be accessed by journalists using a suitable query language. Some corporations have a

large number of computer solutions for engineering design or evaluation. These are made available to engineers throughout the corporation via a corporate network.

1.2 International Standards Organizations

Computer networks may be found in most countries today. The equipment and protocols that are used is varied and so it is necessary to define standards for obtaining maximum compatibility. The ISO, CCITT and IEEE organizations make major contributions towards standards in many areas.

The International Organization for Standardization (ISO) is a body which produces recommendations of all types. It has a computer technical committee which is responsible for all types of computer standards. One of its most important achievements is the Open Systems Interconnection (OSI) reference model.

The International Telegraph and Telephone Consultative Committee (CCITT) is a United Nations committee which makes recommendations about telephone and data communications services. The recommendations are revised every four years. Many of the CCITT's recommendations have been adopted to an extent that they have become the standard. X.25 is one such example.

The Institute of Electrical and Electronics Engineers (IEEE) is a large, professional organization which has comprises of a number of smaller interest groups to which members can belong. One of these groups deal with standardization and develop standards in the electrical engineering and computing areas. IEEE's 802 standard for local-area networks is widely used and has subsequently been taken over by ISO as the basis for ISO 8802.

1.3 Protocols

To enable the increasing variety of computers to communicate with one another, there must be a well-defined set of rules about how messages are exchanged between machines. These rules are known as protocols. A protocol is an agreed set of rules and procedures which, if followed by all participants, will allow the orderly transmission of information among the participants.

Many different protocols exist for use over computer networks. A protocol may be used on its own or as part of a 'layer' or hierarchy of different protocols which collectively make up a networking system. Two of the most well-known protocols used today are X.25 and TCP/IP.

1.4 Reference Models

To reduce the complexity in network design, most networks are organized as a hierarchy of levels or layers. This approach was conceived when it was realized that the functions required for data communications are best implemented in a hierarchical fashion. The purpose of each layer is to offer certain services to the layers above whilst shielding them from the details of how the services are actually implemented. A layer on one machine communicates with the corresponding layer on another machine using a protocol suitable for servicing that layer. In this instance, the protocol will not be suitable for use with other layers. The set of layers and protocols is called the network architecture. The network architecture is based on an abstraction called the Reference Model.

The most widely known reference model is ISO's OSI. Work on this was first completed in 1979. In June 1982 it was proposed as a draft international standard and it was promoted to an international standard at the end of 1983. The OSI reference model for network architectures contains seven layers. In reality information is not transferred directly from layer n on one machine to layer n on the peer. Instead, each layer passes data and control information to the layer below it until the lowest layer is reached. Below layer 1 is the physical medium through which the actual communication occurs. At the peer information which is received is taken from the physical medium and passed to the layers above until the originating layer is reached. International standards exist for all seven layers.

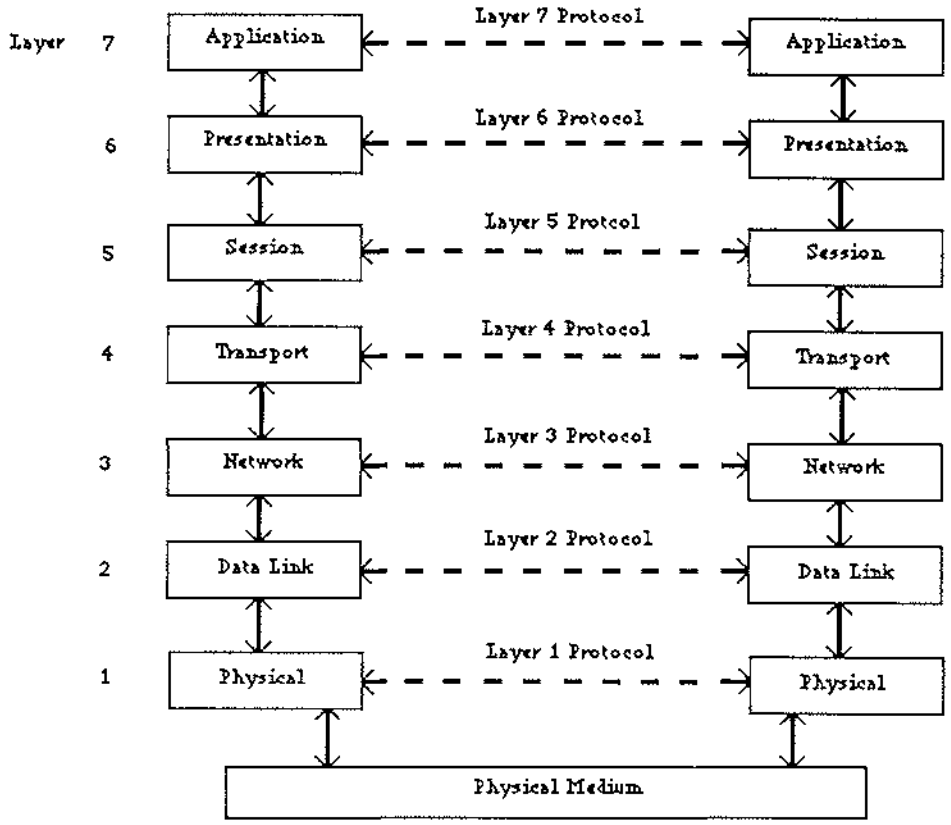


Figure 1.1 - The OSI Reference Model

Layer 1, called the physical layer, is responsible for transferring raw bits from one end of a communication channel to the other. The design issues in this layer are mainly concerned with the mechanical and electrical interfaces to the network and the physical transmission medium.

The data link layer (layer 2) is responsible for taking the raw transmission facility and transforming it into a line that appears free of transmission errors. This is done by breaking down the data that is passed to it by layer 3 above into data frames, typically containing a few hundred octets, and adding error checking information. Layer 2 then passes them to layer 1 which transmit the frames sequentially. The receiver typically uses the error checking information to check that the frame has not been corrupted during transit by noise on the line. It then sends back an acknowledgement frame to the sender, or requests the sender to retransmit the frame if data corruption has occurred.

Layer 3, the network layer, is concerned with how packets are to be routed through the network. It uses the error free channel provided by the data link layer below to

communicate packets of data between different hosts on the network. Layer 3 must also control the maximum number of packets in the network at any time. If too many are in transit, the network will become congested and it's performance will degrade.

The transport layer (layer 4) is responsible for accepting data from the session layer, splitting it up into smaller pieces if necessary, passing these to the network layer, and ensuring that the pieces all arrive correctly at the other end. All this must be done efficiently, and in a way which makes the session layer independent of changes in the hardware technology. The transport layer must create network connections as required by the session layer. If the session layer requires a high throughput then the transport layer may create multiple network connections and divide up the data among the network connections to improve throughput. Alternatively, the transport layer may multiplex several transport connections onto a single network connection if creating and maintaining a network connection is expensive. The transport layer also determines the type of service to provide the session layer, such as message broadcasting to multiple destinations, message transmission with no guarantee of the order of delivery, or an error-free point-to-point channel which delivers messages in the order in which they were sent. Transport layer protocols have end-to-end (host-to-host) significance, meaning that this layer and the layers above it are implemented on the host, rather than in the network. The transport layer also provides packet resequencing and flow control to ensure that data arrives safely at the destination.

Layer 5, the session layer, allows users on different machines to establish sessions (connections) between them by providing procedures for initializing, controlling the dialogue and terminating the sessions. A session might, for example involve a user logging onto a time-sharing remote host to use a facility that it offers. One of the services that the session layer provides is dialog control, where it must determine whether the message traffic is exchanged in a full-duplex or a half-duplex fashion. Another essential session service is synchronization. If a large file transfer crashes, then retransmitting the entire file can be avoided since the session layer is able to provide checkpoints into the data stream and therefore only the data after the last checkpoint must be repeated.

The presentation layer (layer 6) provides commonly used functions such as text compression or conversion between different file formats. These functions are requested often enough by users to warrant providing solutions for them rather than letting them solve the problem each time. The presentation layer relies on the underlying layers to

provide error-free data transmission. This way, the presentation layer is free to concentrate on the syntax and the semantics of the data.

Layer 7, the application layer, is the highest of the seven layers. This provides a means by which application programs can gain access to the OSI environment for the purpose of exchanging information with application programs on remote hosts.

The network layer and below are largely used by the CCITT X.25 definition. However there have been some alterations at layer 3 since the existing X.25 definition was not completely satisfactory for direct use in the OSI environment.

Variations of the OSI reference model exist under different names. One widely used variation is the TCP/IP Internet Layering model where applications reside directly above the transport layer and have application-specific session and presentation layers. This model forms the basis of the TCP/IP protocol suite.

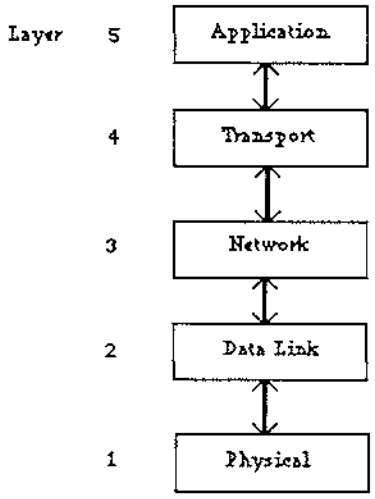


Figure 1.2 - The TCP/IP Internet Layering Model

1.5 Objectives of the Project

This thesis examines two widely used networking protocols, X.25 and TCP/IP. It covers much of the details pertaining to layers 1 to 3 of their architecture. It then describes the design and implementation of software for running X.25 over TCP/IP.

The implementation of X.25 over TCP/IP was stimulated by the networking requirements of the Department of Computer Science at Massey University. The department runs two Ethernet local-area networks which are used to connect a wide variety of devices. These devices include Sun workstations, Apple Macintosh and IBM Personal Computers, and a laser printer. TCP/IP is run over the Ethernet which allows an already extensive networking environment to grow even further. However, the computing facilities at the department do not include a connection to a public data network. Therefore software development in the networking area is restricted to the TCP/IP environment.

The Massey University Computer Center runs a Pyramid minicomputer, which is also connected to the Internet. However, this machine has the added feature that it is connected to a public data network. This provided the motivation to develop software that would allow users at the Department of Computer Science to write local X.25 software, use TCP/IP to reach the Pyramid, and use the Pyramid's connection to the public data network.

The implementation was developed in 'C' under the UNIX operating system. The project consisted of three parts. A programmer's interface in the form of an X.25 library was written to provide users with routines for opening and closing virtual circuits, sending and receiving data in the form of suitably formatted X.25 packets or data, and getting status information for connections. On the Pyramid side, two server programs were written to provide the X.25 services for the X.25 library.

Chapter 2 and Chapter 3 describe in detail the X.25 and TCP/IP protocols respectively.

Chapter 4 describes some underlying, medium-specific network protocols used for the data link layer which are essential for the operation of X.25 and TCP/IP. It describes how these protocols are used to transport X.25 and TCP/IP packets.

Chapter 5 describes the implementation details of this project and presents some examples of how the implementation may be used.

Chapter 6 goes on to propose further developments to the existing system.

CHAPTER 2

CCITT X.25 RECOMMENDATIONS

2.1 Packet Switching Data Networks

Packet Switching Data Networks (PSDN) are networks designed to carry data over medium to long distances. They are based on a transmission technique called Packet switching. Packet switching is a technique where large blocks of data are broken down into smaller chunks of a fixed length (usually of 128 octets) called packets. These packets are then transmitted through the network to a destination host.

The route that each packet takes is determined by the network. Each packet contains sufficient control information for it to reach its destination and be reunited with the other packets in the correct order. Once a packet reaches its destination the control information is removed and the data is assembled to form the original message.

A PSDN is a wide-area network (WAN), in contrast to a local-area network (LAN) such as Ethernet. Wide area networks consist of widely distributed packet switching exchanges (PSE) which are connected together using high-speed lines. A host device commonly known as Data Terminating Equipment (DTE) is connected to the PSDN via a modem, commonly known as Data Circuit Terminating Equipment (DCE). A Network Access Protocol is used for exchanging information across the access link to the PSDN. The PSDN can then be relied upon to transport information reliably to the remote DCE and DTE.

The packet switching concept was first proposed in the early 1960's by Paul Baran at the RAND Corporation whilst working on a military communications system mainly handling speech. The primary requirement was to have a network which would retain some functionality in the event of damage due to hostile action. ARPANET was the first private packet switching network which derived its techniques and management from TELENET, the first major public packet switching network. TELENET was introduced by the American Telenet Corporation in 1975. It is a value-added network, a type of network constructed from leased lines to serve many customers in selected geographical areas. Telephone companies which operate nationwide packet switching networks are known as common carriers. They provide a number of different kinds of networks but have the

advantage over other networking vendors in that they have access to very high-speed and very reliable links.

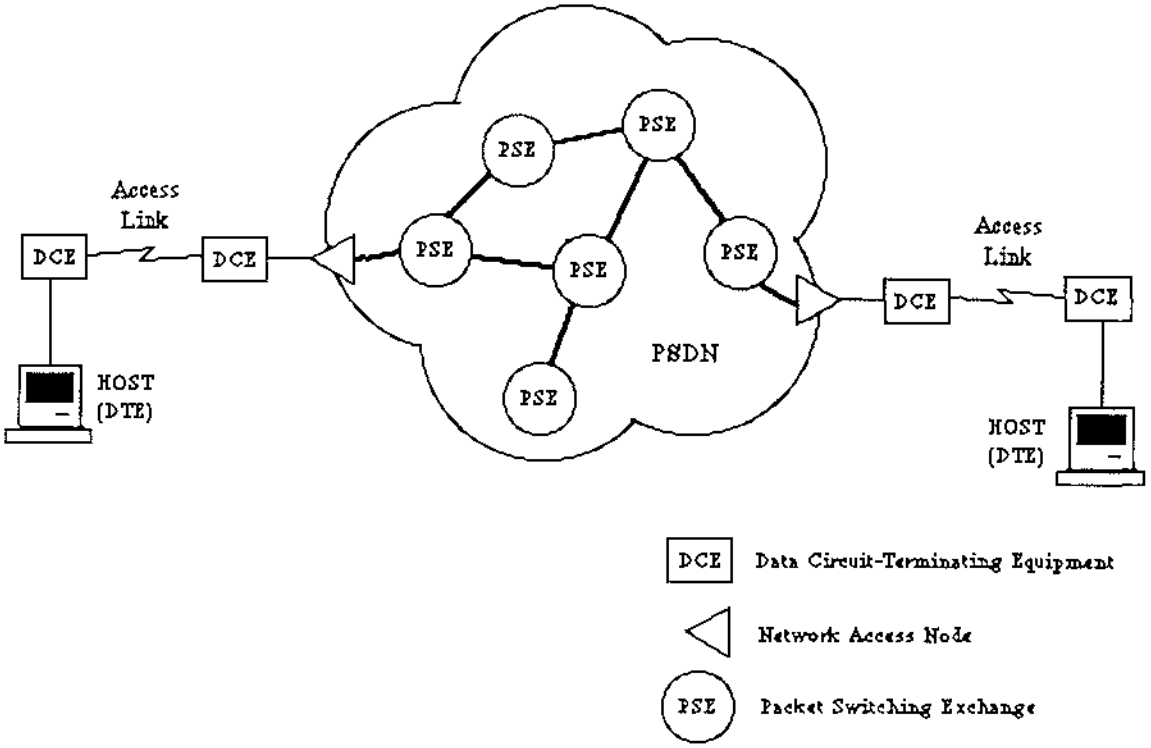


Figure 2.1 - Topology of a Packet Switching Data Network

2.2 CCITT X.25 Recommendations

The CCITT X.25 recommendation defines the interface between the DTE and the DCE. The DTE is the user's equipment which is typically a host computer or terminal. The DCE is the carrier's equipment which provides the interface between the DTE and the network. X.25 defines the format and the meaning of the information exchanged across the DTE - DCE interface. X.25 is a network access protocol which does not make any assumptions about the way in which the network functions except that packets are used in the interaction between two DTEs and that these packets are delivered in the order in which they enter the network. Therefore two DTEs employing X.25 should be able to communicate via a network which appears transparent to them.

2.3 The Structure of X.25

X.25 is made up of three distinct layers corresponding closely to levels 1, 2 and 3 of the OSI reference model architecture (figure 1.1). The procedures at one layer utilise the functions of the layer immediately below but do not assume any knowledge of its implementation. The X.25 specification defines the format and meaning of the information exchanged across the DTE-DCE interface for these three layers.

Layer 1 is the physical layer. It references the X.21 and X.21 bis standards which define the digital and analog interfaces respectively. This layer deals with the electrical, mechanical, procedural and functional interface between the DTE and the DCE. It is characterized by bit-serial, synchronous, full-duplex, point-to-point connections.

The second layer (the data link layer) provides reliable communication between the DTE and the DCE for packets generated at level 3. X.25 provides error detection and flow control by using the frame structure of the High-Level Data Link Control (HDLC). The HDLC frame structure (figure 2.2) encapsulates the raw bits which are being transmitted. It contains addressing information, control information and a Frame Checking Sequence (FCS). It is also preceded and succeeded by a special flag character. X.25 utilizes a Link Access Procedure (LAPB), a variant of HDLC, which specifies that either station may send commands at any time and initiate responses without receiving permission from the remote device.

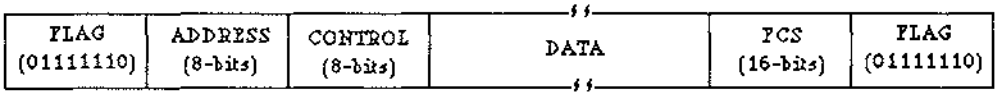


Figure 2.2 - HDLC Frame Structure

Layer 3 is the Packet Layer. Central to X.25 is the concept of the connection (virtual calls in CCITT terminology). The X.25 layer 3 manages connections between a pair of DTEs. Layer 3 also defines a number of different packet types and the transitions which occur when the packets are exchanged between DTEs during virtual calls. In addition to data transfer, packets are used for error recovery, interrupt facilities and flow control between the DTE and the DCE.

2.4 Virtual Calls

Two kinds of connections are provided, these being virtual calls and permanent virtual circuits. A virtual call is analogous to an ordinary telephone call where a connection is established, data are transferred, and the connection is released. In contrast a permanent virtual circuit is like a leased line that is always present, and the DTE at either end can just send data whenever it wants to without any set-up.

Each virtual call is assigned a unique identifier by the originating DTE known as the logical channel number (or virtual circuit number). An X.25 terminal can support several virtual calls simultaneously, where each call is assigned a logical channel number and all packets related to that call carry that channel number. Both the DTE and its DCE use the same logical channel number. Hence X.25 layer 3 provides a multiplexing function by converting the single channel provided by layer 2 into a number of logical channels. The choice of circuit number on outgoing calls is determined by the DTE, and on incoming calls by the DCE.

A call collision occurs when the DTE sends an outgoing call and the DCE receives an incoming call, both choosing the same virtual circuit number for their respective calls. Since each call must have a unique virtual circuit number, the call collision is resolved by getting the DCE to abandon the incoming call and allowing the outgoing call to proceed. To minimize the chances of getting a call collision, the DTE normally chooses the highest available identifier for outgoing calls and the DCE chooses the lowest one for incoming calls.

2.5 X.25 Packet Formats

An X.25 packet may either be a control packet or a data packet. The first few octets of a packet is called the header and contains control information for higher level protocols. The control information varies across packet types. Packets fall into three broad categories, namely call set-up & clearing packets, data & interrupt packets, and resetting & restarting packets. Figure 2.3 presents the four main packet formats used in X.25.

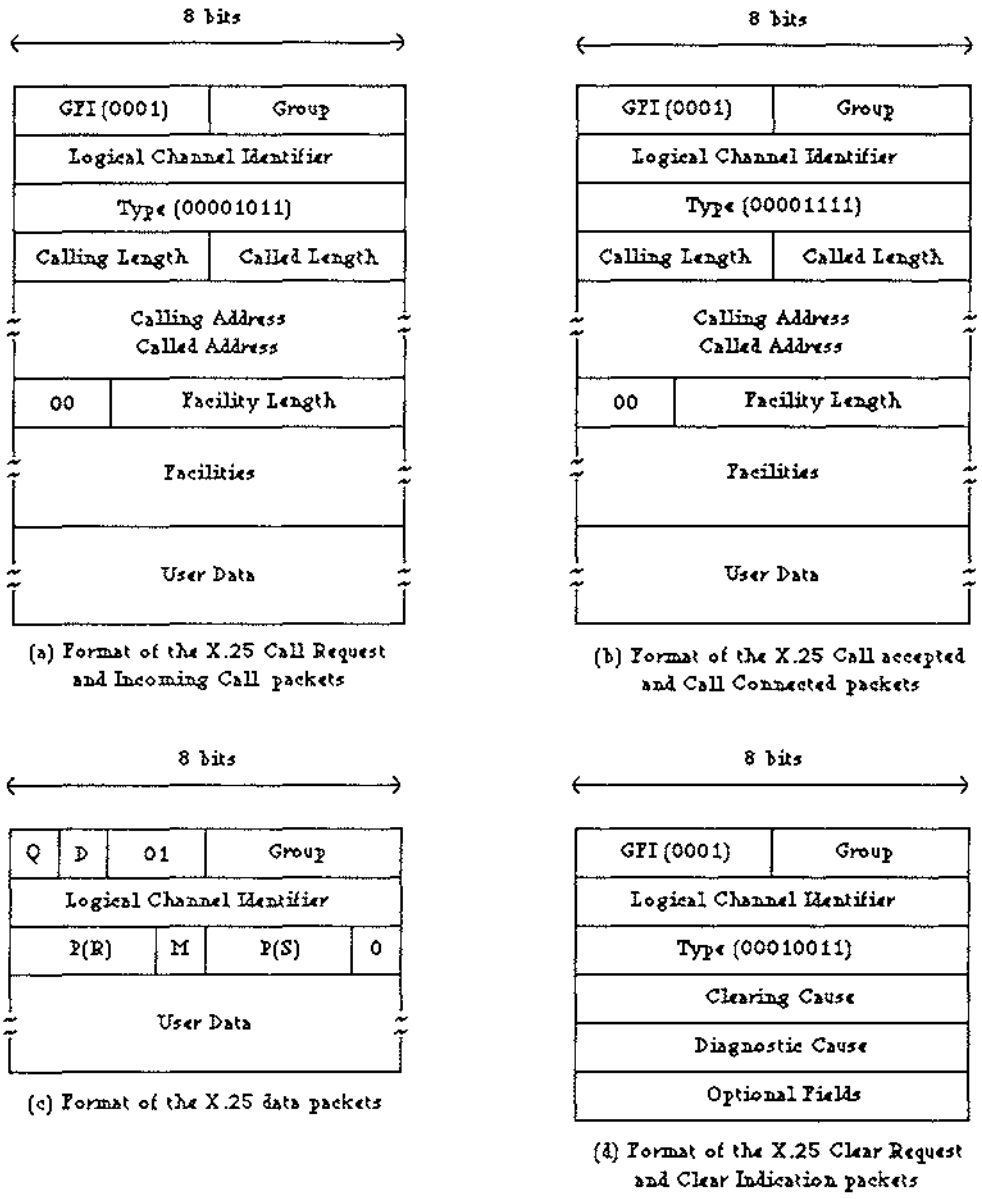


Figure 2.3 - Main X.25 Packet Formats

2.5.1 Call Set-up and Clearing Packets

A connection is made in the following manner. When a DTE wants to communicate with another DTE it must first set up a connection. To do this, the DTE builds a CALL REQUEST packet and passes it to its DCE. The subnet delivers the packet to the destination DCE which then gives it to the destination DTE in the form of an INCOMING CALL packet. If the destination DTE wishes to accept the call it sends a CALL ACCEPTED packet back. The originating DCE receives the CALL ACCEPTED packet and

passes it on to its DTE as a CALL CONNECTED packet. Thus when the originating DTE receives the CALL CONNECTED packet, the virtual circuit is established.

Once a virtual call has been established the full-duplex connection may be used to exchange data packets. When either side has had enough it sends a CLEAR REQUEST packet to the other side which arrives as a CLEAR INDICATION packet. The recipient DTE must respond with a CLEAR CONFIRMATION packet as an acknowledgement.

2.5.2 Data and Interrupt Packets

After the call set-up phase is completed and a virtual call has been established, data may be exchanged across the subnet in the form of data packets. These packets contain three octets of header information followed by 0 to 128 octets of data.

An INTERRUPT packet is a high priority data packet which can overtake normal data packets to inform the destination DTE of a special event such as when a specific key has been struck on the remote DTE's keyboard. Unlike normal data packets, an INTERRUPT packet does not contain packet sequence counts and can therefore by-pass normal flow control procedures. Only one INTERRUPT packet may be outstanding in a virtual call at any time and should be acknowledged with an INTERRUPT CONFIRMATION packet as soon as possible.

2.5.3 Resetting and Restarting Packets

During the data transfer phase it may become necessary to reset a virtual call. For example, if a gap has been detected in the packet sequence numbers, the simplest solution will be to reinitialise the call so that it appears as though it has just been set up and then begin the data transfer phase again. This can be done by issuing a RESET REQUEST packet to the remote DTE which must respond with a RESET CONFIRMATION packet. Resetting a call will result in the next expected sequence number being set to 0 and all data packets in the network being discarded.

When a RESET REQUEST packet is issued by a DTE, the remote DTE must respond with the RESET CONFIRMATION packet for the call to be reset. Similarly when a RESET REQUEST is issued by the network, both DTE's must respond with RESET CONFIRMATION packets for the call to be reset.

Restarts are used for more severe conditions than those resulting in a reset. A restart reinitialises all calls by clearing all virtual circuits and resetting all permanent virtual circuits. A RESTART REQUEST packet may be issued, for example when a DTE or a DCE has crashed and is forced to abandon its connections.

2.6 Flow Control

The third octet inside a data packet contains two sequence numbers, P(S) and P(R). These represent the number of data packets that have been sent and received respectively. When X.25 was first introduced these were only of significance across the DTE-DCE interface and were not used for end-to-end flow control. This was because X.25 was intended as an interface specification and therefore no assumptions could be made that the number of packets identified by P(R) had actually reached the destination DTE. This lack of end-to-end significance caused much concern and the 1980 revision introduced the D (delivery confirmation) bit for end-to-end flow control.

The D bit is bit 7 in octet 1 of the header in a data packet. If D is 0 in a data packet, then the P(S) and P(R) counts are of significance across the local DTE-DCE interface only and are independent of network propagation delays. As a result there could be many packets in transit at the same time in the network, which was the intention of the original recommendation. If D is 1 in a data packet, then it can be assumed that P(R) number of packets have reached the destination. The use of this facility requires additional communication between the DCEs which results in a significantly lower throughput especially on large networks where the sending DCE must wait for delivery confirmation from the remote DCE before acknowledging packets to its DTE.

The M (more data) bit is bit 5 in octet 2 of the header in a data packet. This bit is set to indicate to the receiving DTE that the logical unit of data is longer than can be fitted into a data packet and has therefore been divided into a sequence of smaller packets. In such a sequence all but the final data packet will have the M bit set to 1. The M bit is provided for higher level protocols and is not usually used by X.25 except in cases where the network may concatenate two or more short packets with the M bit set for reasons of operational efficiency.

The Q (data qualifier) bit is bit 8 in octet 1 of the header in a data packet. This bit is not used by X.25 and is used by higher level protocols. For example, the Q bit may be used to distinguish between commands and data.

2.7 Packet Assembly/Disassembly Devices

A terminal that can assemble and disassemble X.25 packets is referred to as a packet-mode DTE. Normally computers act as the packet assembler/disassembler (PAD) for a number of character-mode terminals which are connected to it. Character mode terminals cannot be connected directly to a PSDN. Instead, they must be connected to a PAD facility on the PSDN which performs the necessary packet assembly/disassembly.

2.8 Efficiency Considerations

PSDNs make an effort to work efficiently. Within the network there will normally be packets in transit belonging to several users. The PSDN uses a procedure called packet interleaving which uses gaps in the flow of packets to carry new packets entering the network. This means that a particular user's packets will not necessarily flow through the network consecutively. By filling these gaps wherever possible, PSDNs can use their lines to full capacity. The PSDN does guarantee that a user's packets is received in the same order in which they were sent, and that two or more users' packets will not get mixed up.

PSDNs are responsible for routing packets through the network. This means that packets may have to be rerouted if the chosen path is not available or may not be the most efficient. Regardless of the path that is taken, packets still arrive intact and in the correct order at the destination.

When a packet arrives at its destination, the PSDN sends an acknowledgment from the remote DCE. A DTE may transmit more packets as acknowledgements arrive for previous packets that were sent. The windowsize is the number of packets which may be transmitted without an acknowledgement. The windowsize limits the maximum number of packets which have not yet been acknowledged. If the windowsize is reached then no more packets can be sent until an acknowledgement arrives. Incoming acknowledgements are often carried by packets that are also carrying data for maximum networking efficiency.

There are two kinds of windowsizes. The frame windowsize is fixed by the PSDN and is the number of LAPB frames which may be transmitted from the DTE to the PSDN without an acknowledgement. The packet windowsize is the number of packets which may be sent without an acknowledgement. When subscribing to a PSDN users can specify the windowsize. Users can also specify a packetsize at subscription time. This is the amount of

data which can be sent inside a packet. This is usually 128 octets but smaller or larger sizes are available.

CHAPTER 3

THE INTERNET PROTOCOL SUITE

3.1 Background

The Internet is a collection of networks comprising of a number of local area networks at university and research institutions, regional and military networks. These networks are connected to each other and allow users to send messages from any of them to any other except where restrictions have been imposed for security reasons.

The Internet Protocol Suite is a set of protocols developed to allow computers to communicate among host computers across the Internet. The suite includes protocols such as User Datagram Protocol (UDP), Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP) and Internet Protocol (IP). Of these, TCP and IP are the best known and therefore the Internet Protocol Suite has become commonly known as TCP/IP.

Initially TCP/IP was used mostly between mainframes or minicomputers and so TCP/IP was traditionally used for:

a) File Transfer - The File Transfer Protocol (FTP) allows a user to get files from, or send files to another computer. The user has to provide a usercode and a password before being allowed to proceed with the transfer. Provisions are made for handling file transfers between machines with different character sets.

b) Remote Login - The Network Terminal Protocol (TELNET) allows a user to log in on any other computer on the network. A remote session is started by specifying the computer to connect to, as well as a usercode and a password. The TELNET program uses this information to log into the remote computer and thereafter whatever is typed on the local keyboard is transmitted to the remote machine. The session is terminated by logging off the remote computer at which point the TELNET program exits and control of the local computer is restored.

c) Computer Mail - This allows users to send messages to other users with computers connected to the Internet. Initially, computer mail had problems where the people used

microcomputers. A primary problem was that microcomputers were often switched off when not in use, and the mail programs worked on the assumption that these machines were always on. For this reason, nowadays mail is normally handled by a larger system where a mail server is running at all times. When switched on, the user's computer retrieves mail from the mail server.

These services should be present in any implementation of TCP/IP, with the exception of some micro-based implementations which may not support computer mail.

3.2 Layering in the Internet

The Internet Protocol Suite is a layered set of protocols. In a similar fashion to the layering principles described in section 1.4, a layer provides services to the layer above whilst shielding it from the implementation details of the layer below. Generally TCP/IP applications are modelled on the TCP/IP Internet layering model (figure 1.2) as follows:

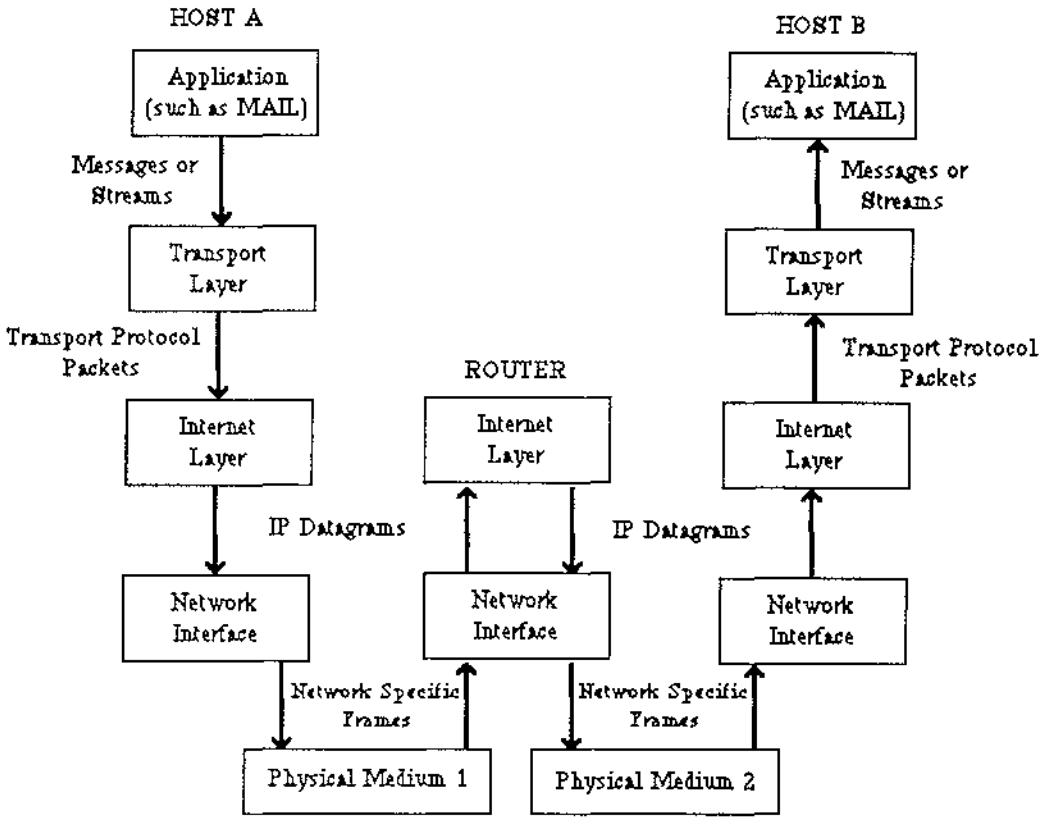


Figure 3.1 - Layering in TCP/IP using a router

An application such as mail is written at the topmost layer. The application communicates with a similar mail application on a peer host by breaking down its transmission into messages. The messages are passed to the transport layer to a protocol such as UDP, TCP or ICMP. The transport layer protocol encapsulates the message into packets which are then passed to the Internet layer below. The Internet layer takes these packets and encapsulates them into IP datagrams. The IP datagrams are passed to a network-specific protocol which is used to manage a specific medium such as Ethernet. The network-specific protocol encapsulates the IP datagrams passed to it from the layer above into frames. These frames are then transmitted along the physical medium.

TCP/IP is based on the assumption that a large number of independent networks have been connected together by IP-layer routers. A router is a switch that receives data transmission units from input interfaces, and depending on the addresses in those units, routes them to the appropriate output interfaces. The user should therefore be able to access other computers and resources on any of these networks. Datagrams will often pass through a number of different networks before reaching their final destination and the routing needed for this should be completely invisible to the user.

In figure 3.1, frames are passed from Host A on one network to Host B on a different network via the router which connects both networks. When a frame arrives at Host B's network, it is sent upwards through the layers. Information is decapsulated at each layer until the original message is extracted and passed to the mail application.

3.3 Internet Addressing

Each computer interface connected to the Internet is given a unique 32-bit Internet address which is normally written as four 8-bit numbers separated by dots, for example 130.123.3.10. The structure of the address generally gives some information about how to get to the system. For example, 130.123 is a network number which has been assigned to Massey University. 130.123.3 is an Ethernet in use by the Department of Computer Science, and 130.123.3.10 is a particular host on that Ethernet. Internet addresses are assigned by a central authority. Many users prefer to call computers by name, rather than having to remember a 4-octet number. This is easily done by using a database to hold a list of all commonly used network addresses and the associated domain names. One computer may have more than one domain name. When a host on a network is referred to by name,

the networking software searches the database for the name and comes up with the corresponding Internet address.

Similar to X.25 which works on the concept of a virtual circuit, the TCP at the transport layer is connection-oriented but accesses the underlying network by utilizing the services of the connectionless IP at the network layer. A stream of data is systematically encapsulated into a number of frames and each frame is sent individually through the network. The network does not assume that the frames are related in any way and it is quite likely that two logically consecutive frames may arrive at their destination via different routes. It is also possible that some frames may not get through because of an error and may have to be sent again.

3.4 The Transmission Control Protocol (TCP)

The TCP offers a full-duplex connection service to two end-peers. It implements a three-way handshake for connection establishment and uses a windowing scheme for flow-control purposes. The TCP is responsible for breaking up a stream of data into packets, reassembling them in the right order at the destination and resending any packets which may have got lost or discarded due to errors. It is quite possible that the destination host has more than one connection currently open. Therefore in addition to being able to reassemble packets in the correct order, the TCP must be able to identify the particular connection to which a packet belongs. This is known as 'demultiplexing'. To perform demultiplexing, TCP/IP uses a series of headers at the start of packets. The TCP breaks down a message into packets and attaches a header in front of each one. The packet then looks like this:

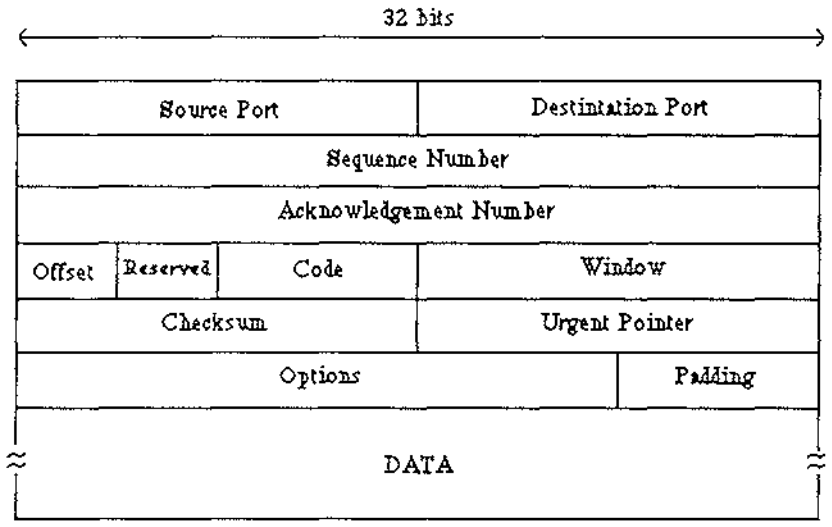


Figure 3.2 - TCP Packet Format

The TCP packet header contains information such as the source and destination port number and sequence number. The port numbers are used to keep track of different connections and the sequence number is used to reassemble the packets at the destination in the correct order. For error checking, a checksum is also included in the header which is used by TCP on the destination host to ensure that the packet received has not been altered in any way.

The TCP provides end-to-end flow control with the use of an acknowledgement field in the packet header. In order to ensure that a packet has arrived at its destination, the recipient sends back an acknowledgement packet which has the acknowledgement field containing the number of octets received so far. The sender repeats the transmission if it does not receive an acknowledgement packet within a certain length.

TCP/IP does not require each packet to be acknowledged before another can be sent as this would become inefficient.Therefore, many packets could be in transit at any one time. This may become a problem if the sending machine is sending packets faster than the receiving machine can accept and process them. Therefore it becomes necessary to restrict the number of packets which can be in transit at a particular time. This is done using the window field in the packet header. The receiver puts the number of octets that it is prepared to receive inside this field and the sender ensures that there cannot be more than this number of unacknowledged packets at any time. The receiver may frequently change the value in the window field depending on how much free space it has remaining to hold

incoming packets prior to processing. Often the same packet is used for acknowledging the receipt of data and specifying the amount of new data which can be sent.

Similar to the X.25 INTERRUPT packet, TCP's urgent field in the packet header can be used to inform the other end of a specific event, for example if the user typed a control character. It is up to the application protocol to decide how this is handled.

3.5 The Internet Protocol (IP)

The packets which have been created in the TCP are passed down to the IP at the network layer. The IP in TCP/IP is a connectionless service which is responsible for routing individual packets through the network to their assigned destination. The TCP must provide information about where to send the packets and it is the IP's job to find a suitable route to the destination.

IP encapsulates TCP packets into IP datagrams by appending its own headers:

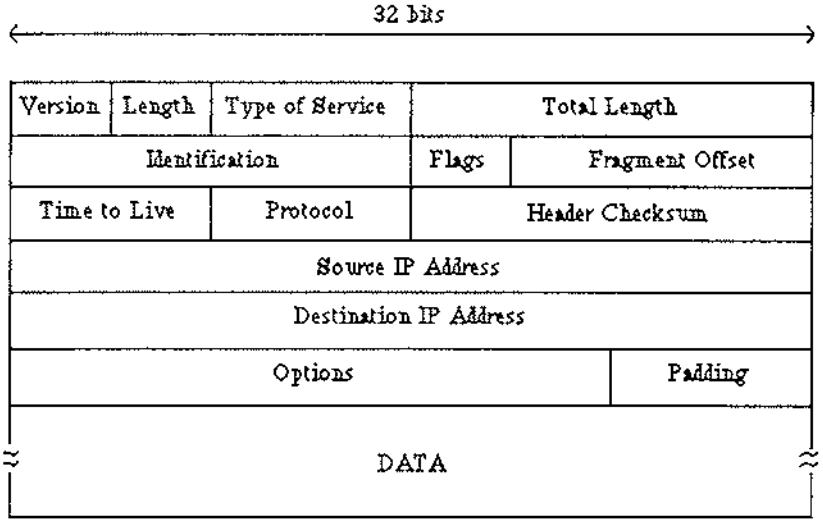


Figure 3.3 - IP Datagram Format

A 4-bit version field specifies the IP protocol version and is used to verify that the sender, receiver and gateways in-between agree on the format of the datagram. If standards change then machines will reject datagrams with version numbers different to their own to avoid the chance of misinterpreting the datagram contents.

The length field gives the length of the header, measured in 32-bit words. The most common type of header (without options) is 20 octets long, so the length field contains a value of 5. The total length field gives the length of the whole IP datagram, including the header.

The type of service field specifies how the datagram should be handled. The field is subdivided into a number of smaller fields called Precedence, D, T & R. The Precedence field is used to give the datagram a precedence ranging from 0 to 7. This describes the importance of each datagram. The D bit requests low delay, the T bit requests high throughput, and the R bit requests high reliability.

This header contains information such as the 32-bit source and destination Internet addresses, a protocol number and another checksum. The source Internet address is included so that the destination machine knows where the datagram originated from. The destination Internet address is necessary for gateways between networks to know in which direction to forward the datagram so that it will eventually reach its destination. The protocol number is used to tell the remote machine which protocol the datagram belongs to. Because TCP/IP is a layered protocol suite where protocols such as TCP, UDP and ICMP use IP as the primary transportation mechanism, the remote host must be told the nature of the datagram so that it is interpreted correctly.

The time to live field specifies how long (in seconds) the datagram is allowed to remain in the Internet system. Whenever a host transmits a datagram into the system, it sets a maximum time that the datagram should survive. Each router along the path from the source host to the destination host checks the time remaining when they process the datagram. Routers decrement the time to live value by one unit if the time to live is greater than zero, otherwise they discard the datagram if the time to live has reached zero. This ensures that a datagram does not travel in the Internet forever.

3.6 Underlying Network Services

IP datagrams are passed to a network-specific protocol which is used to manage a specific medium. Several such protocols exist to service different transmission mediums. These are described in Chapter 4.

CHAPTER 4

UNDERLYING NETWORK SERVICES

Chapters 2 and 3 described two protocols which are based on the OSI Reference Model. The protocols X.25 and TCP/IP have network architectures comprising of layered protocols. It can be seen from figure 1.1 that underneath the network layer lies the data link layer and the physical layer. The data link layer provides an error free channel for the network layer above. The physical layer transmits the raw data bits across the medium.

A number of different physical mediums can be used. The data link layer protocol that is used depends on the physical medium. Together, the physical medium and its data link protocol provide underlying network services to the layers above. This chapter describes some of these network services.

4.1 Link Access Protocol B (LAPB)

IBM's Synchronous Data Link Control (SDLC) protocol is a data link level protocol which was submitted for acceptance as an international standard. Instead, it was modified by ISO to become HDLC (High-level Data Link Control). CCITT then adopted and modified HDLC to LAPB as part of the X.25 network interface standard.

4.1.1 LAPB Frame Format

LAPB shares the same frame format as HDLC (figure 2.2). There are three types of LAPB frames. The Control field is used differently depending on the type of the LAPB frame. Not all frames contain a Data field, but the other fields are always present. The Data field contains arbitrary information ranging from user data to diagnostic information for use by LAPB. The Data field may in theory be of any length, although in practice many implementations insist on it being an exact number of octets long. The frame does not specify the length of the Data field. Instead, it is possible for a link to detect the frame delimiter (Flag field) and calculate the length of the Data field.

Frames are separated by Flags. A Flag field contains the bit sequence 01111110 which precedes and succeeds a frame. Because of its significance as a frame separator, the flag

sequence cannot appear inside a frame. When there are no frames to be sent on the link, the flag sequence may be repeated indefinitely.

The Address field is used in one of two ways. On multidrop lines, it is used to identify one of the terminals. On point-to-point lines, it is sometimes used to distinguish commands from responses.

The Frame Check Sequence (FCS) field is a variation of the cyclic redundancy code, using CRC-CCITT as the generator polynomial. The variation allows lost Flag octets to be detected.

4.1.2 LAPB Frame Types

There are three types of LAPB frames. These are Information, Supervisory, and Unnumbered. The Control field is used differently depending on the type of the LAPB frame:

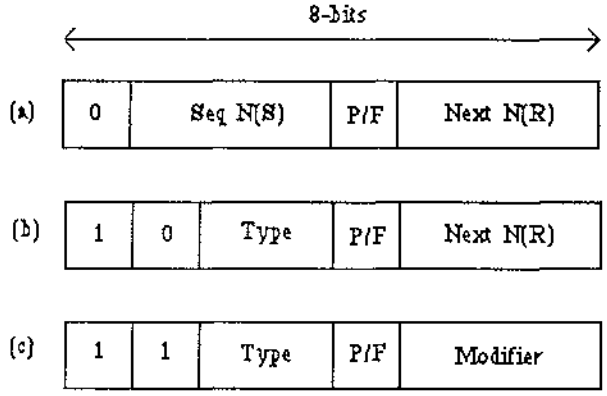


Figure 4.1 - Control field of a LABP frame

Information Frames

Information frames are used to send user data. The principle type of data is X.25 packets. The data is transported inside the Data field of the LAPB frame. The frame type is determined by the first bit of the Control field, which is a 0.

A send sequence count, N(S), represents the sequence number of the frame in modulo-8 sequencing. The receive sequence count, N(R), is the sequence number of the last received

frame plus one. In other words, it is the sequence number of the next unacknowledged frame. It also uses modulo-8 sequencing.

The P/F bit stands for Poll/Final. It is used when a computer is polling a group of terminals. When polling, the computer sets the bit to P. When the terminal responds, all frames sent by the terminal except the last one have the bit set to P. The final frame has the bit set to F.

Supervisory Frames

Supervisory frames are used to send commands. They do not contain an information field. The frame type is determined by the first two bits of the Control field, these being 10. There are various kinds of Supervisory frames, these being identified by the Type field. There are three types of Supervisory frames in LAPB. Type 0 frames are called RECEIVE READY, these being used to indicate the next frame expected.

Type 1 frames are called REJECT, these being negative acknowledgement frames used to indicate that a transmission error has been detected. The Next field indicates the first frame in the sequence which has not been correctly received, therefore requiring re-transmission. The sender must re-send all frames starting at N(R).

Type 2 frames are called RECEIVE NOT READY, and is used to tell the sender to stop sending because there are temporary problems at the receiver, such as a shortage of buffer space. It acknowledges all frames up to but not including N(R).

The P/F bit is used to force the other machine to send a Supervisory frame containing window information, rather than waiting for a normal transmission of this information.

Unnumbered Frames

Unnumbered frames are used for control purposes. The Type field is used to indicate the type of command being issued. LAPB shares some common commands with other bit-oriented protocols. One of these is a DISC (Disconnect) command that allows a machine to announce that it is going down. Another command, SNRM (Set Normal Response Mode), allows a machine that has just come back on line to announce that it has returned and reset all transmissions to it.

SABM (Set Asynchronous Balanced Mode) resets the line and declares the sender and the receiver as equals. SABME and SNRME are similar to SABM and SNRM respectively, except that they specify an extended frame format using modulo-128 sequencing.

FRMR (Frame Reject) is used to indicate that a frame with a correct checksum but incorrect semantics was received, for example if a Supervisory frame of Type 3 arrived. Although Type 3 frames are available in HDLC, they are not in LAPB.

4.2 Bus Networks

A bus network is characterized by a single coaxial cable called the bus which is run through one or more rooms. Hosts and associated interfaces are 'tapped' into the cable at any point. There is no central point, and all connections to the cable have an equal status. If a single coaxial cable is insufficient, then several sections can be joined together using devices called repeaters. Up to two repeaters may be placed between any two machines, making the total length about 1500 meters.

The advantages of such an arrangement are immense. Because all points on the cable have an equal status, hosts can be moved around freely. They are simply 'pulled out' of the cable and 'pushed in' elsewhere on the cable. Network broadcasting is simplified since any signals which are fed into the cable must reach all hosts tapped into it.

Hosts are connected to the coaxial cable by means of devices called tranceivers. These devices listen to the cable and can also transmit on the cable. Information placed on the cable by a transceiver will be propagated along it and reach all other transceivers. Thus by listening to the cable, a transceiver can 'hear' the traffic and pass on any frames heard on the cable to its host. The transceiver is connected to the host via a controller board. The controller has several functions. It encapsulates data from its host into a frame and passes it to the transceiver for sending. It also receives frames from the transceiver, decapsulates the data and passes on to its host whatever is intended for it.

Ethernet is one implementation of a bus network. It was originally developed in the early 1970's by Xerox Corporation to connect office workstations to computers and filestores. The main design aims included low running cost, reliability, no central controller, easy access to the transmission medium for all hosts, and suitability for handling 'bursty' traffic. The name "Ethernet" was derived from the coaxial cable called the ether.

Despite their success, bus networks have problems. For example, a transceiver may transmit only if the ether is free. By listening to the ether, a transceiver can avoid transmitting when another transceiver has control of the ether. However an obvious problem is when two or more transceivers simultaneously decide that the ether is free and proceed to transmit data. The result is that the colliding signals are destroyed. Therefore, a number of different protocols were developed to overcome this problem:

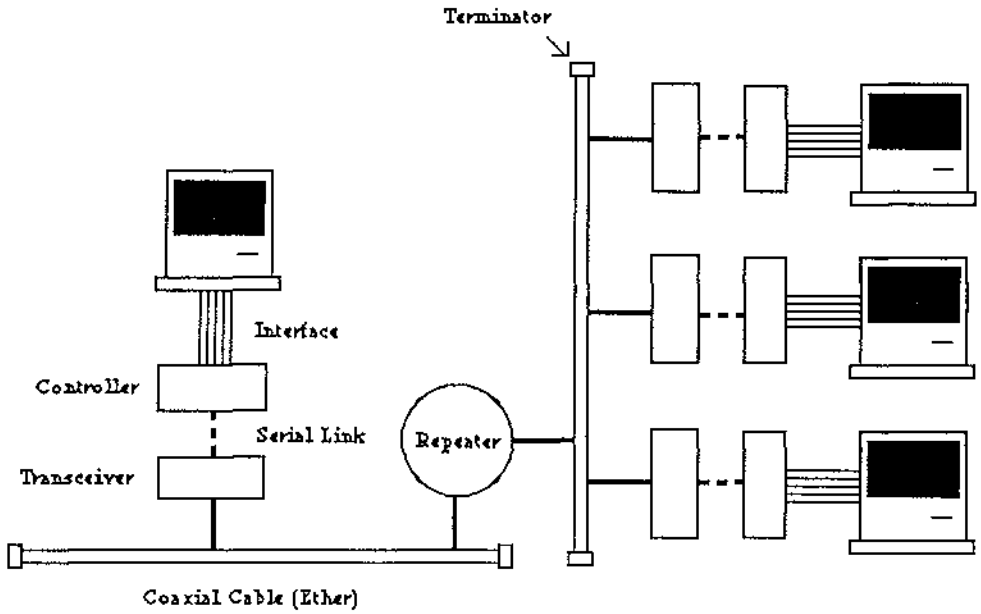


Figure 4.2 - Bus network Topology

a) Token Passing - This technique requires one station to generate tokens which are then passed between stations. A station must receive a token before it is allowed to transmit on the ether. It may however decline the offer to transmit, in which case it must pass on the token to another (pre-determined) station. The station which possess the token may transmit one or more frames of data to any destination. When it has finished transmitting, it must pass on the token to the next station.

b) Carrier Sense Multiple Access (CSMA) - This is a technique where the transceiver listens to see if the ether is free and if so, it transmits data but does not listen to what it has just sent. Under heavy loading, collisions will occur which will not be detected. Re-transmissions initiated by a higher level protocol will be necessary, and this will add to the congestion. CSMA is the simplest technique and it is normally used with the Collision Avoidance (CA) or the Collision Detection (CD) techniques described below, rather than on its own.

c) Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) - The Collision Avoidance Technique was originally used in the ALOHA packet radio network. In this technique, transceivers may transmit only during predetermined timeslots. These timeslots must be long enough to ensure that collisions will not occur. Transmissions are acknowledged by the acknowledger sending an acknowledgement during its own timeslot,

therefore preventing collisions even further. If all timeslots are unused, then the network enters a free-for-all state in which any station can transmit. Collisions may occur at this stage if more than one station chooses to transmit, in which case the transmissions are not acknowledged but are re-transmitted in an orderly fashion during the next round of timeslots. In the free-for-all state, no attempt is made to detect collisions.

A major problem with the CSMA/CA technique is that the more stations there are, the greater the number of timeslots which will be required. Consequently, there will be an increase in the number of unused timeslots leading to a wastage in bandwidth. Also, the longer the ether becomes, the longer each timeslots will have to be to ensure that no collisions occur. Therefore, CSMA/CA is best suited for networks with a small number of hosts in a small, localized area.

d) Carrier Sense Multiple Access with Collision Detection (CSMA/CD) - The Collision Detection Technique is the one used in standard Ethernet implementations. This was developed by the IEEE as part of the 802 collection of standards. CSMA/CD is discussed below.

4.3 IEEE 802

The IEEE 802 is a collection of standards for local area networks. The standards are divided into parts, each published as a separate book. The 802.1 standard gives an introduction to the set of standards. The 802.2 standard describes the Logical Link Control (LLC) protocol. Standards 802.3 through 802.5 describe the CSMA/CD, token bus, and token ring protocols respectively.

IEEE 802 divides the data link layer into two sections. The lower section is called the Medium Access Control (MAC) sublayer which has three variants, namely CSMA/CD, token bus and token ring. The upper section is called the Logical Link Control (LLC) sublayer.

4.3.1 IEEE 802.2 (LLC)

IEEE 802.2 is a standard which defines the Logical Link Control (LLC) sublayer which is used by standards 802.3 through 802.6. The LLC protocol is based closely upon HDLC and is therefore similar to the LAPB protocol described in section 4.1.

The LLC sublayer provides either a connectionless (type 1) or a connection-oriented (type 2) service to the network layer above. With the connectionless service, the LLC sublayer accepts a packet from the network layer and sends the packet to the destination with no acknowledgement or guarantee of delivery. When the connection-oriented service is used, a connection must first be set up between the source and the destination. The connection is then used to transmit packets from the network layer in order with guaranteed delivery. When the connection is no longer required, it is released.

The LLC services are made available to the network layer by means of primitives. In the OSI model, the service primitives can be divided into four classes:

a) Request Primitive - The Request primitive is used to request a specific task from the service, such as proposing a new connection with a remote host.

b) Indication Primitive - The proposal for the new connection reaches the remote host in the form of an Indication primitive.

c) Response Primitive - The remote host uses the Response primitive to accept or reject the proposed connection.

d) Confirm Primitive - The initiating host uses the Confirm primitive to find out the remote host's decision.

For the type 1 connectionless service, LLC uses two of the primitives. The Request primitive is used for sending a frame and the Indication primitive is used for indicating that a frame has arrived. The type 2 connection-oriented service uses the four primitives substantially. The primitives are divided into five groups, each group performing a different function. The first group of primitives has a prefix of L_CONNECT. The primitives in this group are L_CONNECT.request, L_CONNECT.indication, L_CONNECT.response and L_CONNECT.confirm. This group is used for establishing a connection with a remote host. The second group has a prefix of L_DISCONNECT and is used for releasing a connection. The third group has a prefix of L_DATA_CONNECT and is used for transferring data. The fourth group has a prefix of L_RESET and is used for resetting the connection after a serious error has been detected. The fifth group has a prefix

of L_CONNECTION_FLOWCONTROL and is used to regulate the flow of information between the network layer and the data link layer.

The LLC frame structure is similar to that of HDLC:

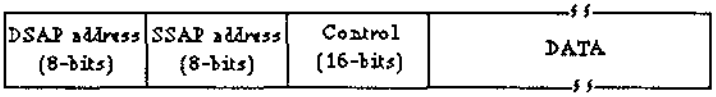


Figure 4.3 - The IEEE 802 LLC Frame Format

DSAP and SSAP are Destination Service Access Point and Source Service Access Point respectively. Each layer in the OSI model provides a service to the layer above it. A service access point is the point where a layer n+1 can access the services offered by layer n below. When a connection is set up, the network layer pushes a service request through the Source Service Access Point (SSAP) and the request emerges at the remote end at the Destination Service Access Point (DSAP). The SSAP-DSAP pair identifies a connection and is used for the duration of the connection.

The control field is similar to the control field in the LAPB and HDLC protocols (figure 4.1). However, there are some differences between LLC and the others. Firstly, LLC allows a 16-bit control field. This allows the inclusion of a 7-bit sequence number. Secondly, the LLC protocol only supports asynchronous balanced mode supporting peer-to-peer communication, not computers polling terminals. Thirdly, two new frame types have been included in addition to the Information, Supervisory and Unnumbered frame types. These are the Exchange Identification (XID) frame for communicating information between LLC peers about service types and window sizes, and the TEST frame for requesting the other side to generate a response to test the connection.

The IEEE 802 standard also defines the interface between the LLC sublayer and the MAC sublayer below. Three primitives are used for a connectionless interface, namely MA_DATA.request, MA_DATA.indication and MA_DATA.confirm.

4.3.2 IEEE 802.3 (CSMA/CD)

The Carrier Sense Multiple Access with Collision Detection (CSMA/CD) technique is used in standard Ethernet implementations. It exists within implementations as the lower data link layer (MAC sublayer).

CSMA/CD works as follows. If a transceiver detects a collision it briefly jams the ether using a special signal to warn other transceivers not to transmit anything. Then it can either try again immediately, try again after a random period of time or adopt a combination of both approaches. The first option of trying again immediately is not usually suitable because another transceiver may do exactly the same thing, therefore leading to another collision. The second option of waiting for a random length of time is the standard CSMA/CD technique. In this implementation, transceivers must wait until the delay has elapsed before transmitting, even if the ether is free, so it is not entirely efficient. Some implementations elect to use a combination of trying again immediately and waiting for a random length of time before trying again. This is achieved by sometimes trying again immediately, and at other times waiting for a random length of time. The probability used to decide which method to use must be calculated carefully to avoid the problem in the first option.

A break in the coaxial cable will appear as if the cable is unterminated. Therefore, frames that are normally absorbed by a terminator will be reflected back along the cable, causing collisions. Therefore if all frames from all transceivers appear to be colliding, this is indicative of a cable fault. It is also possible for a faulty transceiver to jam the cable continuously.

Ethernet

Ethernet is an example of a CSMA/CD local area network. Like the addressing provided by the IP layer in TCP/IP, Ethernet has its own addressing. In Ethernet for example, it was the intention of the designers that no two machines would have the same Ethernet address. They also wished to relieve users of the task of creating unique addresses for their machines, and so adopted a strategy of assigning Ethernet controllers with addresses at the factory. It is necessary therefore for all Ethernet manufacturers to confer with a central authority about the addressing for the controllers.

An Ethernet frame has the following format:

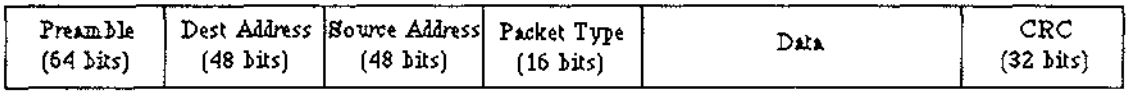


Figure 4.4 - Ethernet Frame Format

A frame contains a 14-octet header containing a destination Ethernet address, a source Ethernet address and a type code. Each machine's controller listens only for frames destined for itself. The type code allows several different protocol suites such as TCP/IP, DECnet etc, to use the Ethernet. Each suite has an identifier which is copied into the type code field. TCP/IP's use of Ethernet is discussed later.

Following the header is the datagram specific to the protocol suite. In TCP/IP for example, the data field will hold an IP datagram. Finally, there is a checksum calculated by the Ethernet controller.

When the Ethernet frame is received by the destination Ethernet controller, the Ethernet interface proceeds to decapsulate the datagrams from within the frame. The type code is checked and the datagram is passed up to the appropriate family of protocols for further decapsulation if necessary.

4.4 IP over Ethernet

The procedure for transmitting IP datagrams over Ethernet is now discussed. IP datagrams may be encapsulated and transmitted within Ethernet frames, as discussed earlier. The Ethernet frame header contains a number of fields including the destination Ethernet address, the source Ethernet address and the type of protocol it is supporting.

4.4.1 Address Resolution Protocol

When an IP datagram is encapsulated inside an Ethernet frame, the destination field of the latter must be filled in with a 48-bit Ethernet address. In other words, there must be a way of finding out the Ethernet address of the host identified by the 32-bit Internet address. The mapping of the 32-bit Internet address to a 48-bit Ethernet address is done using the Address Resolution Protocol (ARP).

ARP is not a member of the TCP/IP suite. The protocol works as follows. When given an IP datagram containing a 32-bit Internet address such as 130.123.3.10, the system accesses a lookup table containing a list of all hosts on the local network along with their Internet addresses and Ethernet addresses. This list is searched for the appropriate Internet address, and if it is found then its associated Ethernet address is copied into the destination field of the Ethernet frame header. If the Internet address is not found, then the protocol broadcasts an ARP request. Essentially an ARP request says "I need an Ethernet address for 130.123.3.10". All hosts must listen to ARP requests, and the host to which the Internet address belongs is expected to reply with its 48-bit Ethernet address. The requesting host receives this information and uses it to send future frames directly to the destination host. Although hosts which do not have an Internet address as that requested in the Address Resolution Protocol are expected to ignore the ARP requests, some hosts may use the ARP request to update their own knowledge about other hosts on the network, although they are not directly involved.

4.4.2 Ethernet Frame Contents for IP

As discussed in section 4.3.2, a type field is used to identify the protocol family which is using the Ethernet to transport its datagrams. For TCP/IP, this field must contain the hexadecimal value 0800.

The data field contains the IP datagram consisting of the IP header followed immediately by the data. The data field has a minimum length of 46 octets, and therefore it should be null padded if necessary. Note that this padding is not considered to be a part of the IP datagram and is not included in the total length field of the IP header.

The data field has a maximum length of 1500 octets. This means that the maximum size of an IP datagram is 1500 octets. Implementations are encouraged to support full-length packets and gateways on the Internet must be able to accept full-length packets and fragment them if necessary. A system which cannot accept full-length packets discourages others from sending them by using the TCP Maximum Segment Size Option.

4.4.3 TCP Maximum Segment Size Option

During the connection establishment phase, TCP provides an option to indicate the maximum size TCP segment that can be accepted on that connection. This announcement is sent from the receiving host to the sending host and effectively says "I can accept TCP segments up to size X". The value of X may be larger or smaller than the default value. Similarly, the sending host sends a maximum segment size which is suitable for itself. The smaller of the two values is then used for the remainder of the connection.

The TCP Maximum Segment Size counts only the data octets in the segment. It does not count the TCP header or the IP header. The TCP Maximum Segment Size is thus defined as being the IP maximum datagram size minus 40 octets. For example, the default IP maximum datagram size is 576 octets, so the TCP maximum segment size is 536 octets.

4.4.4 Datagram Fragmentation

The default IP maximum datagram size is 576 octets. This is considered to be a 'safe' number and it is supported by all implementations. Hosts generally keep this in mind when sending datagrams to another host which is on a different Ethernet network. A host may however elect to send smaller datagrams to prevent the datagrams from being fragmented at intermediate gateways. Fragmentation is a process where a large datagram is split into two or more smaller pieces because at an intermediate step such as another network, it becomes too big to handle. For example, assume that during the connection establishment phase two hosts on separate Ethernet networks agree on a maximum segment size of 1500 octets. However, it is also necessary for the Ethernet packets to pass through an intermediate

network which cannot handle packets larger than 1000 octets. It then becomes necessary for the Ethernet packets to be fragmented into two or more smaller pieces, pass through the intermediate network, and then be reassembled after passing through. The IP header contains fields to indicate that its datagram has been fragmented, and contains enough information for the datagrams to be reassembled in the correct order.

It is fairly common for implementations to use 576 octet datagrams whenever it is uncertain whether intermediate networks can cope with larger datagrams. This strategy is adopted because the reassembly algorithms in some intermediate networks have been found to be faulty. The maximum size datagram that all hosts must be able to accept, or reassemble from fragments is 576 octets. Hosts may accept larger fragments and assemble these into larger datagrams as they wish.

It is important that once the TCP Maximum Segment Size has been agreed upon at connection establishment, hosts must not defer from the agreed value. Similarly, hosts must not send datagrams larger than 576 octets unless they have specific knowledge that the destination host can handle larger datagrams.

4.5 X.25 over CSMA/CD

The JNT Ethernet Advisory Group is a technical committee responsible for defining the implementation details of connection-oriented protocols on CSMA/CD local area networks. A report produced in 1985 provided the details for implementing X.25 over CSMA/CD LANs.

4.5.1 Data Link Layer Management

The X.25 over CSMA/CD implementation must support both connectionless (type 1) and connection-oriented (type 2) services as described in section 4.3.1. An implementation can choose to omit the generating of TEST and Exchange Identification (XID) frames but must provide a means of responding to these. However, the XID frame generation feature is strongly recommended to support link window size negotiation as described below.

A LAN station that wishes to set up a virtual call with another LAN station must initiate a link establishment phase if a link does not already exist. A LAN station may support many link connections. An inactive link is one which after establishment has never sustained any X.25 virtual calls, or has sustained virtual calls but all of these have been cleared. For reasons of efficiency, inactive links are disconnected.

Links are disconnected as follows. A LAN station maintains a count of established X.25 virtual calls. This count is initialised to zero following the restart procedure, and an internal timer is started. When an X.25 CALL CONNECTED packet is received or a CALL ACCEPT packet is sent in response to a CALL REQUEST or INCOMING CALL respectively, the timer stopped if the count is zero and the count is incremented. When a CLEAR CONFIRMATION is sent or received in response to a CLEAR REQUEST or CLEAR INDICATION, the count is decremented and the timer is restarted if the count becomes zero. If a RESTART REQUEST is sent or a RESTART INDICATION is received and the count is non-zero, then the count is set to zero and the timer is re-started. If the timer expires, the link is disconnected.

A number of link parameters may be used to configure the nature of the link:

a) Maximum number of outstanding Protocol Data Units (PDU) - A PDU is an LLC frame (figure 4.3). Once a data link connection has been made an XID frame can be used to

determine the receive window sizes for each end-point of the link. The XID frame exchange comprises of a command and a response. If a station cannot support a window size of 127 then it must use the XID command to inform the remote LAN station of the maximum window size that it can support. Each station is responsible for generating an XID command if necessary, and should not rely on the XID response to indicate its maximum window size. The window size can be changed at any time by either station issuing an XID command. XID command frames which are received before a link is established are ignored. When a link is established, the default window size of 127 is assumed until an XID command frame is issued or received.

b) Maximum number of octets in an PDU - The Medium Access Control (MAC) sublayer places a restriction of 1518 octets per frame. The presence of header information in the frame therefore restricts the maximum number of octets in a PDU to 1030 octets.

The minimum number of octets in an in a PDU is 262. This is because the maximum length of the facility field in a CALL REQUEST packet is 109 octets, and the maximum size of the user data field of the X.25 fast select CALL REQUEST packet is 128, therefore requiring an information field of 256 octets. With the addition of packet headers and addressing information, the minimum becomes 262 octets.

c) Acknowledgement Timer - This is a timer which measures the interval during which an acknowledgement to one or more outstanding PDUs can be expected by the LLC. The suggested value for this timer is 0.3 seconds but may be set to an alternate value.

d) P bit Timer - This is a timer which measures the interval during which the LLC can expect to receive a PDU with the final (F) bit set in response to a type 2 command with the poll (P) bit set. The suggested value for this timer is 0.1 seconds.

e) Reject Timer - This is a timer which measures the interval during which the LLC can expect to receive a reply to a rejected PDU. The suggested value for this timer is 0.1 seconds.

f) Busy State Timer - This is a timer which measures the interval during which the LLC can wait for an indication following the clearance of a busy condition at the remote LLC. The suggested value for this timer is 1 second.

g) Maximum Number of Transmissions - This is the maximum number of times that a PDU is sent following the expiration of the Acknowledgement timer, the P bit timer or the Reject timer. The suggested value for this is 10 transmissions.

Addressing at the data link layer consists of a pair of data link level addresses for the source and destination. Each address consists of a concatenation of a physical address, a MAC address such as an Ethernet address (figure 4.4) and an LLC address (figure 4.3).

4.5.2 Network Layer Management

The X.25 over CSMA/CD implementation is designed to work in a direct DTE-to-DTE connection, therefore avoiding an intermediate PSDN. This eliminates the need for standard X.25 addressing so all of the DTE address fields in the X.25 CALL REQUEST, INCOMING CALL, CALL ACCEPT, CALL CONNECTED, CLEAR REQUEST, CLEAR INDICATION and CLEAR CONFIRMATION packets should be set to null. The network level addressing will then consist of a Network Service Access Point (NSAP) address which is carried in the called and calling address fields. The NSAP address provides a Network Service Access Point for the transport layer above.

The following procedures should be observed in the implementation:

a) The Address Length and the Facility Length fields are necessary in the X.25 CALL ACCEPT packets even if the address and facility information is not provided.

b) The Diagnostic Code field in the RESTART REQUEST, RESET REQUEST and CLEAR REQUEST packets must be provided even if it indicates that there is no additional information.

c) A data packet which has its M bit set and D bit cleared but contains less data than the maximum cannot be transmitted.

d) One of the DTEs must act as a DCE for logical channel selection during the call set-up phase and for resolving call collisions.

e) A DTE must be able to accept RESTART INDICATION, RESET INDICATION and CLEAR INDICATION packets if the cause field of these packets indicate that they were generated by the remote DTE.

f) DIAGNOSTIC packets are not used.

g) The window and packet sizes should be negotiable. The initial values for these are 2 and 128 respectively. Modulo-8 packet sequencing is used but modulo-128 sequencing is reserved for further investigation.

The implementation makes use of a number of optional user facilities:

- Throughput class negotiation
- Transit delay detection and indication
- Fast select
- Fast select acceptance
- Called address extension
- Calling address extension
- End to end transit delay negotiation
- Minimum throughput class negotiation
- Reverse charging

The On-line facility registration facility and the Packet re-transmission facility are not used in the implementation.

4.6 Serial Line IP (SLIP)

Serial Line IP (SLIP) is a protocol for the transmission of IP datagrams over serial lines. It is currently the de-facto standard, commonly used for point-to-point serial connections running TCP/IP.

SLIP originated in the early 1980's in the 3COM UNET TCP/IP implementation. It became a popular and reliable way to connect TCP/IP hosts and routers over serial lines after Adams, 1984, implemented SLIP for 4.2 Berkerly UNIX and Sun Microsystems.

SLIP is a packet framing protocol, which defines a sequence of characters as a framework to hold IP datagrams on a serial line. SLIP is a simple protocol which does not provide any addressing, packet type identification, error detection/correction, or data compression. Therefore SLIP is very easy to implement.

4.6.1 SLIP Protocol Description

SLIP defines two special characters, END (decimal 192) and ESC (decimal 219). The latter must not be confused with the ASCII escape character. SLIP sends an IP datagram by beginning and ending the transmission with an END character. If a character in the datagram should be an END, a 2-octet sequence consisting of ESC and decimal 221 is sent instead.

SLIP does not have a standardized maximum packet size. The Berkeley SLIP drivers have been written for a maximum packet size of 1006 octets including the IP and the transport protocol headers (not including framing characters) and so this has been generally accepted as the de-facto maximum packet size for new SLIP implementations.

4.6.2 Deficiencies

Because SLIP is a simple protocol which was developed a number of years ago, certain design issues were overlooked which are nowadays considered highly desirable, if not necessary.

a) Addressing - SLIP does not provide a means for hosts to communicate addressing information. This is necessary for routing purposes.

b) Type Identification - SLIP does not provide a type field and so only one protocol from the Internet suite may be used at any one time. If SLIP did provide such a field, then two multi-protocol hosts would be able to communicate across the serial line using more than one protocol.

c) Error Detection/Correction - SLIP does not provide error detection and correction. Although the IP level and the TCP/UDP levels provide error detection with the use of checksums, some applications such as NFS may elect to ignore the checksums and assume that the network has taken care of damaged packets. SLIP is run over a serial line at low speeds (about 2400 baud) which makes retransmission expensive. If SLIP provided some form of simple error checking, then an increase in efficiency could be achieved.

d) Data Compression - It is usually desirable to compress a stream of data before it is transmitted to increase throughput. This is especially the case over serial lines which tend to be the slowest mediums. SLIP does not provide such a facility. The streams of packets in a TCP connection usually have few altered fields in the headers, so an example of a simple compression algorithm would be one that just sent the altered parts of the header instead of the complete header.

Research is being done by various groups to address these problems.

4.7 Point-to-Point Protocol (PPP)

The Point-to-Point Protocol (PPP) provides a method for transmitting datagrams over serial point-to-point links. It was developed as a replacement for the SLIP protocol described in section 4.6 but is not widely used. One reason for the lacking in the use of PPP is because there is no standard data encapsulation protocol. Although non-standard protocols exist which perform this task, none have been agreed upon as being an Internet standard. The PPP attempts to remedy this problem by presenting a suitable encapsulation protocol.

Relatively few hosts are connected using this kind of link despite the fact that almost all hosts can support point-to-point links, and also that point-to-point links are one of the oldest methods of data communications.

PPP consists of three parts:

a) A method for encapsulating datagrams over serial links. The PPP encapsulation protocol uses HDLC as a basis for encapsulating datagrams over point-to-point links. PPP uses asynchronous or synchronous duplex circuits, either dedicated or circuit switched.

b) A Link Control Protocol (LCP) to establish, configure and test the link connection.

c) A family of Network Control Protocols (NCP) of which the IP Control Protocol (IPCP) is a member, for establishing and configuring different network-layer protocols. PPP allows the use of more than one network control protocol simultaneously.

4.7.1 PPP Layering

The Physical Layer

PPP is capable of operating across any DTE-DCE interface. The only requirement is that a duplex circuit must be provided, this being either asynchronous or synchronous, dedicated or switched.

The only restrictions regarding the transmission rate are those imposed by the DTE/DCE interface. PPP does not require the use of modem control signals, such as Request To Send

(RTS), Clear To Send (CTS), Data Carrier Detect (DCD), and Data Terminal Ready (DTR), although provisions are made for these which if used, provide greater functionality.

The Data Link Layer

The standard PPP frame structure is based on the HDLC frame structure (figure 2.2). However, it contains an additional protocol field:

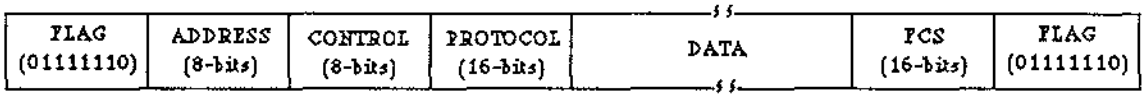


Figure 4.5 - Standard PPP frame structure

The 2-octet Protocol field identifies the protocol encapsulated in the data field in the frame. IP is assigned the value 8021 (hex). LCP packets, which are also transferred using the Information field, are assigned a hexadecimal value having a format of c???.

The LCP can negotiate modifications to this frame structure if required. In addition to the fields described, there are start/stop bits included for asynchronous links.

4.7.2 Link Control Protocol (LCP)

The LCP is responsible for establishing, configuring, maintaining and terminating point-to-point links. There are four phases:

a) Link establishment/Configuration negotiation - The link establishment phase involves establishing a PPP link with the remote host by exchanging link establishment packets. This phase is only concerned with the PPP link to the remote host, not the IP connection in the network layer.

b) Link quality determination - This is an optional phase which may follow the link establishment phase. In this phase, the link is tested to ensure that the link quality is sufficient to begin exchanging IP level datagrams.

c) Network Control Protocol configuration negotiation - When the PPP link is established and the link's quality has been tested, the network control protocols such as IPCP establish connections with the remote host. The network-level connections can be brought down at any time by either the LCP or the particular NCP. If LCP closes the underlying PPP link, then all NCPs are informed so that appropriate actions can be taken.

d) Link termination - LCP may terminate the links at any time. This is usually done when a user wishes to close the session, but it may also happen in the event of a loss of the carrier signal or timeouts.

LCP Packets

There are three classes of LCP packets:

a) Link Establishment Packets - These are used to establish and to configure a PPP link. Examples are Configure-Request, Configure-Ack, Configure-Nak, Configure-Reject.

b) Link Termination Packets - These are used to terminate a PPP link. Examples are Terminate-Request and Terminate-Ack.

c) Link Maintenance Packets - These are used to manage and debug a PPP link. Examples are Code-Reject, Protocol-Reject, Echo-Request, Echo-Reply and Discard-Request.

LCP is based upon a finite set of events and states. A link can be in one of a number of states at any time. The move from one state to another is caused by one of a number of possible events and events are caused by one of a number of actions.

LCP Actions and Events

a) Send-Configure-Request - This action causes a Configure-Request packet to be sent along with configuration options, indicating the desire to open a LCP connection with the remote host. A timer is also started which will cause a timeout event to occur if a response has not been received within a reasonable length of time. The action will generate a Receive-Configure-Request event at the remote host.

b) Send-Configure-Ack - This action causes a Configure-Ack packet to be sent, acknowledging the receipt of a Configure-Request packet with an acceptable set of configuration options. A Receive-Configure-Ack event is generated at the remote host.

c) Send-Configure-Nak - This action causes a Configure-Nak or a Configure-Reject packet to be sent, acknowledging the receipt of a Configure-Request packet with an unacceptable set of configuration options. The Configuration-Nak packet can be used to suggest acceptable alternative configuration options. The Configuration-Reject packet can be used to refuse the connection request because the configuration options are not recognized or not implemented. This action generates a Receive-Configure-Nak event on the remote host.

d) Send-Terminate-Request - This action causes a Terminate-Request to be sent, indicating the desire to terminate the LCP connection. A timer is also started, which will cause a timeout event to occur if a response has not been received within a reasonable length of time. This action will generate a Receive-Terminate-Request event at the remote host.

e) Send-Terminate-Ack - This action causes a Terminate-Ack to be sent. This is used either to indicate that a Terminate-Request packet has been received, or to confirm that a LCP connection has been closed. A Receive-Terminate-Ack event will be generated at the remote host.

f) Send-Code-Reject - This action causes a Code-Reject to be sent to indicate the receipt of an unknown packet type. This error is fatal and causes the LCP connection to the peer to be closed. The Receive-Code-Reject event is generated at the remote host.

g) Send-Echo-Reply - This action causes an Echo-Reply packet to be sent in response to an incoming Echo-Request packet. An incoming Echo-Request packet is identified by a Receive-Echo-Request event.

Other LCP events are:

a) Active-Open - This event indicates that a user wishes to start a session. The machine should therefore initiate the link establishment/configuration negotiation phase by exchanging configuration packets.

b) Passive-Open - This event waits for its peer to start the configuration negotiation. In other words, an Active-Open event should have occurred at the peer.

c) Close - This event indicates that the session is to be closed. When this event occurs, the LCP should begin closing the connection.

d) Timeout - This event indicates that the timer has expired. The timer is used to time out transmissions of Configure-Request and Terminate-Request packets, which results in their being sent again.The timer defaults to 3 seconds.

e) Receive-Unknown-Code - This event occurs on receipt of an unrecognizable packet from the peer. The usual action is to respond with a Receive-Code-Reject packet.

f) Physical-Layer-Down - This event occurs when the physical layer indicates that it is not operational.

LCP events cause the link to move from one state to another. The current state determines the next state of the link, depending on the events which occur and the actions which are performed.

LCP Configuration Options

LCP Configuration options are negotiated at the PPP link establishment phase. Configuration options are transported across the PPP link inside LCP packets. Examples of options which may be negotiated are Maximum Receive Unit, Link Quality Monitoring, Protocol Field Compression and Address & Control Field Compression.

4.7.3 IP Control Protocol (IPCP)

The IP Control Protocol (IPCP) is a Network Control Protocol for the transmission of IP datagrams over the PPP link. The IPCP is responsible for configuring, establishing and terminating the IP connection between two PPP links. This is done with the exchange of packets after the PPP link has been successfully established and tested by the LCP. IPCP packets which are received before the LCP has completed the configuration negotiations and testing are discarded.

IPCP packets transported inside the data field in a PPP frame (figure 4.4) in the same way as LCP packets. However IPCP packets are distinguished from LCP packets by assigning the value 8021 (hex) in the Protocol field. There are seven kinds of IPCP packets.

When IPCP has established a connection with the remote host, IP datagrams may be exchanged across the link. IP datagrams are transported inside the data field in a PPP frame. IP datagrams are distinguished from LCP packets by assigning the value 0021 (hex) in the Protocol field. Exactly one IP datagram may be transported within the data field at a time. The maximum size of the datagram must be the same as the maximum size of the data field. Larger datagrams may be fragmented, so to prevent this the TCP Maximum Segment Size option is used to discourage remote hosts from sending oversized datagrams.

IPCP Configuration options are negotiated at the conclusion of the LCP link establishment and testing phase. Configuration options are transported inside IPCP packets. There are two main kinds of Configuration options:

a) IP-Addresses - This configuration option provides a way of negotiating the IP addresses to be used at each end of the link. By default, no IP addresses are assigned to either end. The source and destination IP addresses must be specified, but it is possible to get the remote machine to provide the address by setting the address field to zero.

b) Compression-Type - This configuration option provides a way of negotiating the use of a compression protocol. By default, compression is not enabled.

4.8 ISO Development Environment (ISODE)

The ISO Development Environment (ISODE) is software which provides an ISO transport level protocol interface on top of TCP/IP. ISODE was designed to allow researchers to experiment with ISO's higher-level OSI protocols on the Internet which supports the lower levels of the OSI suite (section 1.4). Although the Internet community pioneered computer communications in the early 1970's and were leaders in the field, they did not take an active role in OSI research. The ISODE was developed in the hope that the OSI protocols will be adopted in the Internet and that it would provide a useful transition path between the Internet and OSI communities. ISODE can support several different underlying network services including the underlying levels of OSI itself. It is not limited to using TCP/IP.

4.8.1 ISODE Communities

The ISODE Transport Switch is an abstraction which has the goal of hiding the complexities of the 'real world' of open systems architecture from the user of the transport service.

In ISODE, an OSI community is a collection of hosts which share a common TS-Stack. A TS-Stack is a combination of transport protocols and network services used to provide end-to-end transport services. There are several communities using OSI at present. These are the International X.25 community (int-x25), the JANET X.25 network in the United Kingdom (janet), the Internet (internet), and the TCP loopback address (localTCP).

The first task ISODE performs is to determine what communities a site belongs to. A run-time variable called ts_communities keeps track of the communities that a host can directly or indirectly reach. The ts_communities variable defaults to the four communities mentioned above. Therefore, if a host belongs to the International X.25 community but not to the JANET community, the following line is added to the ISODE configuration file:

ts_communities: int-x25

If a host belongs to the International X.25 community as well as the Internet community, the line in the ISODE configuration file will be:

ts_communities: int-x25 internet

In the event that a host belonging to two communities wishes to communicate with another host which also belongs to the same two communities, then the first entry in the ts_communities variable is tried first. In the above example, the International X.25 community is tried first and then the Internet community.

4.8.2 Transport Service Bridges

A Transport Service Bridge (TS-bridge) is used to allow two hosts to communicate if they do not have any communities in common. The bridge is a third host which shares a community with the two other hosts. One of the two hosts is the client and the other is the server. The server host is the one connected to a community which the other host is trying to access via the TS-bridge. Because of the different functionality, client hosts have some ISODE variables declared in a different manner to server hosts.

When configuring client hosts the user must examine the ts_communities variable to find out which communities the host does not belong to. Then an entry must be added to a variable called tsb_communities which identifies a particular TS-bridge through which the community can be reached. The entry consists of the name of the community and the transport address of the TS-bridge. The transport address is made up of a DTE number and a process identification (PID) number.

A community usually contains a TS-bridge which connects that community to other communities. Therefore, server programs must know the address of the TS-bridge to listen to for incoming requests. A runtime variable called tsb_default_address is used to define the transport address.

4.8.3 ISO-TP0 Bridge between TCP and X.25

The ISO-TP0 Bridge is a host which offers gateway services between X.25 hosts and TCP/IP hosts running higher-layer ISO protocols, allowing either to initiate connections, exchange data and terminate connections using the TP0 protocol.

The ISO Transport Protocol (TP) implements the ISO Transport Service which provides a reliable, packet-stream to its users. There are five classes of the ISO Transport Protocol. A class is chosen depending on the kind of underlying network service that it offers. For

example, Transport Class 0 (TP0) is used when the network service being offered is connection-oriented and error-detecting. TP0 is therefore a relatively simple protocol because the underlying network service provides such facilities.

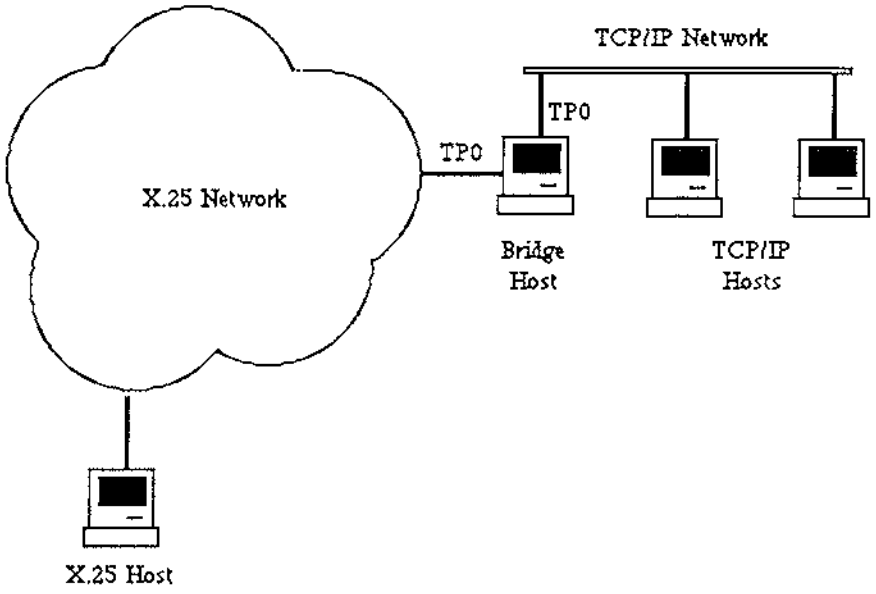


Figure 4.6 - ISODE Bridge

The TP0 bridge host acts as a protocol converter. It receives TP0 packets using TCP from the TCP host, converts them into TP0 packets using X.25 and passes these on to the X.25 host. It also does the opposite, converting TP0 packets using X.25 into TP0 packets using TCP and passes these on to the TCP host. The usefulness of this bridge is obvious since it allows a host on an IP based internet to communicate with hosts on X.25 based networks, providing all hosts involved are running higher layer ISO protocols.

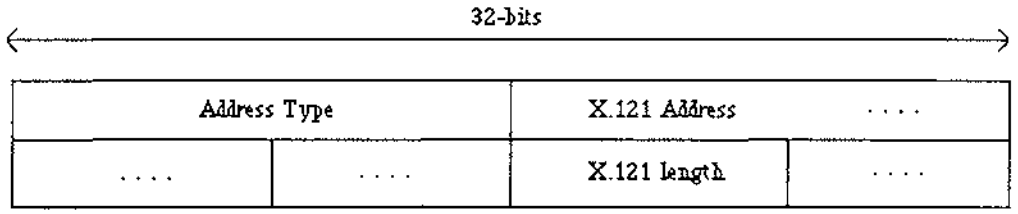
A session consists of the connection establishment phase, data transfer phase and the connection termination phase. A connection establishment phase is initiated by the TCP host. There are two kinds of connections, these being incoming calls and outgoing calls. The TCP host begins the connection establishment procedure by establishing a connection to the reserved port 146 on the TP0 bridge. It then requests a particular type of connection by sending one octet which may be 0 (illegal), 1 (connect to a particular X.25 host), 2 (listen for incoming X.25 connections) or 3 to 255 (reserved).

If the initiating host sends a 1 (connect to a particular X.25 host) to the TP0 bridge, it must also send the X.25 address to connect to. The TP0 bridge then uses this address to call the

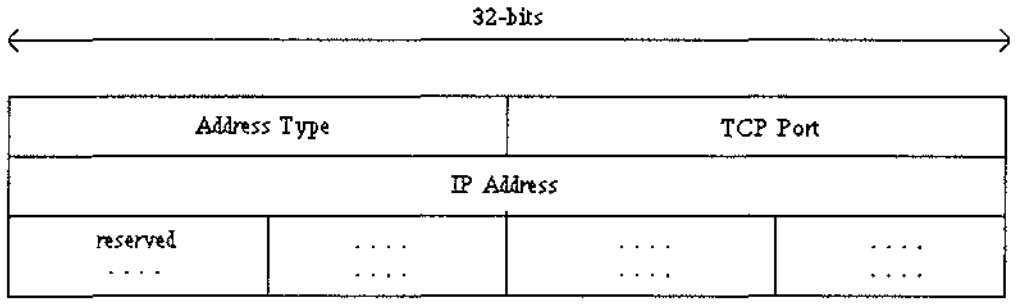
specific X.25 host. If the call succeeds, the connection establishment phase is successful and the data transfer phase may begin. If the call fails, the TP0 bridge closes the TCP connection.

If the initiating host sends a 2 (listen for incoming X.25 connections) to the TP0 bridge, it must also send an X.25 address which is a subaddress of the TP0 bridge's X.25 address on which to listen for incoming connections. The initiating host also sends an IP address and a TCP port number which will service incoming calls for the X.25 address. The TP0 bridge then listens for incoming connections on the specified X.25 subaddress. If an incoming call is received, a TCP connection is initiated to the specified IP address. If the connection is established then data transfer may begin, otherwise the connection to the initiating TCP host is closed and the incoming call is refused. If the initiating host terminates the connection, the TP0 bridge assumes that the initiating host wishes to discontinue the listening function and so incoming calls are no longer accepted.

The addressing formats are as follows:



(a) X.25 Address Format



(b) TCP/IP Address Format

Figure 4.7 - TP0 Addressing Formats

X.25 address encoding contains three fields:

a) Address type (2 octets) - This is a binary-encoded value in network order. The value 3 is used to indicate X.25 addressing of this format.

b) X.121 address (16 octets) - This is the ascii encoded value of the X.121 address.

c) Address length (1 octet) - This is a binary-encoded value in network order indicating the number of meaningful octets in the X.121 address.

The TCP/IP address encoding has also contains three fields:

a) Address type (2 octets) - This is a binary-encoded value in network order. The value 2 is used to indicate Internet addressing.

b) TCP Port (2 octets) - This is a binary-encoded value in network order.

c) IP address (4 octets) - This is a binary-encoded value in network order.

The data transfer phase may begin as soon as network connections between the TCP host and the TP0 bridge, and between the TP0 bridge and the X.25 host have been established. The data transfer phase consists of the TP0 bridge reading Transport Protocol Data Units (TPDU) from one network connection and writing them to the other. When one network indicates a disconnect to the bridge, the bridge disconnects the other network unless both networks simultaneously request disconnects to the bridge, in which case the bridge does not take any action.

Because TP0 is built on an underlying network service that is error-detecting, the protocol does not attempt to report any network generated errors which are recoverable. Instead, only errors generated by TP0 are usually reported.

Security is handled locally. A major concern is that any TCP host which does not have restrictions imposed on it can establish a connection with the TP0 bridge and make valid but unauthorized connections with X.25 hosts by providing an authentic X.25 address.

4.9 VMS Packet Switch Interface (PSI)

Digital's packet switching products allow Digital users to connect to an X.25 PSDN. This is especially useful for Decnet users because the Packet Switch Interface (PSI) manages the PSDN virtual circuits on their behalf. During the communication the PSDN is not seen by the users. This implementation of Decnet over X.25 is called Data Link Mapping.

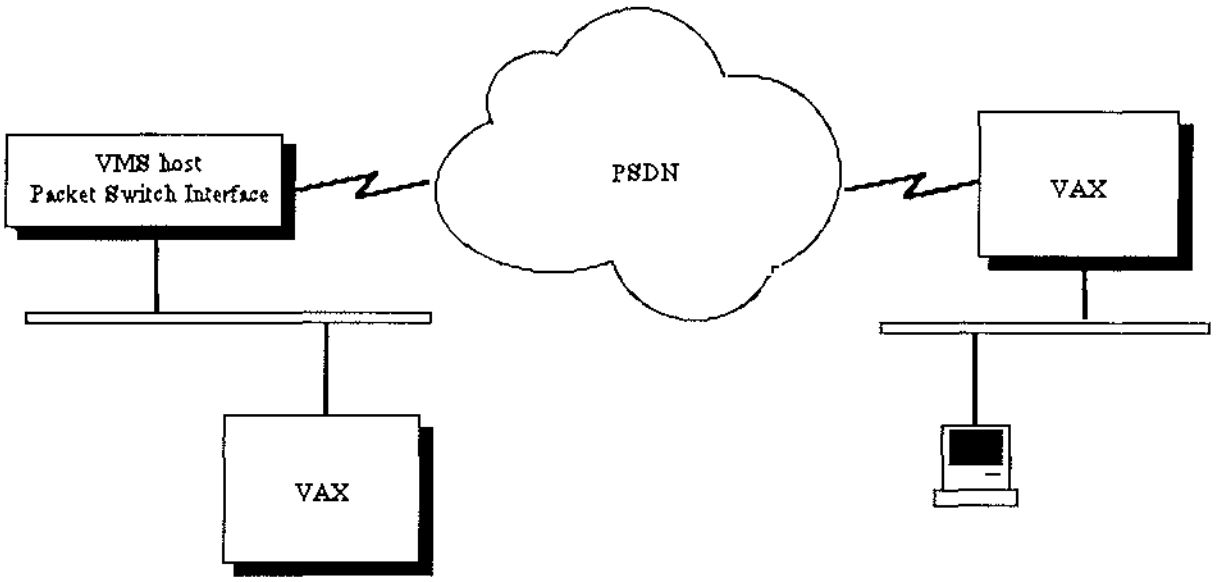


Figure 4.8 - Topology of VMS PSI Data Link Mapping

The Packet Switch Interface is a server which runs on a VMS host and enables a hosts on a LAN such as CSMA/CD to communicate with a wide-area network.

4.10 IP over X.25

So far this chapter has discussed various protocols at the data link layer and how these provide underlying services to the network layer above. This thesis studies an implementation of X.25 over TCP/IP so this section investigates how the opposite of TCP/IP over X.25 can be used.

A PSDN requires an interface such as X.25 to define the communication between the DTE and the DCE. In a configuration where a user's terminal is a packet-mode DTE and is connected to the PSDN, the user's data is encapsulated inside X.25 data packets.

Alternatively, if the user's machine is located on a CSMA/CD LAN running TCP/IP and the LAN is connected to the PSDN by a gateway, the IP datagrams will get encapsulated into X.25 packets. The X.25 packets will be transmitted through the PSDN and the IP datagrams will be extracted from the X.25 packets on arrival at the remote end. In this case the remote end will be another LAN running TCP/IP.

TCP/IP packets may be transported over X.25 if the remote end is running a different set of protocols such as OSI instead of TCP/IP. A bridging computer is necessary for this.

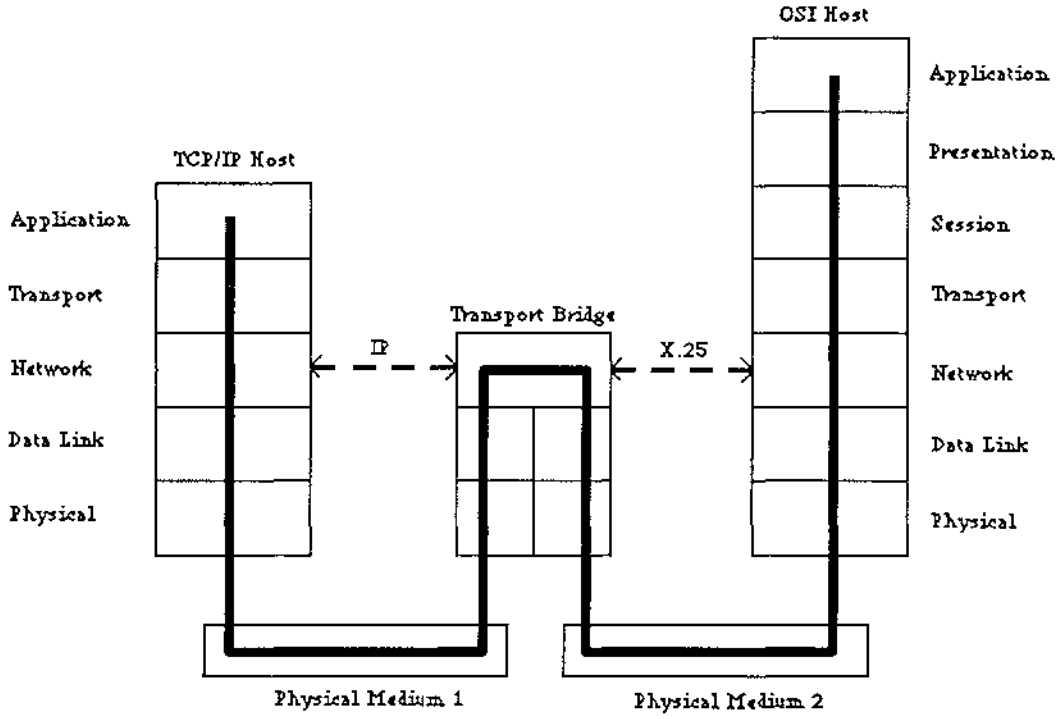


Figure 4.9 - Transport Service Bridge

The concept of using a bridge between TCP/IP and X.25 was discussed in section 4.8. A bridge is another way to transport TCP/IP packets to a dissimilar network. The bridge is a host which connects two hosts on different networks and forwards packets among them. The bridge in figure 4.9 connects a host on a TCP/IP network to a host on an OSI network. The application layer on the TCP/IP host passes data to the transport layer. The transport layer encapsulates this data into TCP packets which are then sent to the network layer and encapsulated into IP packets. The IP packets are passed down to the data link layer where they proceed through the LLC and MAC sublayers and given appropriate headers. The resulting frame is then transmitted to the bridge host through physical medium 1, where on

arrival the frame is decapsulated as it moves up through the layers on the left-hand side of the bridge. When the network layer of the bridge is reached, the bridge will be left with an IP datagram. The information contained in the IP datagram is encapsulated into an X.25 packet which is passed down the right-hand side of the bridge host. The X.25 packet is encapsulated into an appropriate data link protocol such as LAPB and the LAPB frame is sent to the physical layer where it is transmitted to the OSI host. Finally, the OSI host begins decapsulating the frame as it moves up through the OSI levels.

"XI" is a BSD UNIX network interface between the X.25 and IP protocols. It provides access to X.25 packet switching networks through the socket abstraction in UNIX (and therefore TCP/IP). The interface functions as a finite state machine where each X.25 virtual circuit switches between the states CLEAR, CALLWAIT, CLEARWAIT, RESETWAIT and ESTABLISHED in the process of transmitting and receiving IP datagrams. X.25 virtual circuits are opened to remote hosts when IP datagrams for those hosts arrive. The virtual circuits are kept open as long as possible, not opened and closed for each individual datagram as this would be very expensive to do. Because TCP/IP is built on a connectionless service, each datagram is treated individually. They are not matched with any particular X.25 connection.

A count down timer is associated with each active virtual circuit which count down while the virtual circuit remains idle. When the timer reaches zero, the virtual circuit is closed. Therefore if there is a long delay between IP datagram n and n+1 the virtual circuit may be closed after datagram n is transmitted, in which case when datagram n+1 arrives a new virtual circuit must be made.

CHAPTER 5

IMPLEMENTATION OF X.25 OVER TCP/IP

5.1 Introduction

The development of X.25 over TCP/IP was stimulated by the networking requirements of the Department of Computer Science at Massey University. The department has two Ethernet local-area networks running TCP/IP. X.25 is not supported but is available on the Pyramid computer at the Massey University Computer Center. The reason for undertaking the project was because the end-result would allow users within the department to develop X.25 software on IP workstations.

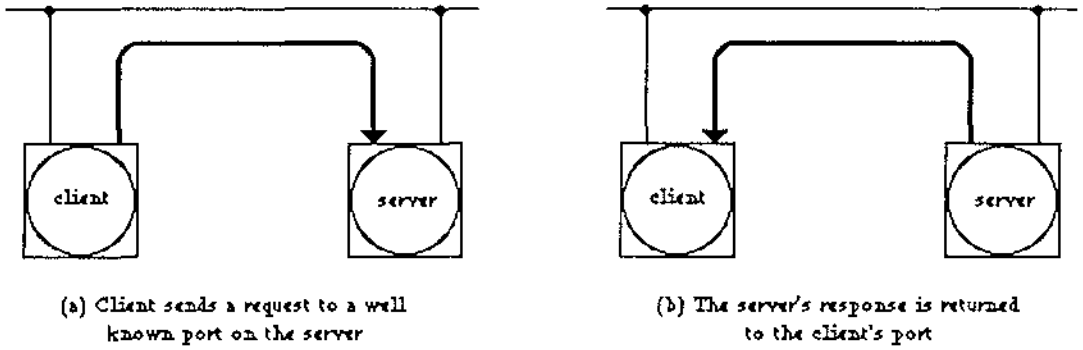
Providing users with the ability to develop X.25 software on IP workstations could have been done in two ways. One way would have been to have a transport bridge (figure 4.9) which would forward packets between the IP hosts and the X.25 host. This would have been a satisfactory option had the IP hosts been running ISODE because they would have provided the necessary X.25 interface. However, the X.25 host and the IP hosts would have had to be running OSI protocols and ISODE. Since this was not the case, this option was not investigated further.

The implementation of X.25 over TCP/IP was the other option. X.25 packets would be sent from the IP workstations to the Pyramid using TCP/IP. The Pyramid would then forward the X.25 packets to the X.25 network, therefore acting as an X.25/IP bridge. Similarly, incoming X.25 packets received from the X.25 network would be forwarded to the IP workstation along the same TCP/IP connection by the Pyramid. This option was clearly straightforward to implement and did not require ISODE.

The implementation consisted of a programmer's interface in the form of a library for use by users on IP workstations, and two server programs on the Pyramid computer which would provide the necessary X.25 services to the library. The implementation was based on the client-server model of interaction described below.

5.2 The Client-Server Model of Interaction

The client-server model forms the basis of most network communication. It is a concept whereby two or more applications can interact. For interaction, at least one application must be a client program and at least one application must be a server program. The client and the server are usually, but not always, run on separate hosts.



(a) Client sends a request to a well known port on the server

(b) The server's response is returned to the client's port

Figure 5.1 - The Client-Server model

The server program offers a service that can be reached over the network. Servers accept requests that arrive over the network, perform the specialized service that they were designed to do, and then send the result back over the network to the requester. The client program is the requester. The client formulates a request, sends it to a server, and awaits a response. One example of a client-server implementation is a file server. A file server performs operations on files usually residing on a hard disk. The file server receives requests such as add, delete or modify from client programs, performs the request, and sends the results back to the client program.

Servers are usually implemented as application programs. On larger, timesharing systems, a server may be run alongside other applications. On smaller systems, a server may be the only application running, making the machine a dedicated server. To increase reliability, it is acceptable to run several copies of the same server either on the same machine and/or on different machines. The purpose of this redundancy is to make the service offered by the server more readily available.

A server must begin execution before interaction begins and ideally does not ever terminate without manual intervention. It continues to accept requests from client programs and perform services until it is stopped. In contrast, the client program cannot obtain services

from a server until the server has been started. The client program may then be executed several times to obtain the services.

A server must listen in on a well-known software port for an incoming request from a client. This port is reserved for the service that is offered. The client must specify this port when sending a request to the server and all communication between the client and the server is through that port.

The services that servers typically provide are usually in demand by many client programs. It is not adequate therefore, for a server to accept a client request and become pre-occupied with it whilst other clients are queueing for the same service. To overcome this, servers are written in such a way that they can handle multiple requests. This means that a server is always listening for new clients, but also processing unfinished requests.

In addition to handling concurrent requests, server programs must be responsible for maintaining security. Servers usually need to operate in an environment which is highly privileged because they need to be able to read system files, maintain logs and access protected data. Therefore, servers should not allow a client to request its services without first verifying that the request is authorized.

Servers must be robust. It is quite possible for a client to send invalid requests. If this happens, the server should be able to handle it in an appropriate manner such as discarding it without aborting itself. They should be written in such a way that they require the minimum amount of human intervention.

5.3 Overview of the Implementation

The X.25 programming library provides a programmer on an IP workstation with a number of routines for establishing X.25 virtual calls, sending & receiving data, and obtaining statistical information about the calls. The library obtains X.25 services from the two server programs which operate on the Pyramid.

On the Pyramid side, the main server is called the Pyramid server. This server must be operational to allow the X.25 library to initiate an outgoing X.25 virtual call or to accept an incoming X.25 virtual call. The second of the two servers is known as the Incoming Call

server (ICS). The ICS program is a reduced version of the Pyramid server and is used specifically with incoming X.25 virtual calls.

The X.25 library provides the interface through which a programmer can make an outgoing call or accept an incoming call. Outgoing calls are made in the following way. The X.25 library connects to the Pyramid server as a client and sends it certain information which is necessary for call establishment (figure 5.2(a)). On receiving this information, the Pyramid server forks a child process (figure 5.2(b)) which proceeds to service the call and leave the parent copy free to accept requests from new client programs (figure 5.2(c)).

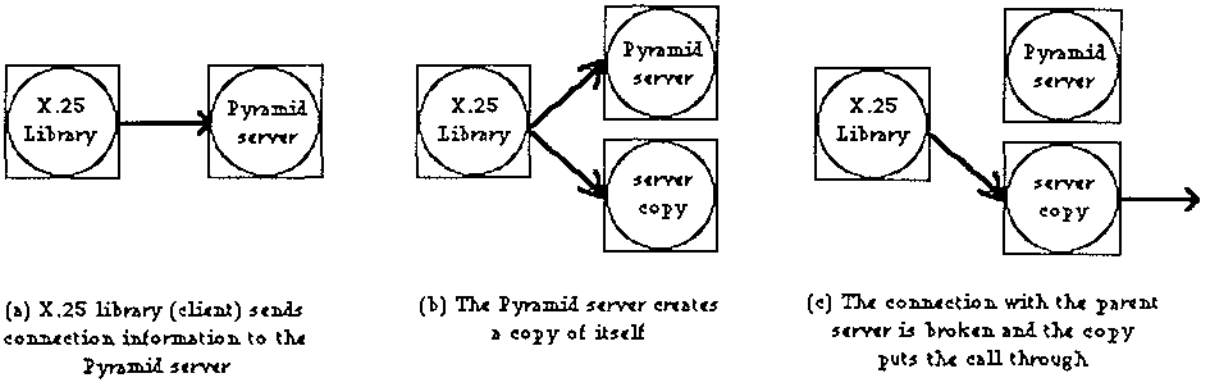


Figure 5.2 - Outgoing call example

The procedure for accepting incoming X.25 virtual calls is more complicated. The X.25 library connects to the main server program on the Pyramid as a client and informs it of the type of call that it is prepared to accept by sending it certain 'binding' information (figure 5.3(a)). It then ceases being a client and imitates a server by going into a listening state where it listens out for incoming calls sent to it by the Pyramid server. An incoming X.25 virtual call to the Pyramid will cause the activation of the ICS program. On start-up, the ICS program connects to the Pyramid server and sends it the characteristics of the call that it is 'carrying' (figure 5.3(b)). The Pyramid server then attempts to match these call characteristics with binding information previously placed there by an X.25 library. If a match is found, then the ICS program is re-routed to the appropriate IP workstation where the X.25 library will be listening for incoming calls (figure 5.3(c)). The X.25 library can then accept the incoming call and begin the data transfer phase. When the X.25 library does not wish to accept further incoming calls, it re-connects to the Pyramid server and cancels the binding information it previously placed there (figure 5.3(d)).

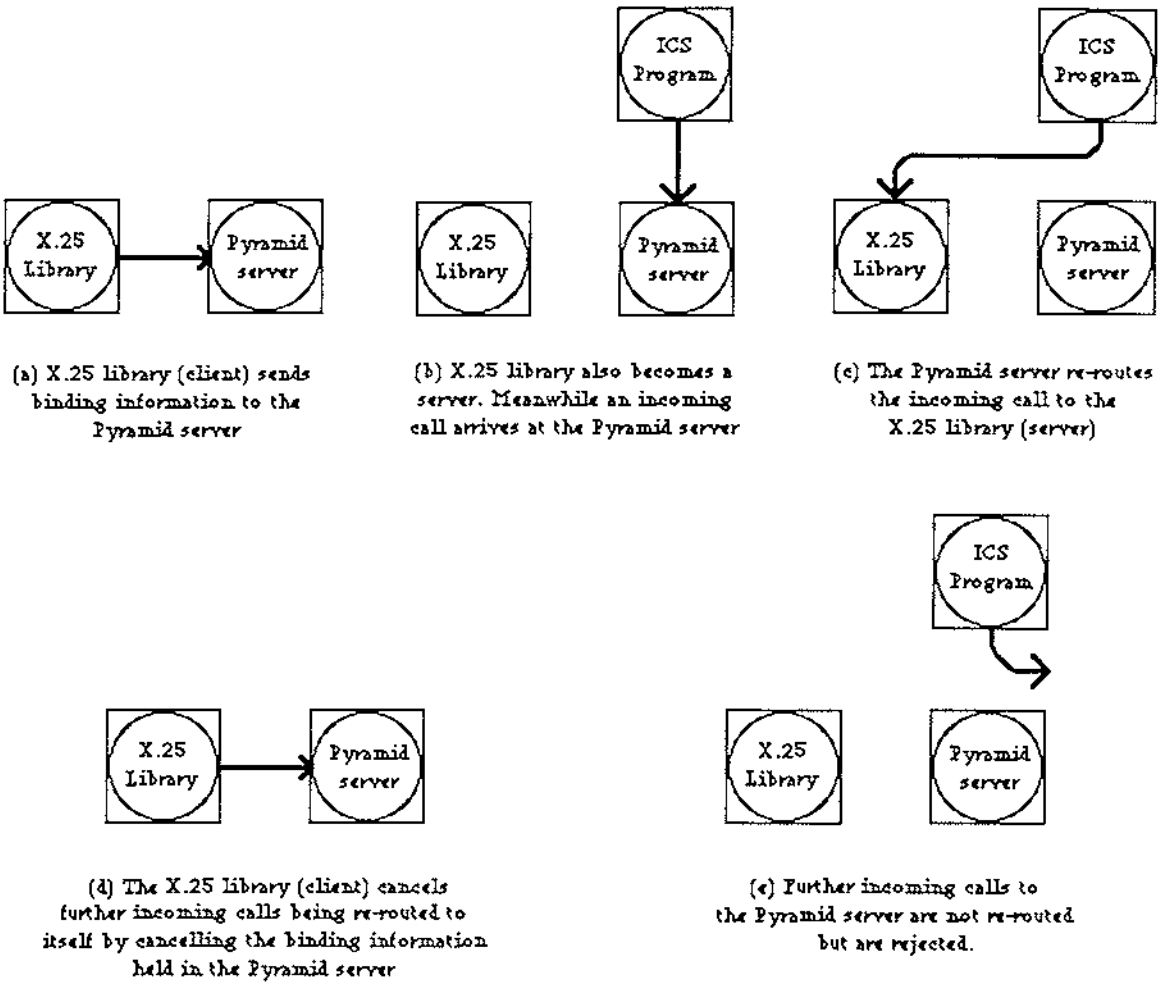


Figure 5.3 - Incoming call example

5.4 The X.25 Library

The X.25 programming library exists as an object file which must be linked with the user's (object) file to produce an executable run file. Under UNIX for example, a 'C' program called MyProgram.c will initially be compiled with **cc** to produce an object file called MyProgram.o. This would then be linked with the X.25 library x25lib.o to produce an executable run file. Ideally, a programmer would utilise the **make** facility offered within UNIX.

The interface to the library is discussed in section 5.5.

5.4.1 Modes of Operation

The implementation allows a user to work in packet mode or data mode. This is a feature that is found in the Pyramid's native X.25 environment. In packet mode, the user must format all outgoing data into X.25 packets following the guide-lines for packet formats defined by the CCITT (section 2.5). Similarly the user will receive data formatted into standard X.25 packets. Only the logical channel numbering and packet sequencing is handled by the X.25 software on the Pyramid system. Packet mode is needed to use certain facilities such as the D bit.

In data mode, the user can pass data free of X.25 headers to the X.25 library, and similarly receive back only data. The library takes responsibility for splitting up input data into 128-octet chunks, encapsulating these chunks into standard X.25 packets and then forwarding them to the server for transmission on the network. The library is also responsible for decapsulating data from X.25 packets received back from the server and sending it to the user program. The ioctl() call provides a means of performing certain actions on packets which would normally require the user to operate in packet mode, such as setting or getting the Q bit in a packet and sending INTERRUPT or RESET REQUEST packets.

The mode of operation may be set using the second parameter to the socket() call.

5.4.2 Buffering

An X.25 data packet may contain up to 128 octets of data. In data mode it becomes necessary to buffer the data following decapsulation. The read() call allows a user to specify the maximum number of octets to read which can be one at the least. The X.25 library maintains a buffer for each virtual call which is filled with decapsulated data from a newly arrived data packet.

5.4.3 Handling Special Packets

The X.25 library uses the TCP/IP out-of-band facility to receive urgent signals from the servers. The servers send an urgent signal to indicate the arrival of INTERRUPT or RESET REQUEST packets. The out-of-band facility allows the servers to by-pass normal flow control by sending the special signal to the X.25 library in lieu of sending the packet

itself. The X.25 library on the IP workstation receives this signal in the form of an out-of-band octet. The octet indicates the kind of special packet that was received and the library may then elect to generate a lookalike packet to present to the read() routine.

The action taken by the X.25 library when an urgent signal is received depends on the mode that the user is working in. In data mode the X.25 library ignores INTERRUPT signals but a RESET REQUEST signal is handled by returning from a read() or a write() with a -1.

Signals received in packet mode are handled with a little more sophistication. INTERRUPT and RESET REQUEST urgent signals will cause the re-construction of INTERRUPT and RESET REQUEST packets respectively which are made available on the next read(). The user can then handle these packets in an appropriate manner.

The X.25 specification requires an INTERRUPT packet to be acknowledged with an INTERRUPT CONFIRMATION packet. When the Pyramid computer receives an INTERRUPT packet from the X.25 network, it assumes the responsibility for sending back the INTERRUPT CONFIRMATION packet. However, the user who is working in packet mode should disregard the Pyramid's action by sending an INTERRUPT CONFIRMATION packet in accordance with the X.25 specification. These packets will be filtered out by the servers on the Pyramid computer and not forwarded to the X.25 network.

5.5 The X.25 Library Interface

The interface to the X.25 programming library has been designed to resemble the UNIX socket interface. A socket is created for use in a specific communication domain. The domain type is specified in the first parameter to the socket() call when the socket is created. Two commonly used domains are AF_UNIX (UNIX system internal protocols), and AF_INET (ARPA Internet protocols). A third domain called AF_X25 has been provided for X.25 sockets which a programmer must specify when using the X.25 programming library.

Routines return values in two ways, either as the result of the function or by modification of the parameters. A header file called "extras.h" provides data structures for the programmer's convenience and must be included at the beginning of any programs.

5.5.1 socket()

```
#include "extras.h"

int socket(domain, type, protocol)
int domain;
int type;
int protocol;
```

The socket() call is used to create a socket. The socket abstraction is an endpoint for communication which may be used for sending or receiving data from a remote machine. As when opening files, opening a socket will return a unique identifier called the socket descriptor which must be specified whenever an operation is being performed on that socket. Initially sockets are created without any association to local or destination addresses. A connect() or a bind() will normally be expected after calling socket().

The domain parameter specifies a communication domain within which communication will take place. To create an X.25 socket, the domain is AF_X25. This states that the X.25 address family is to be used with the socket. The second parameter specifies the type of the socket. In the AF_X25 domain, the type may be SOCK_STREAM for data mode or SOCK_RAW for packet mode. The final parameter identifies a protocol within an address family to be used with the socket. This value should be set to 0 in the AF_X25 domain.

If the socket() call is unable to perform the requested action it will return with a -1. The variable errno will contain the error number and the user may then call the perror() routine for a null-terminated string which describes the error more fully.

5.5.2 connect()

```
#include "extras.h"

int connect(s_desc, name, namelen)
int s_desc;
struct addr_type *name;
int namelen;
```

The connect() call is used to initiate a connection on a socket. In the AF_X25 domain, connect() works as follows. In packet mode the connect() call is used to make an IP connection to the Pyramid server. After the connection has been made, the user is responsible for initiating the X.25 call establishment procedure by exchanging the appropriate X.25 control packets using read() and write(). The parameter s_desc is a socket descriptor returned by socket() which must be provided. The name and namelen fields may be cast to NULL pointers in packet mode.

In data mode the connect() call is used to make a connection to the Pyramid server and then establish the X.25 virtual call. The routine generates an X.25 CALL REQUEST packet and follows the call set-up procedure described in section 2.5.1. The parameter s_desc is a socket descriptor returned by socket(). The second parameter contains connection information to be used by the routine. The connection information is a structure with the following format:

```
struct addr_type {
   char address[14];
   char subaddress[2];
   char protocol_id[8];
   char call_user_data[12];
   char mask[12];
   int port_number;
};
```

The address field contains the X.25 address to connect to, for example "63000064". The subaddress is a 2-octet field which contains an X.25 secondary address. This secondary address is appended to the end of the X.25 address.

The protocol identifier field is a 4-octet field used by X.25 software to decide which high level protocol handler to pass CALL REQUEST packets to when they arrive. The standard protocol identifier for the X.29 packet assembler/disassembler (PAD) is an octet containing one in the first octet. The second octet may contain a PAD profile of the terminal which is expressed as an integer, although this facility is not generally used. The third octet may contain a copy of X.3 parameter 11 (section 5.8.1) indicating the speed of the connection between the PAD and the terminal. The last octet is left zero. The procotol identifier should be specified in hexadecimal, so the protocol identifier field must contain "01000000".

The 4-octet protocol identifier field forms the first part of 16 octets of user data. The remaining 12 octets of user data may be specified in the call_user_data field. The address, the subaddress and the 16 octets of user data are encoded into the CALL REQUEST packet that is generated when operating in data mode. The mask and the port_number fields in the structure are unused.

If the connect() call is unable to perform the requested action it will return with a -1. The variable errno will contain the error number and the user may then call the perror() routine for a null-terminated string which describes the error more fully.

5.5.3 bind()

```
#include "extras.h"

int bind(s_desc, name, namelen)
int s_desc;
struct addr_type *name;
int namelen;
```

The bind() call is used to assign a name to an unnamed socket. In the AF_X25 domain, a bind() tells the Pyramid server to listen out for incoming calls containing the attributes held in name. The name field is an addr_type structure described in section 5.5.2. The address and subaddress fields contain the X.25 address to listen to for incoming calls.

Since several users may be waiting for incoming calls at the same X.25 address, further options are necessary to identify the true recipient of an incoming call. A listener must provide a unique string in the call_user_data field and a port number on the local IP workstation which the incoming call can be re-routed to. For example, if a user named John provides the string "John" with a port number of 1025 then any incoming calls containing "John" will be re-routed to port 1025 on John's workstation where he will be waiting. Since "John" has been reserved by John it cannot be used by anyone else until he performs a close() operation which will inform the Pyramid server that "John" is no longer reserved.

A port number is of local significance only. If John is listening on port number 1025 and on the same machine another user called Shirley wants to listen for connections for "Shirley", then she must use a different port number such as 2049. However if Shirley was working from a different IP workstation, then she may specify port number 1025 as long as this was not being used.

5.5.4 listen()

```
#include "extras.h"

int listen(s_desc, qlength)
int s_desc;
int qlength;
```

The listen() call is used to tell the X.25 library to queue up to qlength incoming connections. The qlength defines the maximum length the queue of pending connections may grow to. If the listen() call is unable to perform the requested action it will return with a -1. The variable errno will contain the error number and the user may then call the perror() routine for a null-terminated string which describes the error more fully.

5.5.5 accept()

```
#include "extras.h"

int accept(s_desc, addr, addrlen)
int s_desc;
struct addr_type *addr;
int *addrlen;
```

The accept() call is used to accept an incoming connection. In the AF_X25 domain, accept() tells the X.25 library to listen for incoming connections from the ICS program. The ICS program will bring in an incoming X.25 virtual call with it. The routine blocks until a connection arrives but blocking may be avoided by calling select() first to check for a pending connection before calling accept(). The addr parameter returns a pointer to a structure of type addr_type and will contain information about the incoming call.

The routine returns a new socket descriptor which must be used for the remainder of that call. In packet mode the accept() must be followed by a read() to get the INCOMING CALL packet. The programmer is responsible for completing the call establishment phase by sending a CALL ACCEPT packet or a CLEAR INDICATION packet. In data mode the accept() routine will complete the call establishment phase by sending a CALL ACCEPT packet.

If the accept() call is unable to perform the requested action it will return with a -1. The variable errno will contain the error number and the user may then call the perror() routine for a null-terminated string which describes the error more fully.

5.5.6 write()

```
#include "extras.h"

int write(s_desc, user_buffer, size)
int s_desc;
char *user_buffer;
int size;
```

The write() call is used to write size octets of data to the object referenced by the descriptor s_desc from the buffer pointed to by user_buffer. In the AF_X25 domain, the function of write() has been modified for handling X.25 packets. In other domains write() behaves normally.

Users send data or packets using the write() call. In packet mode write() expects to be given X.25 formatted packets which it sends out immediately. The user is responsible for following the CCITT guide-lines for formatting X.25 packets. In data mode write() takes over this responsibility by formatting the data itself. It expects to be given unformatted data which it divides into 128-octet chunks, encapsulates the chunks into X.25 packets and sends out the packets.

The size parameter must be included and represents the size of the user buffer. The first parameter is the socket descriptor which is returned from socket() or accept().

5.5.7 read()

```
#include "extras.h"

int read(cd, user_buffer, size)
int cd;
char *user_buffer;
int size;
```

The read() call is used to read up to size octets of data from the object referenced by the descriptor s_desc into the buffer pointed to by user_buffer. In the AF_X25 domain, the function of read() has been modified for handling X.25 packets. In other domains read() behaves normally.

The read() call is used to receive data or packets from the X.25 library. The routine maintains one internal buffer for every connection that is in use. When called, the number of pending octets in the internal buffer is assessed. If the buffer is empty the routine reads an X.25 packet from the IP connection to the server. This packet is decapsulated and the data is copied into the internal buffer. Next, the routine checks how much space is available in the user's buffer, identified by the size parameter. It copies as much data as possible from the internal buffer to the user's buffer. If the user's buffer is still not exhausted and the internal buffer becomes empty, the process is repeated until either there are no more packets waiting to be read on the IP connection or the user's buffer becomes full. If the internal buffer is only partly read before the user's buffer becomes full, then a subsequent read() operation will carry on reading the internal buffer from the point where the previous read() operation ended.

The routine then copies data from its buffer of incoming information into the user's buffer. The user must indicate the size of the user buffer. The first parameter is the socket descriptor which is returned from from socket() or accept().

5.5.8 select()

```
#include "extras.h"

int select(nfds, readfds, writefds, exceptfds, timeout);
int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
```

The select() call examines the I/O descriptor sets whose addresses are passed in readfds, writefds and exceptfds to see if some of their descriptors are ready for reading, ready for writing, or have an exceptional condition pending. The nfds parameter is the number of bits to be checked in each descriptor set. On return, select() replaces the given descriptor sets with sets containing those descriptors that are ready for the requested operation. The total number of ready file descriptors in all of the sets is returned.

The select() call has been modified for use in the AF_X25 domain. The modified select() checks readfds for any descriptors created using socket() or accept(). The routine then checks the internal buffer of each one found for any buffered data waiting to be extracted. If a buffer is not empty then the descriptor is temporarily removed from readfds. After all descriptors in readfds are processed in this fashion, a conventional select() system call is performed using the modified readfds parameter and the remaining (unmodified) parameters. Finally, any descriptors which were temporarily removed from readfds are put back.

If select() is unable to perform the requested action it will return with a -1. The variable errno will contain the error number and the user may then call the perror() routine for a null-terminated string which describes the error more fully.

5.5.9 ioctl()

```
#include "extras.h"

int ioctl(s_desc, request, arg)
int s_desc;
int request;
char *arg;
```

The ioctl() call performs a special function on the object referred to by the open descriptor s_desc. The set of functions that may be performed is dependent on the object that s_desc refers to. The function is identified by the second parameter. In the AF_X25 domain, ioctl() can be used to perform the following functions:

a) IOGETSTAT - If request is IOGETSTAT, the ioctl() routine returns a structure containing statistical information about the X.25 virtual call described by s_desc. The information is returned in arg and is contained within a structure of the following type, as defined in the header file "extras.h":

```
struct stat_type {
   enum modes mode;
   char x25_address[14];
   char x25_subaddress[2];
   char x25_protocol_id[8];
   char x25_call_user_data[12];
   char x25_mask[12];
   int tcp_port_number;
   int time_elapsed;
   int packets_in;
   int packets_out;
};
```

b) IOGETMODE - The IOGETMODE request may be used to get the mode of the X.25 virtual call. The return value in arg may be 0 (packet mode) or 1 (data mode).

c) IOSETMODE - If request is IOSETMODE, the mode of the X.25 virtual call may be set to the value in arg. The value contained in arg should be 0 (packet mode) or 1 (data mode). The mode is initially set using the second parameter to socket().

d) IOGETQBIT - The IOGETQBIT request may be used in data mode to check if the next incoming X.25 packet has the Q bit set, therefore making the data received on the next read() qualified data.

e) IOSETQBIT - If request is IOSETQBIT, the Q bit of the next X.25 packet formatted by the library in data mode during a write() will have the qualified data bit set or cleared, as specified in arg. Possible values for arg are 1 (TRUE) or 0 (FALSE).

f) IOX25INT - The IOX25INT request may be used in data mode to send an X.25 INTERRUPT packet to the X.25 network.

g) IOX25RESET - The IOX25RESET request may be used in data mode to send an X.25 RESET REQUEST packet to the X.25 network.

If the ioctl() call is unable to perform the requested action it will return with a -1. The variable errno will contain the error number and the user may then call the perror() routine for a null-terminated string which describes the error more fully.

5.5.10 close()

```
#include "extras.h"

int close(s_desc)
int s_desc;
```

The close() call deactivates the open descriptor s_desc prior to deleting it from the object reference table. In the AF_X25 domain close() performs a number of tasks depending on the current state of the socket referenced by s_desc. If s_desc was created using socket() and was followed by a bind() to log binding information with the Pyramid server, then close() will re-connect to the Pyramid server to remove this binding information so that incoming calls are no longer re-routed to itself.

The procedure is different if s_desc was created using socket() and was followed by a connect() or s_desc was returned from accept(). In data mode close() sends a CLEAR REQUEST packet to terminate an X.25 virtual call if one is in progress. In packet mode this becomes the responsibility of the programmer.

5.5.11 perror()

```
#include "extras.h"

perror(s)
char *s;
```

The perror() call takes a null terminated string as a parameter which it prints out as a prefix to an error message determined by the variable errno. If the string parameter is null then only the error message will be printed.

5.6 The Pyramid Server

The Pyramid server processes requests from client programs either iteratively or concurrently. The kind of processing it performs depends on the type of the request. In the iterative approach, the server accepts a new connection, handles the request, closes the connection and waits for the next client. In the concurrent approach, after a connection has been accepted the server forks a child process which handles the request so that the parent copy is free to accept requests from new clients.

The Pyramid server's role may be described more fully along these terms. In its first capacity, it accepts a connection request from a client such as the ICS program or an X.25 library on an IP workstation, processes the request, terminates the connection and waits for another client. In its second capacity, the server shuttles X.25 packets between an IP workstation and the X.25 network. Figure 5.4 shows how the server's operation changes from an iterative approach to a concurrent approach. Initially, the server will be listening and handling connection requests. When a client connects and places a request for an outgoing call, the server forks a child process. The child process manages the outgoing call by shuttling X.25 packets and the parent server resumes the job of listening for new connection requests.

Server programs generally operate at well known ports. On start-up, the Pyramid server program performs a number of steps. A socket is opened which is bound to port number 1025. This port is used to listen for incoming connections. Next, the Pyramid server tells the operating system how many connections it would like queued. Generally, if a server uses a reliable stream delivery or if processing a request takes a reasonable length of time, it may happen that a new request arrives before the server finishes responding to an old request. Therefore the Pyramid server gives the operating system a backlog number, this being the number of connections to queue. This backlog number has been set to 5 requests.

5.6.1 Servicing Clients

Clients are serviced as follows. The Pyramid server asks the system if there are any clients wishing to connect to it. If there are, the server goes into an accept state in which it can accept the pending connection. After accepting it the server program reads an identity octet sent by the client which identifies the type of service being sought. The server then follows an appropriate course of action.

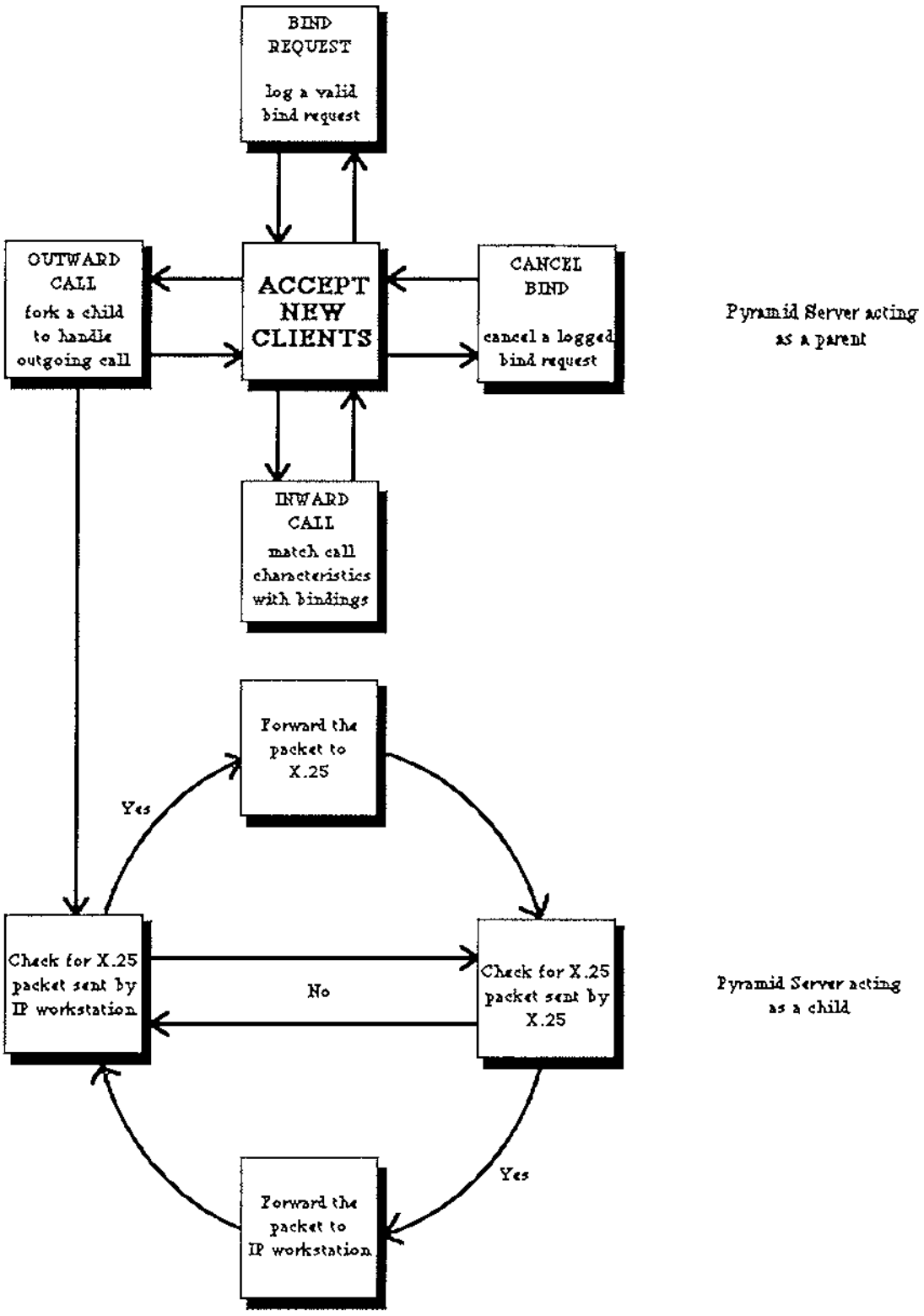


Figure 5.4 - Task Diagram for the Pyramid server

There are four kinds of requests:

a) OUTWARD_CALL - This request is made by a client wishing to place an outgoing X.25 virtual call. The client is an X.25 library on an IP workstation. On receiving this, the Pyramid server forks a child process which takes on the responsibility of managing the outgoing call. The child process reads suitably formatted X.25 packets from the client which it writes out to the X.25 network. Similarly it reads incoming packets from the X.25 network which it forwards to the client. When the client program shuts down the connection, this shuttling activity ceases and the child process terminates.

b) BIND_REQUEST - This request is placed by a client which wishes to accept incoming calls. The client is an X.25 library on an IP workstation which sends binding information to the Pyramid server (section 5.3). The binding information contains five X.25 fields and one IP field. X.25 fields consist of an X.25 address, an X.25 subaddress, a protocol identifier, a 12-octet string containing call user data and a 12-octet mask. The IP field contains an integer port number.

When the Pyramid server receives the binding information, it identifies the client by its 32-bit Internet address and appends this address to the binding information. Existing bindings are checked against the new binding information for similarities. If a binding is found which has identical X.25 fields and IP fields, then the new binding information is not added to prevent redundancy. If a binding is found with the same X.25 fields but different IP fields, the client's request is rejected and the connection is terminated because this implies that there is already a client who is accepting incoming calls with that call user data. Lastly, if a binding cannot be found with similar X.25 fields, then the new binding information is added to the list of current bindings and the connection is terminated.

c) INWARD_CALL - This request is placed by the ICS program which brings an incoming X.25 virtual call with it. The ICS program connects to the Pyramid server and informs it that there is an incoming X.25 virtual call. The ICS program sends the Pyramid server information about the call such as the X.25 address, the X.25 subaddress, protocol identifier and call user data. On receiving this information, the Pyramid server searches its list of bindings for any which have matching X.25 fields. If one is found, then the Pyramid server sends the ICS program the port number and the 32-bit Internet address of the IP workstation which previously sent that binding. The connection is then terminated and the

ICS program uses the port number and the Internet address to connect to the IP workstation. Hence the call is 're-routed' to an IP workstation (figure 5.3).

d) CANCEL_BIND - This request is made by an X.25 library wanting to cancel a binding request which it previously placed in the Pyramid server. The client sends binding information to the Pyramid server which is identical to the binding information originally sent to log the binding request. The Pyramid server uses this information to search its list of current bindings and remove the binding. The result is that any future incoming calls will not be re-routed to the IP workstation because there is no binding information present.

5.6.2 Managing an X.25 virtual call

To manage an X.25 virtual call, the Pyramid server must firstly receive an OUTWARD_CALL request after which it must fork a child process. It is the child copy of the Pyramid server which manages the X.25 virtual call. After forking, the child server inherits all descriptors, signals and variables from the parent. Furthermore, pending connection requests may become queued at both servers. Therefore it is necessary for the child server to sever its association with its parent. This is done by closing off the descriptor which was originally assigned to listen for connection requests.

The server performs four tasks. It first looks for an X.25 packet that has been sent by an IP workstation. If there is one, the packet is forwarded to X.25. After the packet has been forwarded (or if there was no packet), the server looks for an X.25 packet that has been sent by X.25. If there is one, the packet is forwarded to the IP workstation. After the packet has been forwarded (or if there was no packet), the server repeats the procedure (figure 5.4).

When the Pyramid server receives special X.25 packets such as INTERRUPT or RESET REQUEST it by-passes normal flow control by sending a special signal to the IP workstation that a such packet has arrived. This is done in lieu of sending the packet itself. The X.25 library on the IP workstation receives this signal in the form of an out-of-band octet. The octet will indicate the kind of special packet that was received and the library creates a lookalike packet which is presented to the read() routine.

The Pyramid computer responds to an INTERRUPT packet with an INTERRUPT CONFIRMATION packet, therefore eliminating the need for the user on the IP workstation

to send it. However, a user working in packet mode should follow convention and send the INTERRUPT CONFIRMATION. The Pyramid server will filter out INTERRUPT CONFIRMATION packets sent by the user.

Other special packets such as CLEAR REQUEST, INTERRUPT CONFIRMATION etc are sent in the usual fashion to the user's program on IP workstation.

5.7 The Incoming Call Server (ICS)

The ICS program is the combination of a client and a server. It acts as a client program when requesting services from the Pyramid server. It acts as a server when it offers X.25 services to client programs that request it. The ICS program is started up by the system when an incoming X.25 call is received on subaddress 10 on the Pyramid.

Incoming calls are firstly sent to a process on the system called the X.25 daemon. This process studies the incoming call, which is usually in the form of an X.25 call request packet. Embedded within this packet is a destination address which will be the local X.25 address. Appended to this address is a 2-octet field called the subaddress. The subaddress identifies a user process which execution must be passed. In this case the X.25 daemon passes execution to the user process which is the ICS program. The X.25 daemon also provides the ICS program with the logical channel number on which the incoming call has arrived as a parameter.

5.7.1 Operating as a client program

On start-up, the ICS program operates as a client program. It opens the X.25 channel using the logical channel number passed to it by the X.25 daemon. A read() system call is performed to get the X.25 call request packet. The X.25 address, X.25 subaddress, protocol identifier and the call user data are extracted from the packet. A socket is then opened and assigned to port number 1025. This socket is used to establish a connection with the Pyramid server.

When the ICS program has established the connection, it informs the Pyramid server that there is an incoming call by sending an INWARD_CALL request and the information that was extracted from the X.25 CALL REQUEST packet. The Pyramid server attempts to match the X.25 information to bind requests logged by IP workstations. If a match is not found then ICS program's request is rejected and the connection is closed. If the Pyramid server has matched the ICS program's incoming call request to a bind request, it sends back the binding information to the ICS program. This information includes a 32-bit Internet address of the IP workstation and the port number on which the workstation will be listening for incoming calls. On receiving this, the ICS program closes the connection with the Pyramid server.

The ICS program now opens a socket which is assigned to the port number given by the Pyramid server. This socket and the 32-bit Internet address are used to establish a connection with the IP workstation. When the connection has been established, the ICS program introduces itself to the IP workstation by sending it a message about the X.25 call that it is carrying. This includes the X.25 address, X.25 subaddress, protocol identifier and the call user data. This is followed by the X.25 CALL REQUEST packet itself. At this point, the ICS program ceases being a client and becomes a server.

5.7.2 Operating as a server program

The ICS program provides the same X.25 services that the Pyramid server provides and works in an identical fashion. Like the Pyramid server, the ICS program performs four tasks. It first looks for an X.25 packet that has been sent by the IP workstation. If there is one, the packet is forwarded to X.25. After the packet has been forwarded (or if there was no packet), the server looks for an X.25 packet that has been sent by X.25. If there is one, the packet is forwarded to the IP workstation. After the packet has been forwarded (or if there was no packet), the server repeats the procedure (lower half of figure 5.4).

When the ICS program receives special X.25 packets such as INTERRUPT or RESET REQUEST it by-passes normal flow control by sending a special signal to the IP workstation that a such packet has arrived. This is done in lieu of sending the packet itself. The X.25 library on the IP workstation receives this signal in the form of an out-of-band octet. The octet will indicate the kind of special packet that was received and the library creates a lookalike packet which is presented to the read() routine.

The Pyramid computer responds to an INTERRUPT packet with an INTERRUPT CONFIRMATION packet, therefore eliminating the need for the user on the IP workstation to send it. However, a user working in packet mode should follow convention and send the INTERRUPT CONFIRMATION. The ICS program will filter out INTERRUPT CONFIRMATION packets sent by the user.

Other special packets such as CLEAR REQUEST, INTERRUPT CONFIRMATION etc are sent in the usual fashion to the IP workstation.

5.8 Code Examples

This section presents some examples of programs which make use of the X.25 programming library. The programs must include a 'C' compiler directive '#include "extras.h"' which will include a header file containing a number of declarations for use by the programmer. This file has been reproduced in section 5.8.4.

5.8.1 Example 1 - Simple Packet Assembler/Disassembler (Spad)

The following program is a simple packet assembler/disassembler (PAD). It demonstrates the use of the X.25 programming library for writing programs for outgoing X.25 virtual calls. The program has been written in data mode. Spad consists of a user interface which uses X.3/X.28 communication protocols to allow access to X.25 packet switching networks from asynchronous terminals.

Spad Commands

When Spad is invoked, an asterisk (*) appears as a prompt. The asterisk indicates that the program is in command mode and any of the commands from the following table may be typed (in lower case):

Command	Description
c	Establish call to a specified address
clr	Send clear packet and exit
help, ?	List available commands
int	Send interrupt packet
par	Display specified X.3 parameter or all parameters
quit	Quit the Spad program
reset	Send reset packet
set	Set and display specified X.3 parameter
stat	Request status of the virtual call
^P	Enter command mode from data transfer mode

An X.25 virtual call is established using *c address [-d protocol id] [call user data]*. The address field is a valid X.25 address with a maximum size of 15 octets. A protocol identifier may be specified in hexadecimal. If this field is not specified, a default value of

"01000000" will be used. The call user data field may contain up to 12 ascii characters which will be sent over the virtual call as those literal characters. This field is provided for the user's individual purpose. Call requests are sent to the network, and if the call establishment phase is successful, a message "COM: virtual call established" will be displayed.

A virtual call may be cleared using *clr*. A CLEAR REQUEST packet is sent to the X.25 network and the pad exits to the UNIX shell. The message "Call cleared" is displayed as a confirmation that the X.25 virtual call has been terminated.

Help about the commands in Spad may be obtained using *help* or *?*. A list of commands is displayed along with a brief description of each one.

The *int* command may be used to send an INTERRUPT packet to the network. When confirmation is received, the message "Interrupt confirmed" will be displayed. To reset the virtual call, the *reset* command may be used. This command sends a RESET REQUEST packet to the network. When a reset confirmation is received, the program terminates.

X.3 parameters may be viewed with the *par [param]* command. On its own, the command displays the current settings of all the X.3 parameters. The current value of an individual X.3 parameter may be displayed by providing the parameter number as an argument. X.3 parameters may altered to new values using the *set param newvalue* command. When a parameter is changed, the new value will be displayed.

The status of the X.25 virtual call can be obtained with the *stat* command. The command checks the current state of the virtual call and displays information consisting of the address connected to, protocol identifier, call user data, elapsed time, number of packets received and number of packets transmitted.

The program may be exited using *quit* if there are no X.25 virtual calls in progress. The *clr* command should be used otherwise. On exiting, the user is returned to the UNIX shell.

By default, ^P (control-P) is used to enter the command mode from the data transfer mode. However, this is subject to the current value of X.3 parameter 1 (described below) and hence the pad recall character may be configured to a new value.

X.3 Facilities provided by Spad

The X.3 specification defines a set of parameters which control the behaviour of pad programs in response to data flowing between the terminal and the pad. By the alteration of these parameters the host computer to which the user is connected can automatically customise the behaviour of the pad under certain conditions. Alternatively, the user can also change the parameter settings to optimise the use of the pad. The Spad program implements a subset of these parameters, consisting of X.3 parameters 1 to 4 and X.3 parameter 11:

a) Parameter 1 (Escape to PAD command state) - This parameter determines whether the pad can be recalled by the user during data transfer. Values may be 0 (not possible), 1 (possible with the DLE character control-P), or 32-126 (possible with one of the graphic characters defined by the user.

b) Parameter 2 (Character echo) - This parameter determines whether characters entered into the terminal will be echoed by the pad. Values may be 0 (no echo) or 1 (echo).

c) Parameter 3 (Data forwarding characters) - This parameter determines which characters, if any, will cause the pad to forward the data currently held in the pad buffer. The following values may be used:

Value	Description
0	No data forwarding on a character
1	Alphanumeric characters (A-Z, a-z, 0-9)
2	Carriage return (CR)
4	Characters ESC, BEL, ENQ, ACK
8	Characters DEL, CAN, DC2
16	Characters ETX, EOT
32	Characters HT, LF, VT, FF
64	All other characters less than decimal 32 in value

In order to customise the pad characteristics, the above values of parameter 3 may be summed to give the desired forwarding characters.

d) Parameter 4 (Data forwarding on time-out) - This parameter determined whether the pad will automatically forward the data currently in the pad buffer upon expiration of a time

interval. The value may be 0 (no timeout and automatic forwarding), or 1-255 (a timeout value in twentieths (1/20) of a second).

e) Parameter 11 (Terminal speed) - This parameter is read-only and determines the rate of data transmission across the interface between the pad and the terminal:

Value	Terminal Speed (Bits per second)
0	110
1	134.5
2	300
3	1200
4	600
5	75
6	150
7	1800
8	200
9	100
10	50
11	75 (transmit), 1200 (receive)
12	2400
13	4800
14	9600
15	19200
16	48000
17	56000
18	64000

Communication between Spad and the host computer to which it is connected involves the exchanging of normal (user) data and control (qualified) data. Qualified data is contained within an ordinary X.25 data packet which has the Q bit (bit 8) set on the first header octet. The qualified data packet normally contains a message to the pad program by the host computer. The first octet of the qualified data (not including the X.25 packet header) identifies the message type. The following messages are recognised by Spad:

a) Set Pad message - This command is sent by a host to set some or all of the pad parameters. The message contains pairs of octets, the first of which is the binary value of

the parameter number, and the second of which is the value to which the pad should set that parameter.

b) Read Pad message - In order to establish exactly what the state of the pad is at any point in time, the host may issue this command for as many parameters as it is interested in. This message has the same format as the Set Pad message. Following the message identifier, pairs of references to parameters are provided with a dummy value of zero for each one. As many of the parameters may be read, and in any order. The pad fills in these dummy values with its current settings, changes the message identifier to a Parameter Indication message, and returns the data to the host.

c) Set and Read Pad message - This command has the same format as the Set Pad message except for the different value of the message identifier. The effect of this message is to cause the pad to set those parameters which are mentioned in the command to the given values, and then return a Parameter Indication message to the host to show the state of those parameters which were referenced. This is the recommended way in which the host should alter the settings in the pad so the host can check that the desired effect has been achieved at the pad.

d) Parameter Indication message - This returns the values of the parameters in the pad to the host in response to either a Read Pad message or a Set and Read Pad message.

e) Invitation to Clear message - When the host wishes to terminate the virtual circuit to the pad, it will send this command. On receiving an Invitation to Clear message, the pad program must send the host a CLEAR REQUEST packet before terminating. Normally the host will send this command when the pad program notifies it that it would like to terminate the session (by sending a logout command or control-D).

f) Error Pad message - If the host finds that the pad has sent it a message that it cannot understand, it will respond with an Error Pad message. The message identifier is followed by either one or two octets describing the error.

Spad Listing

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/termio.h>
#include <sys/ttold.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include "x25/extras.h"

#define TRUE                1
#define FALSE               0
#define FAIL                (-1)
#define BUFSIZE             128
#define CLEN                128
#define TITLE               "*** SPAD-Simple Packet Assembler/Disassember ***\n"
#define PROMPT              '*'
#define ADDRESS_LENGTH      14
#define PID_LENGTH          8
#define CUD_LENGTH          12
#define NPARS               22
#define PARAM(n)            param[n-1]
#define BIT_ISSET(c,d)      ((c >> d) & 1)
#define IDLE_SEC(c)         (c / 20)
#define IDLE_USEC(c)        ((c % 20) * 50000)
#define MESSAGE_CODE        data[0]
#define ERROR_TYPE          data[1]
#define INVAL_MSG_CODE      data[2]

/* x3 parameters */
#define X3_PAD_RECALL       PARAM(1)
#define X3_ECHO             PARAM(2)
#define X3_FORWARD_DATA     PARAM(3)
#define X3_IDLE_TIMER       PARAM(4)
```

```
/* special characters */
#define ETX             3
#define EOT             4
#define ENQ             5
#define ACK             6
#define BEL             7
#define HT              9
#define LF              10
#define VT              11
#define FF              12
#define CR              13
#define DLE             16
#define DC2             18
#define CAN             24
#define ESC             27
#define DEL             127

/* pad message codes */
#define PAD_INDICATE    0
#define PAD_CLEAR       1
#define PAD_SET         2
#define PAD_BREAK       3
#define PAD_READ        4
#define PAD_ERROR       5
#define PAD_SETREAD     6
#define PAD_RESELECT    7

/* pad error messages */
#define E_TOO_SHORT     0
#define E_BAD_MSG_CODE  1
#define E_BAD_PARAM     2
#define E_MALFORMED     3
#define E_UNSOLICITED   4
#define E_MSG_TOO_LONG  5
```

```
char combuf[CLEN], padbuf[BUFSIZE], tokbuf[CLEN];
char *pb = padbuf, *cb = combuf;
int sd;
int nready;
int len;
int tty_filedes;
int command_mode = TRUE;
int call_in_progress = FALSE;
fd_set readfds;
struct addr_type connection;
struct termio raw, normal, *arg;
struct timeval timeout;
unsigned char *data;
unsigned int param[NPARS];
extern int ermo;


void tty_open(term)
char *term;
{
   tty_filedes = open(term, O_RDWR);
}


void tty_close()   /* restore the original terminal settings */
{
   arg = &normal;
   ( void ) ioctl(tty_filedes, TCSETA, arg);
}
```

```
void tty_conf()        /* configure the terminal to raw mode */
{
    arg = &normal;   /* save current terminal settings */
    ( void ) ioctl(tty_filedes, TCGETA, arg);
    arg = &raw;        /* get extra copy for raw mode */
    ( void ) ioctl(tty_filedes, TCGETA, arg);

    /* set non-canonical, no local echo, no interrupt signals */
    raw.c_lflag &= ~(ICANON | ECHO | ISIG);
    raw.c_cc[VMIN] = 1;  /* wait for a single character to be typed */

    ( void ) ioctl(tty_filedes, TCSETA, arg);
}


void init_x3_params()  /* default settings */
{
    PARAM(1) = 1;          /* use DLE to escape to command mode */
    PARAM(2) = 1;          /* echo */
    PARAM(3) = 126;        /* transmit on any non-alphanumeric character */
    PARAM(4) = 0;          /* no timeout, no automatic forwarding */
    PARAM(5) = 1;          /* flow control with XON and XOFF */
    PARAM(6) = 5;          /* transmit all pad service signals */
    PARAM(7) = 2;          /* send reset packet to reset virtual call */
    PARAM(8) = 0;          /* normal data delivery */
    PARAM(9) = 0;          /* no padding after carriage return */
    PARAM(10) = 0;         /* no line folding */
    PARAM(11) = 14;        /* terminal speed is 9600 bits/sec */
    PARAM(12) = 1;         /* use XON and XOFF for flow control */
    PARAM(13) = 4;         /* insert LF after each CR */
    PARAM(14) = 0;         /* no padding after LF */
    PARAM(15) = 0;         /* do not allow editing */
    PARAM(16) = 127;       /* character delete character is DEL */
    PARAM(17) = 21;        /* line delete character is control-U */
    PARAM(18) = 18;        /* line display character is control-R */
    PARAM(19) = 0;
```

```
    PARAM(20) = 0;
    PARAM(21) = 0;
    PARAM(22) = 0;
}


void init_buffers()        /* set up buffers for spad */
{
    bzero(combuf, sizeof(combuf));
    cb = combuf;
    data = (unsigned char *) malloc(BUFSIZE);
}


void scr(s)          /* send a string to the screen */
char *s;
{
    ( void ) write(tty_filedes, s, strlen(s));
}


void outc(c)             /* send a character to the screen */
char c;
{
    ( void ) write(tty_filedes, &c, 1);
}
```

```
int gettoken(s, t)      /* get a token from the command line */
char *s, *t;
{
    static char *tk;
    register char *q = t;

    if (s != NULL) {
        bzero(tokbuf, 80);
        bcopy(s, tokbuf, strlen(s));
        tk = tokbuf;
    }

    while (*tk == ' ') tk++; /* leading spaces */

    while (*tk != ' ' && *tk != '\0')
        *q++ = *tk++;
    *q = '\0';
    return(strlen(t) > 0 ? TRUE : FAIL);
}


void pad_help()      /* display a help screen */
{
    scr(TITLE);
    scr("    -----          -----------\n");
    scr("    TOPIC      DESCRIPTION\n");
    scr("    -----          -----------\n");
    scr("    c          Establish call to a specific address\n");
    scr("                   format: c address [-d protocol id] [call user data]\n");
    scr("    clr        Send clear packet and exit\n");
    scr("    help, ?    List available commands\n");
    scr("    int        Send an interrupt packet\n");
    scr("    par        Display specified X.3 parameter or all parameters\n");
    scr("                   format: par [param]\n");
    scr("    quit       Quit the SPAD program\n");
    scr("    reset      Send a reset packet\n");
```

```
    scr("    set         Set and display X.3 parameter\n");
    scr("                    format: set param newvalue\n");
    scr("    stat        Request status of the virtual call\n");
    scr("    ^P          Enter command mode from data transfer mode\n");
}


void pad_statistics()       /* display call statistics */
{
    struct stat_type s;
    char str[80];

    if (!call_in_progress)
        scr("STAT: virtual call not established\n");
    else {
        ( void ) ioctl(sd, IOGETSTAT, &s);  /* get statistics */
        bzero(str, 80);
        bcopy("address: ", str, 9);
        bcopy(s.x25_address, str + 9, ADDRESS_LENGTH);
        strcat(str, "\n");
        scr(str);

        bzero(str, 80);
        bcopy("protocol id: ", str, 13);
        bcopy(s.x25_protocol_id, str + 13, PID_LENGTH);
        strcat(str, "\n");
        scr(str);

        bzero(str, 80);
        bcopy("call user data: ", str, 16);
        bcopy(s.x25_call_user_data, str + 16, CUD_LENGTH);
        strcat(str, "\n");
        scr(str);

        sprintf(str, "time elapsed: %d\n", s.time_elapsed);
        scr(str);
```

```
        sprintf(str, "packets in: %d\n", s.packets_in);
        scr(str);

        sprintf(str, "packets out: %d\n", s.packets_out);
        scr(str);
    }
}


void pad_clear()      /* clear a virtual call and exit */
{
    if (!call_in_progress)
        scr("ERROR: no call in progress\n");
    else {
        close(sd);
        scr("Call cleared\n");
        tty_close();
        exit(0);
    }
}


void pad_interrupt()        /* send an X.25 interrupt packet */
{
    ( void ) ioctl(sd, IOX25INT, 0);
}


void pad_reset()      /* send an X.25 reset packet */
{
    ( void ) ioctl(sd, IOX25RESET, 0);
}
```

```
void pad_quit()          /* quit the pad program */
{
   if (call_in_progress)
      scr("ERROR: virtual call in progress\n");
   else {
      close(sd);
      tty_close();
      exit(0);
   }
}


void pad_showparam()       /* display the current X.3 settings */
{
   char the_param[80], str[80];
   int c;

   bzero(the_param, sizeof(the_param));
   if (gettoken((char *)0, the_param) == FAIL) {
      scr("PAR ");
      for (c = 1; c < 22;  c++) {
         sprintf(str,"%d:%d, ", c, PARAM(c));
         scr(str);
      }
      sprintf(str,"%d:%d\n", c, PARAM(c));
      scr(str);
      return;
   }
   else {
      c = atoi(the_param);
      if (c < 1 || c > 22) {
         scr("ERROR: invalid parameter\n");
         return;
      }
      sprintf(str,"PAR %d:%d\n",c,PARAM(c));
      scr(str);
```

```
        return;
    }
}


void pad_setparam()        /* set new values for X.3 parameters */
{
    char str[80];
    char the_param[80], value[80];
    int p, v;

    if (gettoken((char *)0, the_param) == FAIL) {
        scr("ERROR: X.3 parameter number expected\n");
        return;
    }

    p = atoi(the_param);
    if (p < 1 || p > 22) {
        scr("ERROR: invalid parameter\n");
        return;
    }

    bzero(value, sizeof(value));
    if (gettoken((char *)0, value) == FAIL) {
        scr("ERROR: new value expected\n");
        return;
    }

    v = atoi(value);
    if (v < 0 || v > 255) {
        scr("ERROR: invalid parameter\n");
        return;
    }

    PARAM(p) = v;
    sprintf(str,"PAR %d:%d\n",p,PARAM(p));
```

```
    scr(str);
    return;
}


void pad_call()        /* establish an X.25 virtual call */
{
    char str[80], token[80], *p;
    char address[ADDRESS_LENGTH];
    char protocol_id[PID_LENGTH];
    char call_user_data[CUD_LENGTH];

    if (call_in_progress) {
        scr("ERROR: call in progress\n");
        return;
    }

    /* look for the address */
    bzero(address, sizeof(address));
    if (gettoken((char *)0, address) == FAIL) {
        scr("ERROR: address expected\n");
        return;
    }

    /* get the next token */
    gettoken((char *)0, token);
    if (!strcmp(token, "-d")) {
        if (gettoken((char *)0, protocol_id) == FAIL) {
            scr("ERROR: protocol identifier expected\n");
            return;
        }

        /* get the call user data field */
        gettoken((char *)0, call_user_data);
    }
    else {
```

```
        /* set the protocol id to 0 */
        bcopy("01000000", protocol_id, 8);
        strcpy(call_user_data, token);
    }

    bzero(str, sizeof(str));  /* print out what spad's up to */
    strcpy(str, "calling ");
    p = combuf;
    while (*p++ != 'c');
    while (*p++ == ' ');
    strcat(str, --p);
    strcat(str, "\n");
    scr(str);


    /* set up the X.25 call characteristics */
    bzero(&connection, sizeof(struct addr_type));
    bcopy(address, connection.address, ADDRESS_LENGTH);
    bcopy(protocol_id, connection.protocol_id, PID_LENGTH);
    bcopy(call_user_data, connection.call_user_data, CUD_LENGTH);

    /* create an X.25 socket in data mode */
    if ((sd = socket(AF_X25, SOCK_STREAM, 0)) == FAIL) {
        scr("ERROR: cannot open\n");
        return;
    }

    /* make the X.25 call */
    if (connect(sd, &connection,sizeof(struct addr_type))==FAIL) {
        scr("ERROR: cannot connect\n");
        close(sd);
        return;
    }
    scr("COM: virtual call established\n");
    call_in_progress = TRUE;
}
```

```
void pad_recall()     /* recall pad to command mode from data transfer mode */
{
   if (!command_mode) {
      outc(LF);
      outc(PROMPT);
      bzero(combuf, sizeof(combuf));
      cb = combuf;
      command_mode = TRUE;
   }
   return;
}


void flush_pad_buffer()       /* transmit pad buffer */
{
   if (pb - padbuf > 0) {
      ( void ) write(sd, padbuf, pb - padbuf);
      pb = padbuf;
   }
   return;
}


void add_to_padbuf(c)         /* insert a character into pad buffer */
unsigned char c;
{
   if ((pb - padbuf) == BUFSIZE)
      flush_pad_buffer();
   *pb++ = c;
}
```

```
void handle_ttchar_in()         /* handle typed character */
{
    unsigned char c;
    char token[80];

    ( void ) read(0, &c, 1);     /* read the character */

    /* check for pad recall character */
    if (X3_PAD_RECALL == 1 && c == DLE) {
        pad_recall();
        return;
    }
    else if (X3_PAD_RECALL == c && c >= 32 && c <= 126) {
        pad_recall();
        return;
    }

    /* check to see if the character should be echoed */
    if (command_mode || X3_ECHO == TRUE)
        outc(c);

    /* check for data forwarding signal */
    if (!command_mode) {
        if (X3_FORWARD_DATA == 0) {
            add_to_padbuf(c);
            return;
        }
        if (BIT_ISSET(X3_FORWARD_DATA, 0) && ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') || (c >= '0' && c <= '9'))) {
            add_to_padbuf(c);
            flush_pad_buffer();
            return;
        }
        if (BIT_ISSET(X3_FORWARD_DATA, 1) && c == LF) { /* CR mapped to LF */
            add_to_padbuf(c);
            flush_pad_buffer();
```

```
        return;
    }
    if (BIT_ISSET(X3_FORWARD_DATA, 2) &&
            (c == ESC || c == BEL || c == ENQ || c == ACK)) {
        add_to_padbuf(c);
        flush_pad_buffer();
        return;
    }
    if (BIT_ISSET(X3_FORWARD_DATA, 3) &&
            (c == DEL || c == CAN || c == DC2)) {
        add_to_padbuf(c);
        flush_pad_buffer();
        return;
    }
    if (BIT_ISSET(X3_FORWARD_DATA, 4) &&
            (c == ETX || c == EOT)) {
        add_to_padbuf(c);
        flush_pad_buffer();
        return;
    }
    if (BIT_ISSET(X3_FORWARD_DATA, 5) &&
            (c == HT || c == LF || c == VT || c == FF)) {
        add_to_padbuf(c);
        flush_pad_buffer();
        return;
    }
    if (BIT_ISSET(X3_FORWARD_DATA, 6) && c >= 0 && c <= 31) {
        add_to_padbuf(c);
        flush_pad_buffer();
        return;
    }
}
```

```
if (command_mode) {
   if (c == LF) {     /* end of command input - interprete command */

      if (strlen(combuf) == 0) {     /* no command typed */
         if (call_in_progress)
            command_mode = FALSE;
         else {
            outc(PROMPT);
            bzero(combuf, sizeof(combuf));
            cb = combuf;
         }
         return;
      }

      /* get the first token to identify the command */
      gettoken(combuf, token);

      if (!strcmp(token, "c"))
         pad_call();
      else if (!strcmp(token, "clr"))
         pad_clear();
      else if (!strcmp(token, "help"))
         pad_help();
      else if (!strcmp(token, "?"))
         pad_help();
      else if (!strcmp(token, "int"))
         pad_interrupt();
      else if (!strcmp(token, "par"))
         pad_showparam();
      else if (!strcmp(token, "quit"))
         pad_quit();
      else if (!strcmp(token, "reset"))
         pad_reset();
      else if (!strcmp(token, "set"))
         pad_setparam();
      else if (!strcmp(token, "stat"))
```

```
            pad_statistics();
        else
            scr("ERROR: command not found\n");

        /* tidy up after executing command */
        bzero(combuf, sizeof(combuf));
        cb = combuf;
        if (call_in_progress)
            command_mode = FALSE;
        else
            outc(PROMPT);
        return;
    }

    /* add the character to the partially formed command */
    *cb++ = c;
    if ((cb - combuf) > sizeof(combuf)) {
        cb--;
        scr("ERROR: command string too long\n");
    }

    return;
  }

  /* put the character in the pad buffer */
  add_to_padbuf(c);
}


void handle_qualified_data(nbytes)    /* interprete instructions from host */
int nbytes;
{
  register int c;
  char str[80];

  switch(MESSAGE_CODE) {
```

```
case PAD_CLEAR:
   /* clear the current X.25 virtual call */
   scr("CLEAR PAD: received invitation to clear\n");
   pad_clear();
   break;
case PAD_SET:
   /* set the X.3 parameters to the new values given */
   for (c = 1; c < nbytes; c += 2)
      PARAM(data[c]) = (unsigned int) data[c+1];
   break;
case PAD_BREAK:
   break;
case PAD_READ:
   /* return the X.3 pad parameters to the requesting host */
   for (c = 1; c < nbytes; c += 2)
      data[c+1] = PARAM(data[c]);
   MESSAGE_CODE = PAD_INDICATE;
   ( void ) ioctl(sd, IOSETQBIT, 1);
   ( void ) write(sd, data, nbytes);
   break;
case PAD_ERROR:
   /* interprete the error and print out a meaningful message */
   switch(ERROR_TYPE) {
   case E_TOO_SHORT:
      scr("PAD ERROR: Message contained less than eight bits\n");
      break;
   case E_BAD_MSG_CODE:
      sprintf(str, "PAD ERROR: Unrecognized message code (%d)\n",
                        INVAL_MSG_CODE);
      scr(str);
      break;
   case E_BAD_PARAM:
      sprintf(str, "PAD ERROR: Incorrect or invalid parameter field (%d)\n",
                        INVAL_MSG_CODE);
      scr(str);
      break;
```

```
        case E_MALFORMED:
            sprintf(str, "PAD ERROR: Integral number of octets expected (%d)\n",
                                INVAL_MSG_CODE);
            scr(str);
            break;
        case E_UNSOLICITED:
            sprintf(str, "PAD ERROR: Received unsolicited message (%d)\n",
                                INVAL_MSG_CODE);
            scr(str);
            break;
        case E_MSG_TOO_LONG:
            sprintf(str, "PAD ERROR: Message too long (%d)\n", INVAL_MSG_CODE);
            scr(str);
            break;
        }
        break;
    case PAD_SETREAD:
        /* set the X.3 pad parameters */
        for (c = 1; c < nbytes; c += 2)
            PARAM(data[c]) = (unsigned int) data[c+1];

        /* return the X.3 parameters as requested */
        MESSAGE_CODE = PAD_INDICATE;
        ( void ) ioctl(sd, IOSETQBIT, 1);
        ( void ) write(sd, data, nbytes);
        break;
    case PAD_RESELECT:
        break;
    }

    return;
}
```

```
void handle_data_in()       /* get data sent by the host */
{
    int qualified = FALSE;

    ( void ) ioctl(sd, IOGETQBIT, &qualified);      /* is data qualified? */
    bzero(data, sizeof(data));
    len = read(sd, &data, BUFSIZE);
    switch(len) {
    case -1:
        perror((char *)0);          /* read failed */
        if (errno == CALL_CLEARED) {
            close(sd);
            tty_close();
            exit(0);
        }
        return;
    case 0:
        close(sd);              /* remote end closed down */
        tty_close();
        exit(0);
    default:
        if (qualified)
            /* handle qualified data (instructions) separately */
            handle_qualified_data(len);
        else
            /* write normal data on the screen */
            ( void ) write(tty_filedes, data, len);
        break;
    }
}
```

```
main()
{
    tty_open("/dev/tty");
    tty_conf();
    init_x3_params();
    init_buffers();
    scr(TITLE);
    outc(PROMPT);

    for(;;) {
        FD_ZERO(&readfds);
        FD_SET(0, &readfds);
        if (call_in_progress)
            FD_SET(sd, &readfds);

        /* check for automatic forwarding given by X.3 parameter 4 */
        if (X3_IDLE_TIMER > 0 && X3_FORWARD_DATA == 0) {
            timeout.tv_sec = IDLE_SEC(X3_IDLE_TIMER);
            timeout.tv_usec = IDLE_USEC(X3_IDLE_TIMER);
        }
        else {
            timeout.tv_sec = 1;
            timeout.tv_usec = 0;
        }

        nready = select(FD_SETSIZE, &readfds, (fd_set *) 0,
          (fd_set *) 0, &timeout);

        if (nready == 0) {        /* timed out */
            if (X3_IDLE_TIMER > 0 && X3_FORWARD_DATA == 0)
                flush_pad_buffer();
            continue;
        }

        if (nready == FAIL) {
            if (errno == EINTR)    /* error caused by interrupt */
```

```
            continue;
        else {
            scr("ERROR: internal error\n");     /* unknown cause */
            if (call_in_progress)
                close(sd);
            tty_close();
            exit(1);
        }
    }

    /* character has arrived from stdin */
    if (FD_ISSET(0, &readfds))
        handle_ttchar_in();

    /* data has arrived from X.25 */
    if (call_in_progress)
        if (FD_ISSET(sd, &readfds))
            handle_data_in();
  }
}
```

5.8.2 Example 2 - Caser

The following program accepts an incoming X.25 virtual call. It has been written in data mode to accept data from the remote host, inverse the case of each octet and return the modified data.

```
#include <stdio.h>
#include <sys/types.h>
#include <ctype.h>
#include <sys/time.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include "x25/extras.h"

#define TRUE        1
#define FALSE       0
#define FAIL        (-1)
#define ADDRESS     "63000064"
#define BUFSIZE     1024

int sd, nsd;
int len, size, nready;
fd_set readfds;
struct timeval timeout;
struct stat_type s;
struct addr_type connection1, connection2;
unsigned char *buf;
extern errno;
```

```
void chcase(str, l)        /* invert the case */
char *str;
int l;
{
    char *p;
    register int count;

    p = str;
    for (count=0; count < l; count++) {
        if (islower(*p))
            *p = toupper(*p);
        else if (isupper(*p))
            *p = tolower(*p);
        p++;
    }
}


main()  /* case inversion using X.25 socket library */
{
    buf = (unsigned char *) malloc(BUFSIZE);  /* buffer to contain data */

    /*
     *  Set up parameters for a call to address 63000064, subaddress 10,
     *  and ask the host that any calls containing "case" in the call
     *  user data field of the call request packet be re-routed to local
     *  port 2001.
     */

    strcpy(connection1.address, ADDRESS);
    bcopy("10", connection1.subaddress, 2);
    strcpy(connection1.call_user_data, "case");
    strcpy(connection1.mask, "000011111111");
    connection1.port_number = 2001;
```

```
/* create an X.25 socket for data mode transfer */
if ((sd = socket(AF_X25, SOCK_STREAM, 0)) == FAIL) {
    perror("socket");
    exit(1);
}

/* send our petition */
if (bind(sd, &connection1, sizeof(struct addr_type)) == FAIL) {
    perror("bind");
    close(sd);
    exit(1);
}

/* set up a backlog of 5 requests */
if (listen(sd, 5) == FAIL) {
    perror("listen");
    close(sd);
    exit(1);
}

/* hang around for an incoming call */
if ((nsd = accept(sd, &connection2, &size)) == FAIL) {
    perror("accept");
    close(sd);
    exit(1);
}

printf("connection established\n");
printf("address: %s\n",connection2.address);
printf("subaddress: %c%c\n",connection2.subaddress[0],connection2.subaddress[1]);
printf("protocol_id: %s\n",connection2.protocol_id);
printf("call_user_data: %s\n",connection2.call_user_data);
printf("port number = %d\n",connection2.port_number);
```

```
for (;;) {
    FD_ZERO(&readfds);
    FD_SET(nsd, &readfds);
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;

    nready = select(FD_SETSIZE, &readfds, (fd_set *) 0,
      (fd_set *) 0, &timeout);

    if (nready == FAIL)
        if (errno == EINTR)    /* error caused by interrupt */
            continue;
        else {
            perror("select");  /* unknown cause */
            close(nsd);
            close(sd);
            exit(1);
        }

    if (!FD_ISSET(nsd, &readfds))     /* timed out */
        continue;

    len = read(nsd, &buf, BUFSIZE);
    switch(len) {
    case -1:
        perror("read");        /* read failed */
        close(nsd);
        close(sd);
        exit(1);
    case 0:
        close(nsd);         /* other end closed down */
        close(sd);
        exit(0);
    default:
        if (!strncmp(buf, "stat", 4)) {
            ioctl(nsd, IOGETSTAT, &s);
```

```
            printf("\nstats\n");
            printf("-----\n");
            printf("address = %s\n", s.x25_address);
            printf("subaddress = %c%c\n", s.x25_subaddress[0], s.x25_subaddress[1]);
            printf("protocol_id = %s\n", s.x25_protocol_id);
            printf("call_user_data = %s\n", s.x25_call_user_data);
            printf("port number = %d\n", s.tcp_port_number);
            printf("time elapsed = %d\n", s.time_elapsed);
            printf("packets in = %d\n", s.packets_in);
            printf("packets out = %d\n", s.packets_out);
        }

        /* invert case and return the result */
        chcase(buf, len);
        if (write(nsd, buf, len) == FAIL) {
            perror("write");
            close(nsd);
            close(sd);
            exit(1);
        }
    }
  }
}
```

5.8.3 Example 3 - Echoer

The following program accepts an incoming X.25 virtual call. It has been written in packet mode to accept X.25 formatted packets from the remote host and echo back those packets which contain data.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include "x25/extras.h"

#define TRUE        1
#define FALSE       0
#define FAIL        (-1)
#define ADDRESS     "63000064"
#define GRAIN       1024

int sd, nsd;
int size, nready;
int len;
struct stat_type s;
struct addr_type connection1, connection2;
fd_set readfds;
struct timeval timeout;
unsigned char packet[PKT_SIZE];
extern errno;
```

```
main() /* echo incoming characters using X.25 socket library */
{
    /*
     * Set up parameters for a call to address 63000064, subaddress 10,
     * and ask the host that any calls containing "echo" in the call
     * user data field of the call request packet be re-routed to local
     * port 2000.
     */

    strcpy(connection1.address, ADDRESS);
    bcopy("10", connection1.subaddress, 2);
    strcpy(connection1.call_user_data, "echo");
    strcpy(connection1.mask, "000011111111");
    connection1.port_number = 2000;

    /* create an X.25 socket for packet mode transfer */
    if ((sd = socket(AF_X25, SOCK_RAW, 0)) == FAIL) {
        perror("socket");
        exit(1);
    }

    /* send our petition */
    if (bind(sd, &connection1, sizeof(struct addr_type)) == FAIL) {
        perror("bind");
        close(sd);
        exit(1);
    }

    /* set up a backlog of 5 requests */
    if (listen(sd, 5) == FAIL) {
        perror("listen");
        close(sd);
        exit(1);
    }
```

```
/* hand around for an incoming call */
if ((nsd = accept(sd, &connection2, &size)) == FAIL) {
    perror("accept");
    close(sd);
    exit(1);
}

printf("Incoming call!\n");
printf("address: %s\n",connection2.address);
printf("subaddress: %c%c\n",connection2.subaddress[0],connection2.subaddress[1]);
printf("protocol_id: %s\n",connection2.protocol_id);
printf("call_user_data: %s\n",connection2.call_user_data);
printf("port number = %d\n",connection2.port_number);

/* read an incoming X.25 packet - we expect a call request */
bzero(packet, PKT_SIZE);
len = read(nsd, packet, PKT_SIZE);
switch(len) {
case -1:
    perror("read");         /* read failed */
    close(nsd);
    close(sd);
    exit(1);
case 0:
    close(nsd);          /* other end closed down */
    close(sd);
    exit(0);
default:
    if (packet[2] != CALL_REQUEST) {
        printf("bad packet received!\n");
        close(nsd);
        close(sd);
        exit(1);
    }
}
```

```
/* accept the incoming X.25 call */
bzero(packet, PKT_SIZE);
packet[2] = CALL_ACCEPT;
if ((len = write(nsd, packet, 5)) == FAIL) {
    perror("write");
    close(nsd);
    close(sd);
    exit(1);
}


for (;;) {
    FD_ZERO(&readfds);
    FD_SET(nsd, &readfds);
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;

    nready = select(FD_SETSIZE, &readfds, (fd_set *) 0,
      (fd_set *) 0, &timeout);

    if (nready == FAIL) {
        if (errno == EINTR)    /* error caused by interrupt */
            continue;
        else {
            perror("select");  /* unknown cause */
            close(nsd);
            close(sd);
            exit(1);
        }
    }

    if (!FD_ISSET(nsd, &readfds))     /* timed out */
        continue;
```

```
/* read an X.25 packet */
len = read(nsd, packet, PKT_SIZE);
switch(len) {
case -1:
    perror("read");        /* read failed */
    close(nsd);
    close(sd);
    exit(1);
case 0:
    close(nsd);         /* other end closed down */
    close(sd);
    exit(0);
default:
    break;
}


if (packet[2] == CLEAR_REQUEST)
    break;

if (packet[2] == INTERRUPT) {
    printf("Interrupt received - sending confirmation\n");
    bzero(packet, PKT_SIZE);
    packet[2] = INTER_CONFIRMATION;
    if (write(nsd, packet, 3) == FAIL) {
        perror("write");
        close(nsd);
        close(sd);
        exit(1);
    }
    continue;
}

if (packet[2] == RESET_REQUEST) {
    printf("Reset received - sending confirmation\n");
    bzero(packet, PKT_SIZE);
```

```
            packet[2] = RESET_CONFIRMATION;
            if (write(nsd, packet, 4) == FAIL) {
                perror("write");
                close(nsd);
                close(sd);
                exit(1);
            }
            continue;
        }

        /* packet is a data packet, so just write back what was read */
        packet[0] = packet[1] = packet[2] = 0;
        if (write(nsd, packet, len) == FAIL) {
            perror("write");
            close(nsd);
            close(sd);
            exit(1);
        }

        if (!bcmp(packet+3,"stat",4)) {
            ioctl(nsd, IOGETSTAT, &s);
            printf("\nstats\n");
            printf("-----\n");
            printf("address = %s\n", s.x25_address);
            printf("subaddress = %c%c\n", s.x25_subaddress[0], s.x25_subaddress[1]);
            printf("protocol_id = %s\n", s.x25_protocol_id);
            printf("call_user_data = %s\n", s.x25_call_user_data);
            printf("port number = %d\n", s.tcp_port_number);
            printf("time elapsed = %d\n", s.time_elapsed);
            printf("packets in = %d\n", s.packets_in);
            printf("packets out = %d\n", s.packets_out);
        }
    }


bzero(packet, PKT_SIZE);
```

```
    packet[2] = CLEAR_CONFIRMATION;
    if (write(nsd, packet, 5) == FAIL) {
        perror("write");
        close(nsd);
        close(sd);
        exit(1);
    }

    close(sd);
    close(nsd);
}
```

5.8.4 The "extras.h" Header File

The following section of 'C' program definitions may be found in the header file "extras.h". The user must include this file into any programs which use the X.25 library.

```
#include "x25errno.h"

/*
 * error types
 */

#define E_NO_SUCH_HOST                  90
#define E_UNKNOWN_DESCRIPTOR            91
#define E_NO_AVAILABLE_CONNECTIONS      92
#define E_INVALID                       93
#define E_SERVER_DOWN                   94
#define E_IN_USE                        95
#define E_NOT_BOUND                     96
#define E_NOT_ESTABLISHED               97
#define E_BIND_FAILED                   98
#define E_CLOSE_FAILED                  99
#define E_CALL_RESET                    100
#define E_BAD_MODE                      101
#define E_INTER_CONFIRMED               102
#define E_RESET_CONFIRMED               103


/*
 * ioctl options
 */

#define IOGETSTAT                       0
#define IOGETMODE                       1
#define IOSETMODE                       2
#define IOGETQBIT                       3
#define IOSETQBIT                       4
```

```
#define IOX25INT                    5
#define IOX25RESET                  6

/*
 * X.25 facilities ioctl options
 */

#define FAC_PACKETSIZE              7  /* packet size negotiation */
#define FAC_WINDOWSIZE              8  /* window size negotiation */
#define FAC_TC_NEG                  9  /* throughput class negotiation */
#define FAC_CUG_BASIC               10 /* closed user group selection */
#define FAC_CUG_EXTENDED            11 /* closed user group selection */
#define FAC_CUGOA_BASIC             12 /* cug outgoing access selection */
#define FAC_CUGOA_EXTENDED          13 /* cug outgoing access selection */
#define FAC_CUG_BILATERAL           14 /* bilateral closed user group selection */
#define FAC_FS_RESTRICTED           15 /* fast select restricted */
#define FAC_FS_UNRESTRICTED         16 /* fast select unrestricted */
#define FAC_COLLECT_CALL            17 /* reverse charging */
#define FAC_NETWORK_UID             18 /* network user identification */
#define FAC_NATIONAL_MARKER         19 /* national marker */
#define FAC_CHG_REQUEST             20 /* request charging information */
#define FAC_CHG_DISTANCE            21 /* receive distance information */
#define FAC_CHG_MONETARY            22 /* receive monetary unit information */
#define FAC_CHG_SEGMENT_COUNT       23 /* receive segment count information */
#define FAC_RPOA_BASIC              24 /* rpoa selection */
#define FAC_RPOA_EXTENDED           25 /* rpoa selection */
#define FAC_NOTIFY_CLAM             26 /* called line address modified notify */
#define FAC_NOTIFY_CRN              27 /* call redirection modification notify */

/*
 * X.25 related defines
 */

#define PKT_SIZE                    131
```

```
#define DATA_MODE               1
#define PACKET_MODE             0

/* packet types */
#define CALL_REQUEST            0x0B
#define INCOMING_CALL           CALL_REQUEST
#define CALL_ACCEPT             0x0F
#define CALL_CONNECTED          CALL_ACCEPT
#define CLEAR_REQUEST           0x13
#define CLEAR_INDICATION        CLEAR_REQUEST
#define CLEAR_CONFIRMATION      0x17
#define INTERRUPT               0x23
#define INTER_CONFIRMATION      0x27
#define RESET_REQUEST           0x1B
#define RESET_INDICATION        RESET_REQUEST
#define RESET_CONFIRMATION      0x1F
#define REST_REQUEST            0xFB
#define REST_INDICATION         REST_REQUEST
#define REST_CONFIRMATION       0xFF
#define DATA_PACKET             0x00

/*
 * miscellaneous
 */

#define socket                  supersocket
#define connect                 superconnect
#define bind                    superbind
#define listen                  superlisten
#define accept                  superaccept
#define select                  superselect
#define write                   superwrite
#define read                    superread
#define close                   superclose
#define ioctl                   superioctl
#define perror                  superperror
```

```
typedef
struct addr_type {
    char address[14];
    char subaddress[2];
    char protocol_id[8];
    char call_user_data[12];
    char mask[12];
    int port_number;
};

typedef
struct stat_type {
    int mode;
    char x25_address[14];
    char x25_subaddress[2];
    char x25_protocol_id[8];
    char x25_call_user_data[12];
    char x25_mask[12];
    int tcp_port_number;
    int time_elapsed;
    int packets_in;
    int packets_out;
};
```

CHAPTER 6

FUTURE DEVELOPMENTS AND CONCLUSIONS

The current implementation of X.25 over TCP/IP has worked well during testing but there is scope for improving the implementation by making X.25 facilities available to the user working in data mode. The Spad program described in section 5.8 can also be improved in a few areas. Suggestions for further developments are discussed below.

6.1 Addition of X.25 Facilities

X.25 offers a number of optional user facilities. Currently, the user working in packet mode can make use of these facilities because he or she is responsible for formatting data into X.25 packets. In order to use these facilities however, the user must have a thorough understanding of the way X.25 packets are structured.

For the less experienced user who must work in data mode, it is desirable to have a feature which takes care of the X.25 packet formatting and is able to insert or extract the facilities from packets. The easiest way for such a user to work with facilities would be via the ioctl() call. For example, the user would use socket() to create an X.25 socket, call ioctl() to specify the desired facility, and then call connect(). The connect() routine currently formats X.25 CALL REQUEST packets but does not handle the inclusion of facilities. However, provisions have been made within the code for facilities.

The header file "extras.h" contains a number of definitions for X.25 facilities (section 5.8.4). A user who wishes to use the closed user group facility (say) would write code along the following lines:

```
sd = socket(AF_X25, SOCK_STREAM, 0);

arg = 0x12;  /* two digit CUG number */
( void ) ioctl(sd, FAC_CUG_BASIC, &arg);

if (connect(sd, &connection,sizeof(struct addr_type)) == FAIL) {   /* etc */
```

The argument arg is an integer in this example of a class A facility, but this can be a structure containing a variable number of integers when used with class B, class C and class D facilities.

The ioctl() call represents an internal routine in the X.25 library called superioctl. The superioctl routine contains a large switch() statement which defines an action for all possible ioctl requests. The following section of code is a representative sample from the superioctl routine:

```
int superioctl(s_desc, request, arg)
int s_desc;
int request;
char *arg;
{
   struct stat_type s;
   int con;
   int param;
   int flag = FALSE;

   /* perform proper ioctl if s_desc is not one of mine */
   if ((con = which_connection(s_desc)) == FAIL)
      return(ioctl(s_desc, request, arg));

   switch(request) {
   case IOGETSTAT:
      .
      .
   case IOGETMODE:
      .
      .
   case IOSETMODE:
      .
      .
   case IOGETQBIT:
      .
      .
```

```
    case IOSETQBIT:
        .
        .
    case IOX25INT:
        .
        .
    case IOX25RESET:
        .
        .
    case FAC_PACKETSIZE:    /* X.25 facilities */
        break;
    case FAC_WINDOWSIZE:
        break;
    case FAC_TC_NEG:
        break;
        .
        .
    case FAC_COLLECT_CALL:
        break;
        .
        .
    default:
        return(E_INVALID);    /* no such request */
    }
}
```

Each facility request has been followed immediately by the break command. The setting of a facility can be done between the case statement and the break statement. The task of the superioctl routine is to record the setting of facilities and parameters so that other library routines such as the connect() call can note that a facility has been recorded and act on it. It is best to set up some global flags and fields to record facilities settings. Provision has been made for this in the following structure:

```
static struct connect_type {
   int sd;
   .
   .
   int packets_in;
   int packets_out;
   .
   .
} connection[MAX_CONNECTIONS];
```

The connect_type structure, found in the source code for the X.25 library, maintains details about every connection that is opened. To implement facilities, two things must be added to this structure. The first is an integer flag for that facility which indicates that that facility has been selected. The second is a field to contain the parameters for that facility. For class A facilities, one integer is sufficient. An array of integers may be necessary for class B, class C and class D facilities. For example, to add the (class A) closed user group facility, the following lines may be added:

```
int closed_user_group_flag;
int closed_user_group_parameter;
```

Once the facilities and their parameters have been recorded, it is up to other routines which directly handle X.25 control packets to check for the presence of these settings and add these facilities to the control packets if necessary. For example, the connect() call represents an internal routine in the X.25 library called superconnect. The user may modify the packet handling sections of code in superconnect to cater for facilities. These sections have been internally documented for the user's guidance. The following section of code is from the superconnect routine:

```
int superconnect(s_desc, connect_info, info_len)
int s_desc;
struct addr_type *connect_info;
int info_len;
{
   .
   .
   /* format the called dte address */
   .
   .
   p = packet + count + 4;

   /* insert facilities here */
   *p++ = 0;   /* facilities length is zero */

   /* set the protocol identification in the call user data field */
   .
   .
}
```

The pointer p is used to move through the X.25 packet being formatted. At the point where facilities are to be inserted (just before the protocol identification field is added), the pointer p may be used to add facilities to the packet. In the example above, no facilities are added and the length of the facilities field is zero. To add the closed user group facility, the routine may be modified as follows:

```
   /* insert facilities here */
   if (connection[con].closed_user_group_flag) {
      *p++ = 2;          /* facilities length is two */
      *p++ = 0x03;       /* facility code for closed user group */
      *p++ = 0x12        /* two digit CUG number */
      connection[con].closed_user_group_flag = FALSE;
   }

   /* set the protocol identification in the call user data field */
```

Sections of code have been documented in a similar manner in the superaccept, superread, and superclose routines for handling facilities.

6.2 Possible Improvements to Spad

The Spad program described in section 5.8.1 is a simple packet assembler/disassembler that implements a subset of the collection of X.3 parameters. There is much scope for improving this program into a more sophisticated version. Improvements can be made in three areas. Firstly, the remaining X.3 parameters which were not included in Spad can be added. The following table gives a summary of these parameters:

Parameter	Description
5	Flow control of the terminal by the pad
6	Suppression of service signals from the pad
7	Action on receipt of a Break from the terminal
8	Data delivery to the terminal
9	Padding after Carriage Return
10	Line folding
12	Flow control of the PAD by the terminal
13	Line Feed insertion after Carriage Return
14	Padding characters inserted after Line Feed
15	Editing
16	Character Delete character
17	Buffer Delete character
18	Buffer Display character

More detailed information may be obtained about these parameters from the UNIX on-line manual for PAD(1-att).

Section 5.8.1 discussed the different types of messages that are used when Spad communicates with a host computer. Two messages which have not been included in the Spad implementation are the Reselect Pad message and the Indication of Break message. These messages have been defined in the Spad program in anticipation of future inclusion.

Finally, Spad currently supports outgoing calls only. A separate program will have to be written for handling incoming calls which will combine the programming techniques

demonstrated in Spad, Caser and Echoer. Typically, the program would send a bind request to the Pyramid server containing a unique user data field and then enter an accept state where it will wait for incoming calls to be re-routed to it by the Pyramid server. After accepting an incoming call, the program would immediately simulate a login shell by sending send the string "login:" in a data packet to prompt for a usercode. A password can be obtained in the same way, verified and the normal exchanging of packets can follow.

6.3 Conclusion

This project involved the design and implementation of software for running X.25 over TCP/IP. The implementation was stimulated by the networking requirements of the Department of Computer Science at Massey University. The department has an extensive local-area network running TCP/IP but does not have a connection to a public data network. This has restricted the development of networking solutions.

An X.25 connection to a public data network is available through the Pyramid computer run by the Massey University Computer Center. This provided the motivation to develop a system that would allow users at the Department of Computer Science to write X.25 software on IP workstations. X.25 packets would be sent from the IP workstations to the Pyramid using TCP/IP. The Pyramid would then forward the X.25 packets to the X.25 network. Similarly, incoming X.25 packets to the Pyramid from the public data network would be forwarded to the IP workstation along the same TCP/IP connection.

The resulting system was comprised of an X.25 programming library and two server programs. The servers, called the Pyramid server and the Incoming Call server (ICS), operate on the Pyramid and provide X.25 services to the X.25 programming library. An outgoing X.25 virtual call must first contact the Pyramid server which forks a child process to handle the call, therefore leaving the parent copy free to accept requests from new client programs. An incoming X.25 virtual call to the Pyramid is received on subaddress 10 on the Pyramid. Attached to subaddress 10 is the ICS program, which is activated whenever an incoming call arrives. On start-up, the ICS program contacts the Pyramid server for instructions about which IP workstation to connect to.

The X.25 programming library provides a programmer on an IP workstation with a number of routines for users to perform a number of tasks such as opening and closing X.25 virtual circuits, sending and receiving data in the form of suitably formatted X.25

packets or data, and getting status information on connections. The interface to the library resembles the UNIX socket interface. To write X.25 software, a programmer must specify the AF_X25 communication domain as the first parameter to the socket() call.

Three example programs were developed to demonstrate the use of the X.25 programming library. One of these is a simple packet assembler/disassembler called Spad which consists of a user interface and uses X.3/X.28 communication protocols to allow access to X.25 packet switching networks from an asynchronous terminal.

This project has shown that implementing X.25 over TCP/IP is a suitable solution for providing users with the services necessary for developing X.25 software. It overcomes two problems. Firstly, it compensates for the absence of a direct connection to a public data network. Secondly, it overcomes the problem of security as the Computer Center does not need to issue login rights on the Pyramid computer to anyone wishing to develop X.25 software.

REFERENCES

Bleazard, G.B.
Why Packet Switching?
NCC Publications, 1979

Bleazard, G.B.
Handbook of Data Communications
NCC Publications, 1982

Braden, R., Postel, J.
RFC-1009: Requirements for Internet Gateways
Information Sciences Institute, 1987

Comer, Douglas
Internetworking with TCP/IP
Principles, Protocols and Architecture
Prentice-Hall International Inc., 1991

Digital Equipment Corporation
VAX P.S.I (Volume 1)

The International Telegraph and Telephone
Consultative Comittee (CCITT)
Volume VIII: Data Communication Network Interfaces
Recommendations X.20-X.32

Deasington, Richard J.
X.25 Explained:
Protocols for Packet Switching Networks
Ellis Horwood Limited, 1988

Hedrick, Charles.
Introduction to the Internet Protocols
1987

Hornig, C.
RFC-894: A Standard for the Transmission of IP Datagrams
over Ethernet Networks
Symbolics Cambridge Research Center, 1984

ISODE Reference Manual

Joint Network Team Ethernet Advisory Group
Implementation Details for Protocols on CSMA/CD LANs
August 1985

Marsden, Brian W.
Communication Network Protocols
Chartwell-Bratt, 1986

Martin, James
Computer Networks and Distributed Processing
Software, Techniques and Architecture
Prentice-Hall International Inc., 1981

Murphy, Michael
MSc Thesis
Massey University, 1986

Onions, J., Rose, M.
RFC-1086: ISO-TP0 Bridge between TCP and X.25
TWG, 1988

Perkins, D.
RFC-1171: The Point-to-Point Protocol for the Transmission
of Multi-Protocol Datagrams Over Point-to-Point Links
CMU, 1990

Perkins, D., Hobby, R.
RFC-1172: The Point-to-Point Protocol (PPP) Initial Configurations
CMU, 1990

Plummer, D.
RFC-826: An Ethernet Address Resolution Protocol
Symbolics Cambridge Research Center, 1982

Postel, J.
RFC-791: Internet Protocol - DARPA Internet Program
Protocol Specification
USC/Information Sciences Institute, 1981

Postel, J.
RFC-793: Transmission Control Protocol - DARPA Internet Program
Protocol Specification
USC/Information Sciences Institute, 1981

Postel, J.
RFC-879: The TCP Maximum Segment Size Option
and Related Topics
USC/Information Sciences Institute, 1983

Reynolds, J., Postel, J.
RFC-1060: Assigned Numbers
Information Sciences Institute, 1990

Rose, M., Cass, D.
RFC-1006: ISO Transport Services on Top of the TCP Version: 3
Northrop Research and Technology Center, 1987

Tanenbaum, Andrew S.
Computer Networks
Prentice-Hall International Inc., 1981