MULTI-PROCESS STRUCTURING OF X.25 SOFTWARE

by

Stephen Edward Deering

Technical Report 82-11

October 1982

MULTI-PROCESS STRUCTURING OF X.25 SOFTWARE

by

STEPHEN EDWARD DEERING

B.Sc., The University of British Columbia, 1973

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming

to the required standard

Dr. David R. Cheriton...

Gerald Neufeld.....

THE UNIVERSITY OF BRITISH COLUMBIA

October 1982

Abstract

Modern communication protocols present the software designer with problems of asynchrony, real-time response, high throughput, robust exception handling, and multi-level interfacing. An operating system which provides lightweight processes and inexpensive inter-process communication offers solutions to all of these problems. This thesis examines the use of the multi-process structuring facilities of one such operating system, Verex, to implement the protocols defined by CCITT Recommendation X.25. The success of the multi-process design is confirmed by a working implementation that has linked a Verex system to the Datapac public network for over a year.

The processes which make up the Verex X.25 software are organized into layers according to the layered definition of X.25. Within the layers, some processes take the form of finite-state machines which execute the state transitions specified in the protocol definition. Matching the structure of the software to the structure of the specification results in software which is easy to program, easy to understand, and likely to be correct.

Multi-process structuring can be applied with similar benefits to protocols other than X.25 and systems other than Verex.

Dr. David R. Cheriton

Table of Contents

List of Figures

Acknowledgements

This thesis owes its existence to the perseverence and guidance of Dr. David Cheriton. I would especially like to thank David for creating such a stimulating research environment and for leading me into the field of communication software.

Many thanks are due to Pat Boyle, John Demco, Gerald Neufeld, and Stella Atkins for their helpful comments and their fine editing skills.

I am grateful for the support of the U.B.C. Department of Computer Science and the National Sciences and Engineering Research Council of Canada.

1. Introduction

This thesis presents a design for X.25 interface software. The software is structured as a collection of processes communicating via messages. The organization of the processes reflects the architecture of the protocol layers; their internal structure is derived from state-transition descriptions of the individual layers. This design has been successfully implemented on a small timesharing system, linking it to Canada's Datapac network.

CCITT Recommendation X.25 [10] defines a multi-layer protocol for interfacing computer equipment to a packet-switched network. It has become a standard for access to public and private networks in many countries. The interconnection of those networks has made an enormous population of computers accessible at very low cost to any system which supports X.25. Unfortunately, software to support an X.25 interface places significantly greater demands on the resources and capabilities of a computer system than that required for most other peripherals. It requires greater skill on the part of the software designer to meet those demands.

Designers of software for computer communications have many choices but few guidelines for structuring and organizing their systems. The well-developed techniques for structuring sequential programs do not address many of the requirements of modern communication protocols, such as concurrency, time-critical response, efficient data movement, robust recovery from errors and exceptions, and multi-level interfacing with other software. Many mechanisms have been provided or proposed to solve each of these problems; little is known of how to select and combine these various techniques to most effectively translate the description of a

protocol into a working implementation. This thesis explores the use of processes and messages as structuring primitives for communication software. The evolving tools for describing protocols, such as the ISO Reference Model for Open Systems Interconnection [19] and finite state specifications of protocol entities [1], are used as blueprints for an assembly of communicating processes. The resulting software is produced quickly and models closely the protocol descriptions, contributing to its understandability and maintainability.

The abstract notions of "processes" communicating via "messages" were introduced by researchers in operating systems [6], [13] and programming languages [18] to handle the complexities of real-time, parallel programming. Traditional, procedure-based operating systems have proven awkward or unsuitable for supporting the patterns of data flow, the degree of asynchrony, and the critical timing requirements of communication software. For example, efforts to support communication protocols within the classically-structured UNIX operating system [25] have forced implementors to extend UNIX with additional interprocess communication facilities [17], [24] or discard it altogether in favour of more special-purpose systems [7]. On the other hand, most so-called "real-time" operating systems provide an ad hoc assortment of synchronizing and communicating facilities such as semaphores, signals, monitors, software interrupts, etc., each designed to solve a different problem. In contrast, the message-passing process model is a simple and unifying approach to synchronization, data transfer, concurrency, and distribution of function across processors and memories.

The process and message-passing model presented by one operating system, Verex, is described in the following chapter. Verex's small number of primitive operations provide a hospitable environment for a diversity of applications, in particular, our X.25 implementation. Chapter 3 describes the interface presented by the X.25 software to client processes within Verex, either user application programs

or higher-level protocol software. Chapter 4 is a discussion of the multi-process structuring methodology and a layer-by-layer analysis of its application to the Verex X.25 service — detailed knowledge of X.25 is assumed of the reader. In the final chapter, the software and its design are evaluated and its strengths and weaknesses examined in light of other X.25 implementations, other protocols, and other operating system environments.

2. Implementation Environment

The X.25 software has been implemented as part of an experimental operating system named Verex. Much of the design of that software has been influenced by the environment provided by Verex. This chapter briefly describes that environment.

Verex, a descendant of Thoth [14], is a portable operating system for small computers. The system is structured as a kernel and a collection of processes [22]. The kernel provides process creation and destruction, interprocess message-passing, and low-level device support. Processes provide the bulk of the operating system services, such as memory management, a file system, terminal support, interactive session management, and network access. Application programs execute as processes which obtain services by sending messages to system processes. With an appropriate selection of processes, Verex can be configured as a dedicated real-time system or as a multi-user, interactive programming environment.

Central to the design of Verex is the use of multiple processes to support concurrent and asynchronous behaviour. To make multi-process structuring attractive, the implementation guarantees that processes are cheap to create and destroy, that process switching is efficient, and that message-passing is fast. Short, fixed-length messages are sent directly to processes — there are no connections and no ports. Each process has a unique, global identifier and a single FIFO queue for incoming messages.

The kernel provides the following message-passing primitives:

Send (message, id)

sends the specified message to the process identified by **id** and suspends the sending process until a reply is received. The contents of **message** are changed by the reply.

id = Receive (message)

retrieves the first message from the incoming message queue and returns in **id** the process identification of the sender. If the queue is empty, the receiver is suspended until a message arrives.

Reply (message, id)

returns a reply message to the process identified by **id.** This enables the sender to resume execution.

Forward (message, id1, id2)

forwards a message received from process **id1** to process **id2.** The effect is as if **id1** had originally sent the message to **id2.** Figure 1 illustrates the use of these message-passing primitives between processes.

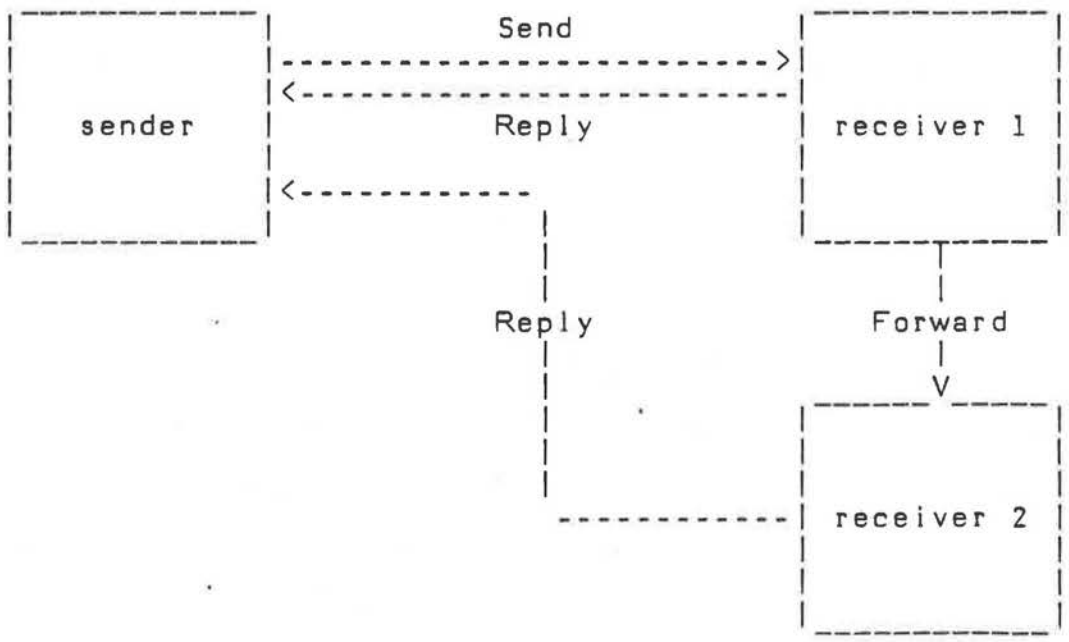

Figure 1. Message-Passing in Verex

In the current implementations of Verex, all messages are 8 words long. This is sufficient for the exchange of control information and small amounts of data. For the movement of larger blocks of data, the kernel provides the following two operations:

Transfer-from (id, n-bytes, remote-vec, local-vec)

copies **n-bytes** of data from **remote-vec** in the address space of process **id** to **local-vec** in the address space of the invoking process.

Transfer-to (id, n-bytes, remote-vec, local-vec)

similarly copies **n-bytes** of data to **remote-vec** in the address space of process **id** from **local-vec** in the address space of the invoking process. Both of these operations are permitted only when the process identified by **id** is suspended awaiting reply from the invoking process.

The provision of a separate mechanism for communicating large amounts of data is analogous to the DMA (direct memory access) facility of hardware interfaces or the "reference parameter" facility of procedure interfaces. It allows the interprocess communication mechanisms to be optimized for the amount of data being communicated.

The message-passing primitives described above support the **remote procedure call** style of interprocess communication. Sending a message and awaiting its reply is much like passing arguments to a procedure and receiving its results. However, processes that receive messages are more powerful than procedures: they can reply to the messages in any order or forward them to other processes for reply. These message facilities are often used to establish a client/server relationship between processes. A **client** process requests some service by sending a message to a **server** process and waiting for a reply. The request message contains all necessary fields to

specify the service required; the reply message contains the results of servicing the request. The client is given the simple procedure-like interface of **Send**, whereas the server can use the greater flexibility of a receiver to provide sophisticated scheduling and interleaving of services.

An example of a server-based facility is the Verex I/O system [12]. Processes obtain input and generate output by sending requests to file servers. A **file server** is any process which implements the Verex file access interface; it may provide disk storage, device access, or any other source/sink of data that can fit the file model. Clients initiate file access by requesting a file server to create an **instance** of a file meeting certain specifications. For example, a file server which provides spooled printer access may be requested to create an instance of a writeable file, or a disk file server may be asked to create an instance of a readable file corresponding to a previously written file. Once an instance is created, a client may read or write blocks of data via further requests to the server. When finished, the client sends a message asking the server to release the file instance.

File servers do not provide character-string names for their files. Instead, file names are maintained for all file servers by a single **name server** which has a well-known process identifier. The name server manages a directory of names and corresponding server information. A client can send a name to the name server and receive in reply the process identifier of the appropriate file server and some server-specific file identification. This file identification can then be passed to the file server as an argument in a "create instance" request.

A standard library of I/O procedures provides a conventional byte-stream interface (**Open, Close, Get-byte, Put-byte**) to the block-oriented message interface. No "asynchronous" I/O interface is provided whereby a process can start a read operation, continue execution for a while, and then pick up the input. Such concurrency is easily obtained in Verex by using multiple, independently executing

processes.

An important feature of the Verex environment is the support for multiple processes executing within a single address space. The set of one or more processes which reside in an address space is called a **team.** Each process on a team has a separate execution stack in the common space but shares code and global data with the other processes. This organization provides efficient support for programs which employ multiple identical processes or applications which require closely cooperating processes to share large amounts of data.

Within a team, each process is assigned a **priority.** The kernel guarantees that the execution of a process will not be preempted by any process of the same or lower priority on the same team. Thus, processes of the same priority can access shared data without danger of mutual interference. Teams as a whole also have scheduling priorities. By suitable assignment of priorities to teams and processes, real-time response to external events can be controlled and guaranteed.

Most of the Verex kernel and all of the system and application processes are written in a portable, high-level programming language named Zed [15] which is quite similar to C [21]. The X.25 software to be described is also written in Zed and is installed in a version of Verex running on a Texas Instruments 990/10 minicomputer.

3. Client Interface

The X.25 Recommendation [10] defines the interface between a host computer system and a network; it does not define the interface presented to application (client) processes within the host. The client interface depends on the nature of the host operating system and the requirements of client processes. In the design of X.25 software for Verex, the primary goal was to provide general access to X.25 virtual circuits for a variety of application programs and higher-level protocol software. It was not to be restricted to handling, say, incoming terminal connections only. A secondary goal was to provide the X.25 service via the standard Verex file access interface rather than inventing an entirely new, customized interface. If virtual circuits could be made to look like Verex "file instances", clients could take advantage of the existing library of file access procedures to handle X.25 data traffic. Adopting the same interface as other devices and files would contribute to the device-independence of application software and to a decrease in the amount of new documentation required. However, the Verex file access interface does not encompass all of the facilities of X.25 virtual circuits, such as "qualified" data or data-bearing interrupts. In order to satisfy the goal of a general X.25 service it would be necessary to supplement the file access interface with mechanisms to access these additional features. A description of the resulting client interface and discussion of the design problems encountered are presented below.

3.1 Call Handling

Access to one or more X.25 networks is provided by a single team of processes, collectively known as the **X.25 server.** The X.25 server supports the establishment of virtual circuits both by placing outgoing calls to the networks and by accepting incoming calls from the networks. These two methods of connection are provided to client processes through two different interfaces.

To place an outgoing call, a client process must first locate the X.25 server by sending to the name server a query message specifying the name of the desired network, e.g. "Datapac". If the X.25 server is present in the system and offering service on the named network, the reply from the name server contains the identifier of a process on the X.25 team. To that process the client then sends a "create instance" request message containing the following information:

— the "create instance" request code.

— an access code requesting both read and write access.

— the logical channel number to be used for the virtual circuit. This value is normally zero to indicate that any free logical channel will do.

— the address of a vector containing the destination DTE (host) address, source DTE address, optional facilities, and call user data. This information is formatted according to the specifications of the X.25 Call Request packet, excluding the first three octets.

— the length of the Call Request vector.

The first two items are standard for all Verex file servers; the last three are specific to the X.25 server.

The formatting of the Call Request packet by the client may appear to be an unnecessary burden for the client and an inappropriate mixing of protocol levels. The alternative is to define a network-independent set of connection parameters to be supplied in some standard format. An example of this approach can be found in the DoD Internet Protocol [23] which provides an abstract "quality of service" parameter to be mapped into network-specific parameters for various underlying network protocols. However, such generality was considered less important than simple but complete X.25 access for the initial design. Instead, the X.25 server takes the view that the format and contents of the Call Request packet are the concern of higher-level, end-to-end protocols; the Call Request vector is simply treated as client data to be passed untouched and unexamined to the network. Any errors in the Call Request detected by the network are reported back to the client.

The Verex library procedure **Open** takes a file name as an argument and requests the name server to provide all corresponding server-specific information required for a "create instance" request. Unfortunately, the current name server is not prepared to store the volume of information required to specify an X.25 connection. Also, it is clearly impractical to provide file names to correspond to all possible combinations of Call Request parameters, although some frequently called destinations could reasonably be given names. Due to the name server limitations, an X.25 client cannot use **Open** to establish a virtual circuit but rather must use some X.25-specific code to collect and format the Call Request information. This is not a problem with current applications but it remains to be seen whether raw X.25 virtual circuits would benefit from having names.

If the Call Request is successful and the virtual circuit is established, the client receives a reply message containing:

— a code indicating success.

— the identifier of a server process to handle all requests pertaining to the new virtual circuit. This process may or may not be the same as the one that received the "create instance" request.

— a unique virtual circuit identifier which is to be included in all subsequent requests. The server uses this identifier to detect attempts by a client to reference an earlier virtual circuit which has been cleared.

The Call Request may fail for many reasons, such as lack of free logical channels (either locally or at the destination), failure of the network or links to the network, or invalid Call Request packet format. In all cases, the reason for failure is returned to the client as the only item in the reply message. True to the X.25 definition, the server does not time out or retry Call Requests; the client can easily do so if desired.

Incoming calls are received and accepted by the X.25 server. When the server is initialized, it is given the name of a file containing a program (team) for handling incoming calls. Whenever a call comes in, the specified program is invoked with the Call Request packet as an argument. Based on the contents of the packet, the program can reject the call, invoke or notify a particular client to take responsibility for the call, or assume the role of client itself. This interpretation of incoming Call Request data by software outside of the X.25 server is consistent with the handling of outgoing calls. It yields a convenient modularity — different configurations of the system can use different call handling software with the same X.25 server — and keeps the X.25 server smaller and simpler.

The explicit invocation of a call handling program has some advantages over the common technique of having clients wait for incoming calls to arrive. No system resources such as memory or swap space are consumed by waiting clients. It avoids the issue of what to do when a call arrives and no client is waiting. Unfortunately,

incoming call service is thus available only in systems configured to support local program invocation, a relatively high-level operation requiring, for example, access to a file store.

A virtual circuit is cleared when a client sends a "release instance" request to the X.25 server. Any other client process awaiting a reply from the specified circuit is immediately returned an indication that the circuit has been cleared. The circuit may also be cleared by request of the network, request of the remote host, or failure of the link to the network.

Another cause for clearing is the disappearance of the **owner** of the virtual circuit. Ownership resides initially with the client process that established the virtual circuit; it may be transferred to another process at the owner's request. The continued existence of all current owners is checked by the X.25 server whenever a new client attempts to establish a virtual circuit. If an owner has disappeared, its circuit is cleared, freeing up a logical channel for other calls. This lazy form of resource recovery is not completely successful since the X.25 server detects call attempts only by local clients. Incoming calls are turned away by the network whenever all logical channels are in use. The X.25 server receives no notification of such events and therefore is not provoked into checking for abandoned circuits. This problem could be alleviated for the price of a timer which invokes periodic channel reclamation.

3.2 Data Transfer

Once a virtual circuit is established, clients transfer data across it using "read instance" and "write instance" requests. A "read instance" request takes the form of a message containing the request code, the virtual circuit identifier, the address of a buffer and its length. When a data packet arrives, the X.25 server transfers the data portion of the packet, preceded by a one-byte header, into the buffer. The client

then receives a reply message specifying the length of the buffer's new contents. The header byte contains two single-bit flags corresponding to the X.25 Q-bit and M-bit received with the incoming packet. For a "write instance" request, the client must provide the address and length of a similar buffer containing a header byte and the data to be transmitted. The X.25 server delays the reply to a write request until packet-level flow control allows transmission of the packet.

The passing of the Q and M bits as a header in the client's buffer reflects the view that packet "qualification" and fragmentation/reassembly are the concern of the layer of protocol above X.25. This is clearly true for the Q-bit, but not necessarily for the M-bit; the X.25 server could perform automatic fragmentation/reassembly for the case where client buffers are larger than packets. However, at the Verex file access interface, the size of the blocks exchanged between client and server is chosen for the convenience of the server, not the client, and the blocks are used to carry byte streams with no natural record boundaries. The goal of providing general X.25 access here conflicts with the desire to conform to the file access interface: the header byte provides full X.25 control to higher-level protocol software, but obstructs straightforward use of virtual circuits as byte streams insofar as the client must be aware of packet boundaries.

Many protocols impose some kind of record structure on top of communication links which are essentially byte or bit streams, for the purposes of error control and flow control. Such records also provide a convenient unit for separating control information from data. Unfortunately, when the application requires a simple byte stream, it appears that an extra layer of protocol is required to transform the record structure back into a byte stream.

An example of this phenomenon was observed when attempting to provide sophisticated terminal support across the network. The Verex terminal handling processes use the byte stream file access interface to access local terminal

interfaces. In order to use the same processes to handle network terminals, it was necessary to interpose another process whose only function was to insert/delete a header byte of zeroes in all blocks exchanged with the X.25 server, and whose only effect was to degrade terminal response time.

3.3 Interrupts and Resets

In order to exchange X.25 Interrupt packets, clients are provided with two additional, X.25-server-specific requests: "send interrupt" and "acknowledge interrupt". A "send interrupt" request message contains a single byte of data to accompany an outgoing Interrupt packet. A reply is returned when the network acknowledges the interrupt. Incoming interrupts are presented to the client as a special reply code and single byte of data in the reply message to a "read instance" request. After receiving such a reply, the client must send an "acknowledge interrupt" request to enable reception of further interrupts.

A problem arises with this scheme: interrupts become subject to the same flow control as data at the client/server interface. A client normally exerts flow control on incoming Data packets by withholding "read instance" requests. However, when no read request is outstanding, Interrupt packets cannot be received either. The provision of a separate "receive interrupt" request was considered but rejected as an unnecessary complication for those applications that either do not require flow control or perform their own using higher-level protocols — they would require an extra process dedicated to waiting for interrupts. Instead, a mechanism for asynchronously notifying the client of the Interrupt was adopted from the terminal servers' support for the "break" function. A client may nominate a "victim" process to be destroyed by the X.25 server when an Interrupt packet is received. The death of the victim is detected by another process on the client's team (normally the creator of the victim process) which can then issue a "read instance" request to pick

up the byte of interrupt data. This differs from dedicating a process to waiting for interrupts in that the victim process can serve other purposes on the client's team and is only required when the client needs asynchronous notification. This method relies on the fact that process creation and destruction are relatively inexpensive operations in Verex.

X.25 Resets can occur as a result of temporary hardware failures of network access links or software errors in either the hosts or the network. A reset can result in the loss of an unspecified and indeterminable number of packets. To guarantee a reliable end-to-end communication path, higher-level software (a "transport service") must be used to detect and retransmit lost packets. Notification of reset is insufficient for complete recovery, and when such higher-level protocols are employed, notification of reset serves little purpose. Therefore, in the current design, the X.25 server hides resets from clients. When a reset occurs, any outstanding "write instance" or "send interrupt" request is satisfied immediately with no indication of reset; an outstanding "read instance" request remains outstanding.

The decision to hide resets contradicts the goal of providing full X.25 access to clients and may be re-evaluated in light of application requirements. The ability to generate and detect resets could conceivably be used by some applications as an additional signalling method. Also, there is at least one case where notification of reset is sufficient for recovery: when the client is a virtual terminal program, a message to the human user saying "Reset has occurred" is enough to alert the user to possible packet loss. The user can then perform whatever interaction is necessary to evaluate the damage and recover.

3.4 Summary

Apart from the handling of resets just described, the primary requirement of general access to full X.25 service is satisfied by the client interface. The generality of the interface causes it to fall short of the secondary goal of compatibility with the Verex file access interface. In particular, the inability to use **Open** to create a virtual circuit and the need for clients to be aware of packet boundaries prevent the transparent use of virtual circuits as byte streams and force X.25 dependencies on the clients. Perhaps it would be more worthwhile to cast files in the image of communication circuits, rather than the other way around. In any case, partial conformance to the file access interface does allow clients to use common library routines which provide block-level access to standard "file instances".

The interface currently supports two applications: a program which emulates an X.29 Network Interface Machine [9] to provide local terminal access to remote hosts, and a matching host-side X.29 module [27] which supports access to the local host from remote terminals. These clients influenced the interface design; future applications may contribute to its evolution.

4. Server Design

Communication software, like most other software, presents the designer with a multitude of choices: how to break it into parts, how many parts to have, how to connect the parts. Unlike most software, the parts that make up communication software often must satisfy rigorous constraints on real-time response, must support a high degree of concurrency, and must provide sufficient throughput for large volumes of data. This chapter describes and justifies the choices made in the design of X.25 software for the Verex operating system. Many of the decisions are applicable to other protocols and other operating systems.

The Verex model of multiple processes interacting via messages was designed to support concurrent and real-time behaviour, precisely the kind of behaviour required of communication software. However, it is not obvious how to best exploit those multi-process facilities to produce software which meets particular performance goals and is also correct, clear, and maintainable. The methodology for multi-process structuring developed by Cheriton [13] for the Thoth operating system and subsequent experience with other (non-protocol) software in Verex have led to the following criteria for using multiple processes:

— Logical Concurrency: A program which is required to perform two or more logically concurrent activities is often best implemented as multiple processes, each a simple sequential program, rather than as a single process with intertwined control structures. Examples are applications requiring concurrent garbage collection, producer/consumer problems, and full-duplex communication handlers. This is the class of programs for which coroutines

are often suggested.

— Real Concurrency: Multiple processes can be used to achieve real concurrency, even within a single processor. For example, a file copy program, though logically sequential, can benefit from separate reader and writer processes which control overlapping operation of external devices. Recognizing parallelism in the form of multiple processes is also the first step towards exploiting multiprocessor machines and distributed systems.

— Synchronization: Programs which must handle asynchronous events, such as device handlers of all kinds, can conveniently be structured as a collection of processes, each waiting for (i.e. synchronizing with) a particular event. Where necessary, the various processes can synchronize with each other using message-passing. This is an attractive alternative to schemes involving a single process which polls for events or traps "software interrupts".

— Access Control: Access to data structures which are shared among multiple processes can be synchronized by placing the shared data under the exclusive control of a single process. Operations on the data are performed only by the managing process in response to request messages from other processes. The data are therefore safe from the dangers of concurrent access. This structure is commonly used to control access to operating system resources, where a resource is represented by a "control block" or "state descriptor" under the management of a single **server** process.

— Modularity: Clearly identified "services", such as those provided by an operating system (device access, directory lookup, etc.) or layers of protocol software, should be partitioned into separate processes even if none of the above criteria directly apply. Such partitioning simplifies understanding, maintenance, replacement (perhaps by hardware), sharing, and possible

distribution across machines.

Concurrent processes communicating via synchronized messages are vulnerable to deadlock. Cheriton in [13] discusses deadlock in the context of the Thoth operating system, which is essentially the same as Verex. He shows how deadlock can be avoided by disciplined use of the message-passing primitives, such that there are no circularities in the blocking behaviour of processes. The essential rule is that processes be ordered hierarchically with higher level processes using the **Send** primitive to pass messages downward and lower level processes using only the non-blocking **Reply** to communicate upward. It is possible to violate this rule without incurring deadlock but it then becomes much harder to demonstrate that the resulting communication structure is in fact deadlock-free.

Within the above guidelines, the number of processes should be minimized. Proliferation of processes can lead to complex and awkward interprocess communication structures as well as unnecessary overhead for process switching and message-passing. For example, several closely related, shared variables are best placed under the control of one process rather than several processes in order to reduce the amount of message-passing required to examine or manipulate them.

In addition to choosing a process structure, the designer of Verex software must also decide how to distribute the processes across address spaces. Verex allows a **team** of processes to share a common address space. The shared memory is initialized with code and data from a file and is the unit of swapping. The following considerations govern the use of teams:

— Code Sharing: Multiple identical processes can share code, and thus reduce memory requirements, by residing on the same team. A good use of this facility is for multiple servers which handle identical devices, such as terminals or disk drives.

— Data Sharing: If a large amount of data must be shared between several processes, placing the processes on the same team can eliminate much costly data copying.

— Swapping Control: For applications which require some non-swapping memory, such as DMA device servers, the program can sometimes be divided into two teams, one memory-resident and the other swappable, to reduce the amount of dedicated memory.

— Program Size Constraints: An application which is too large to fit into the address space of a small machine can sometimes be partitioned into multiple teams. A common example is a multi-pass compiler.

— Dynamic Reconfigurability: A program implemented as a single team can dynamically reconfigure its process structure using process creation and destruction. A program implemented as several teams can, in addition, dynamically reconfigure parts of its executable code using team creation and destruction. This is the basic method used to configure different versions of Verex, to handle changes in device availability, and to execute arbitrary programs.

The above criteria for process and team structuring are not hard and fast rules. In some circumstances, one rule contradicts another and tradeoffs must be made. For example, using a single process to control access to shared data can sometimes create a bottleneck which reduces concurrency. Sometimes considerations of modularity or reconfigurability demand the use of multiple teams for processes which must communicate large amounts of data. This list of guidelines does not eliminate design choices, it merely enumerates them.

The application of these criteria to the design of X.25 software has resulted in a three-layer structure composed of several processes on a single team plus a kernel-level device driver (Figure 2). At the bottom is the **frame layer** which is responsible for data framing, check sequence calculation, and manipulation of the hardware interface to the network. The frame layer is provided by a standard device driver in the Verex kernel. In the middle is the **link layer** which handles the X.25 Link Access Procedure (LAP). The link layer comprises several processes, all on the same team, some of which communicate directly with the frame layer driver. At the top is the **packet layer**, another group of processes on the same team as the link layer processes, which implements the packet-level procedures of X.25. This layer provides the client interface described in Chapter 3, and uses the services of the link layer to exchange packets with the network.

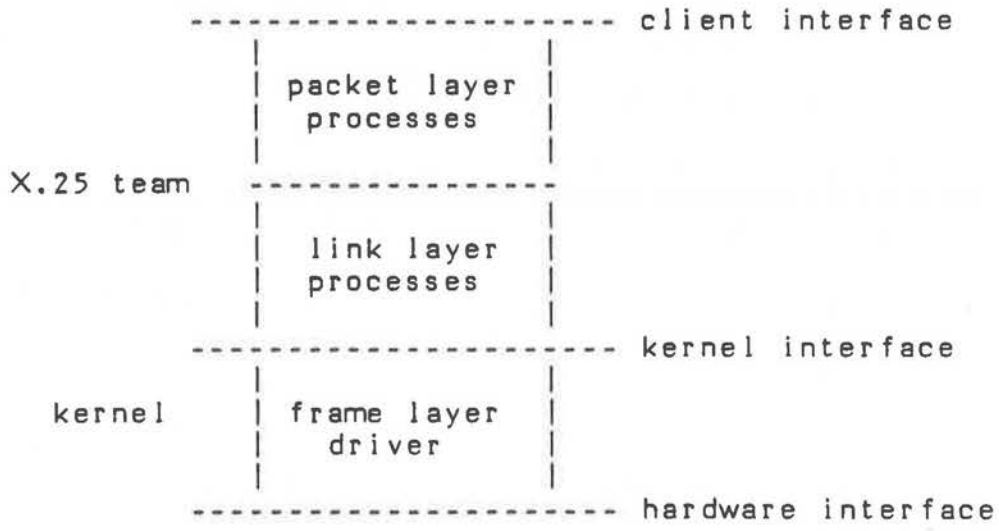

Figure 2. Layer Structure of the X.25 Software

This overall structure of the X.25 software obviously parallels the lowest three layers of the ISO Reference Model for Open Systems Interconnection [19]. This is not surprising, since many of the layering principles used in the Reference Model are similar to the above criteria for multi-process structuring. They are simply principles of modularity which can be applied with equal benefit to descriptive models, protocol

definitions, and process structures used to implement protocols. The Verex software conforms to the structure of X.25, and X.25 conforms to the ISO Reference Model.

The detailed design of each of the three layers of software is described below. The discussion illustrates the multi-process structuring methodology, showing where the above guidelines for using processes and shared memory are applied and where they are violated. The relationship of the three-layer X.25 software to the next higher layer of the ISO Reference Model, the Transport Layer, is also discussed. The design description closes with a discussion of buffer management issues which cut across all the layers.

4.1 The Frame Layer

At the bottom, communication software meets hardware. In Verex, low-level hardware access is provided by device drivers in the kernel. Only the kernel has (or should have) the appropriate level of privilege to access I/O devices and, due to the power of some device interfaces to write into arbitrary memory locations, the kernel must control I/O to maintain its own integrity. Since the kernel creates the process abstraction and implements message-passing, device drivers inside the kernel cannot themselves make use of the structuring tools of processes and messages. Therefore, it is desirable to keep device drivers small and simple, leaving as much of the work as possible to the more structured world outside the kernel.

The smallest, simplest driver for a typical synchronous link to an X.25 network would provide a way for a process to transmit one byte of data and receive one byte of data. Such a driver would be called many times to send or receive the sequences of bytes that make up X.25 link-level frames. However, there are several aspects of serial, synchronous communication that conspire against such a simple implementation. Firstly, unlike asynchronous communication, it is necessary that the bytes be produced and consumed at a fixed rate, at least within a frame. If the

transmission rate is high enough and the process-switching time is long enough, processes will be unable to keep up. A full-duplex, 9600 bits per second interface can generate an interrupt approximately every 400 microseconds — time for only 100 to 200 instructions on a typical minicomputer. The speed problem can perhaps be alleviated by buffering in the kernel, but that greatly increases the complexity of the driver. Secondly, the driver must be aware of frame boundaries to properly switch the interface between transparent data mode and inter-frame idle mode and to make use of cyclic redundancy check (CRC) hardware, if available. This requires additional process-to-kernel interaction beyond simple reading and writing of bytes. Thirdly, for some machines there are direct memory access (DMA) interfaces which read and write whole frames between memory and the communication link. This kind of hardware is essential for very high-speed links (some public X.25 networks provide subscriber access at 56 or 64 kilobits per second). It would be wasteful and, in some cases, too slow to transfer frames between the communication link and kernel space and then pass the frames a byte at a time to and from processes, rather than exchanging complete frames directly with the processes' memory. For these reasons, a better design has the driver deal with complete frames instead of single bytes, even though this may take more code in the kernel. Processes request the driver to read and write a frame as a unit, leaving the driver responsible for frame delimiting, transparency of data within the frame, and CRC generation/validation.

Adopting such a frame-at-a-time driver interface solves the above problems and provides some additional benefits. If DMA hardware becomes available to replace byte-interrupt hardware on a particular machine, only the driver need be changed: the processes and the process-to-kernel interface remain the same. The same is true if alternative framing, transparency, or checksumming mechanisms are employed — even an asynchronous link could be used to carry the frames. The X.25 processes are insulated from device- and processor-specific idiosyncracies; they manipulate only the virtual, "ideal" device provided by the kernel and can therefore be transported

unchanged from one computer to another.

The hardware used for the first implementation of this driver is far from ideal: the only synchronous communication interface available for the TI 990/10 has neither DMA nor CRC capabilities and, much worse, does not support the "bit-stuffing" required for X.25 framing and transparency. Fortunately, the Datapac network, in addition to bit-oriented X.25 framing, supports a byte-oriented framing structure which is compatible with our hardware [8]. Our access to this service is via a 1200 bits per second leased line. CRC calculations are done in software, using a fast table-lookup algorithm [29]. In the absence of DMA support, an effort is made to reduce processor overhead: only one pass is made over each frame on its way between process memory and I/O device — CRC updating and transparency are handled on the fly with no intermediate buffering in kernel address space. This "pseudo-DMA" strategy shares a drawback with real DMA: the processes providing or consuming the frames must be locked into memory for the duration of the transfer. Since frames can arrive, unsolicited, at any time, the receiving process must be permanently memory-resident.

There are real-time constraints both within frames and between frames. Within a frame, interrupts must be serviced at the fixed rate of the link to avoid character loss on input or garbage fill on output. Adequate response is guaranteed by placing the synchronous I/O device at a high interrupt priority. The Verex kernel, including its device drivers, is designed to disable interrupts only for very short, bounded periods of time, thus ensuring that interrupts from high-priority devices can always be serviced within a small, fixed time of their occurrence.

The constraint on response between frames applies to reception only — the transmitter can idle between frames. Since receive buffers are provided by a process, that process must be ready with a new buffer very soon after reception of a frame. This requirement is met by assigning a high scheduling priority to the

receiving process, so that it executes immediately after being unblocked by the driver. This is sufficient for the current implementation where the byte-oriented framing sequence guarantees 4 or 5 byte times between incoming frames; it may not be adequate for handling a high-speed link with normal X.25 framing where frames can arrive separated by only a single flag character. In that case, it would be necessary to either provide buffering in the kernel, with consequent extra overhead, or allow the receiving process to supply more than one buffer at a time for sequences of frames, at the cost of increased complexity at the process-to-kernel interface.

Although the driver is part of the kernel, to the X.25 processes it looks and acts exactly like another process. Requests for reading and writing of frames take the form of messages to a special process identifier which is recognized by the kernel. Disguising the driver as a process has significant advantages for development, debugging, and performance monitoring: it is easy to substitute a real process for the driver or insert a process transparently between the driver and its clients. The format and semantics of the messages to the driver conform to the Verex file access interface [12], making the driver look like a standard file server. Scotton in [27] points out the benefits of a uniform interprocess interface for all I/O-style data transfer: I/O applications are portable over different devices, processes can share a common library of I/O routines, and when used by components of layered communication protocols it allows simple stacking of layers and interchanging of components. The X.25 frame layer driver is thus designed to fit in as the bottom-most component in a hierarchy of protocol servers.

4.2 The Link Layer

The link layer comprises four processes on the X.25 team, as illustrated in Figure 3. The **Link Server** process maintains the state of the link according to the X.25 Link Access Procedure (LAP). It alters the state in response to request messages from the packet layer and notification messages from three "worker" processes, the **Frame Reader, Frame Writer,** and **Timer.** The Frame Reader requests incoming frames from the frame layer and notifies the Link Server whenever one arrives. The Frame Writer requests the frame layer to transmit frames and notifies the Link Server whenever one has been sent. The Timer delays for fixed intervals by sending requests to the system's Time Server (on another team) and notifies the Link Server whenever the time interval elapses.

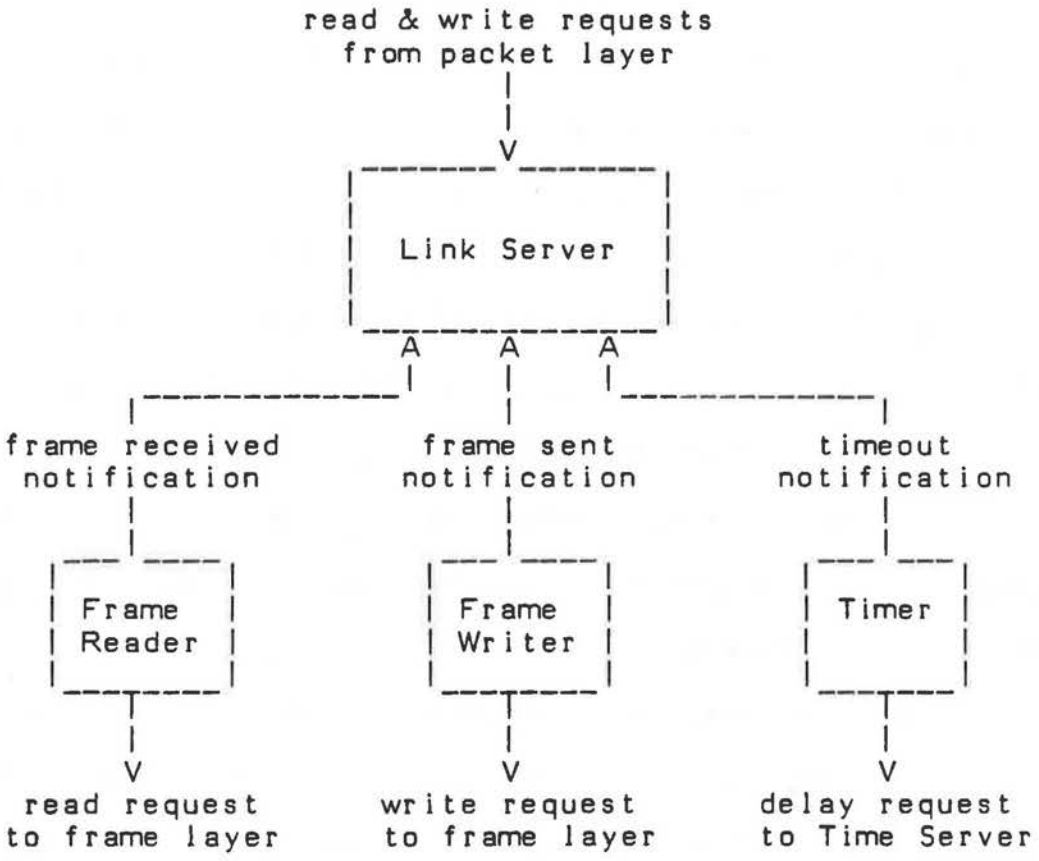

Figure 3. Process Structure of the Link Layer

This process structure is typical of Verex server teams; it arises from application of the guidelines for multi-process structuring. The asynchronous, externally-generated events of frame arrival, frame departure, and timer expiration are each assigned a process which waits for the event's occurrence. The variables which represent the link state are affected by each of the events, and therefore are placed under the control of a single process, the Link Server, which updates them on behalf of the other processes. The Link Server itself never waits for specific events and is therefore always ready to receive requests from other processes — if the handling of a request must wait for another event to occur, the server internally queues the request and immediately becomes ready to accept the next request. A high frame arrival rate is handled by assigning the highest scheduling priority to the Frame Reader and guaranteeing quick service from the Link Server.

It is natural and useful to view the Link Server as a finite-state machine (FSM), performing state transitions in response to events in the form of received messages, and causing external effects by replying to messages. State transition diagrams have become a standard tool for describing communication protocols. A correct implementation of a protocol can be produced quickly and easily by casting its state diagram in the form of an executing FSM. Unfortunately, the state diagrams specified in the X.25 definition [10] are not suitable for direct implementation. Firstly, they are provided for the packet-level only — the link-level is defined only in English prose which is open to ambiguous interpretation. In fact, the implementors of Datapac found it necessary to produce an additional document [26] (also in English prose) to correct many ambiguities and omissions in the X.25 definition. Secondly, the packet-level diagrams refer to the state of a virtual circuit between two machines, whereas the implementor needs a state diagram for the entity at one end of the circuit only. This distinction is discussed by Vuong and Cowen [28], who show how to derive separate state diagrams for the two ends, given a combined state diagram. Thirdly, the X.25 state diagrams do not include such state components as

sequence counters and packet queues, which are of great importance to the implementor. Therefore, it was necessary to derive the FSM structure of the Link Server from the prose description of X.25 LAP.

The LAP definition clearly distinguishes transmitter (primary) functions from receiver (secondary) functions. Once the link is established, these two sets of functions proceed relatively independently and therefore result in simpler state transition graphs if implemented as two separate FSMs. Since LAP defines a full-duplex link, logically the two FSMs execute concurrently. It is tempting to use two processes for the the two state machines — a **Primary Server** and a **Secondary Server** — but in this case there is little benefit in doing so. No real concurrency would be obtained by separating the two, since both would execute on a single processor and neither would ever block in the middle of a state transition. (Overlap of actual transmission and reception is provided by the separate Frame Writer and Frame Reader processes.) Moreover, considerable message traffic would be required between the two servers to support the "piggybacking" of secondary responses on primary frames, to coordinate shared use of the single I/O device, and to synchronize in the case of link disconnection. Therefore, to avoid the extra message and process-switch overhead of two servers, both primary and secondary functions are implemented within the single Link Server. However, the distinction between the two FSMs is retained in the control structure of the single process to take advantage of their simpler state transition graphs.

Figure 4 shows the top-level control structure of the Link Server. It follows a standard pattern for Verex servers. After allocating an initialized state vector, **state**, the procedure executes an endless loop. At the top of the loop, it waits to receive a message. The message format is defined by the template **REQUEST** to contain an identifying request code plus request-specific fields. (All requests except **TIMEOUT** have a single extra field containing a pointer to a frame (packet) buffer;

TIMEOUT requests have no extra fields.) Based on the request code, the Link Server invokes a corresponding event handling routine which updates the state as required and returns either a reply code to be placed in the reply message or an indication that no reply is to be sent. This loop is the only place where the Link Server blocks doing a **Receive**; the nonblocking **Reply** is called within some of the subroutines to respond to those requesting processes which do not get an immediate reply.

```
Link_server( device )
  {
     template REQUEST, REPLY;
     unsigned id, reply;
     word  state[], msg[.MSG_SIZE];

     state = Initialize( device );

     repeat
       {
          id = Receive( msg );

          select( REQUEST_CODE[msg] )
            {
               case READ_BUF:    reply = Read_buf( state, msg, id );
               case WRITE_BUF:   reply = Write_buf( state, msg, id );
               case FRAME_IN:    reply = Frame_in( state, msg, id );
               case FRAME_OUT:   reply = Frame_out( state, msg, id );
               case TIMEOUT:     reply = Timeout( state, msg, id );
               default:          reply = ILLEGAL_REQUEST;
            }

          if( reply == NO_REPLY ) next;

          REPLY_CODE[msg] = reply;
          Reply( msg, id );
       }
  }
```

Figure 4. Main Procedure of the Link Server

Write_buf and **Timeout** perform LAP primary functions, concerned with data transmission. **Read_buf** performs LAP secondary functions, concerned with data reception. **Frame_in** and **Frame_out** are demultiplexing routines that invoke either primary or secondary routines, depending on the frame type. In some cases, primary routines invoke secondary routines, and vice versa, to handle such matters as

piggybacked responses in data frames and resetting of all state information on link disconnection. Therefore, all the routines receive as an argument the same state vector which contains both primary and secondary state variables as well as such shared state as the transmit queue. This state machine structure sacrifices some readability — most of the control flow is implicit in the sequence of values taken on by the state variables rather than explicit in the sequence of statements — but avoids the use of exception trapping mechanisms, "long jumps", or convoluted code to handle exceptions to the "normal" flow of control. In a full-duplex protocol for an error-prone circuit, it is difficult and perhaps meaningless to identify the normal flow of control: exceptions are the rule.

The three worker processes are much simpler than the Link Server: each executes a single loop in which it waits for its particular event and then notifies the Link Server. The Link Server controls their execution by choosing when to reply to the notification messages. In the case of the Frame Reader and Frame Writer, the reply from the Link Server contains a pointer to a new buffer to be filled or emptied.

The Link Server implementation violates one of the main principles of protocol layering: it uses knowledge of higher layer behaviour. Packet layer behaviour influenced the following design decisions:

— The Link Server **never** transmits a Receive Ready (RR) frame in response to an incoming Information (I) frame unless the I frame has the poll bit on. Instead, the Link Server waits to piggyback the acknowledgement on the next outgoing I frame. The success of this strategy depends on the fact that at the packet layer most incoming packets generate outgoing packets in response, and therefore there is usually an I frame available for the piggyback. When there is not, the remote end eventually times out and retransmits its I frame with the poll bit on, prompting an RR frame from the Link Server. As a

result, the number of transmitted frames is cut almost in half, reducing processor overhead and increasing potential throughput. Retransmissions on timeout occur only when the link is otherwise idle; they do not represent delayed reception since the original frame is usually received successfully and the repeated frame discarded. This same technique could be applied to higher levels as well: if the packet layer was always used beneath a higher-level acknowledging protocol, packet layer RRs could be suppressed, and so on up the layers.

— The link layer processes share a common buffer pool with the packet layer processes on the same team. The process holding a pointer to a buffer assumes full rights to modify or discard the buffer. A more complex buffer sharing arrangement would have to be imposed, or sharing eliminated altogether, if the link layer could not make assumptions about the buffer handling behaviour of the packet layer. For example, if the packet layer required a retransmission queue as does the link layer, a buffer might need to reside on both queues at once and could not be unilaterally discarded by one process.

— The link layer does not impose any flow control on incoming frames, i.e., it never sends Receive Not Ready (RNR) frames and it always rotates its receive window when an I frame is successfully received. Instead, it depends on the flow control imposed by the packet level windowing mechanism to prevent overflow. This simplifies the Link Server by reducing the number of states (no need for a "not ready" state) and eliminating all the tests required to determine when buffers are exhausted and when they become available.

The gains in simplicity and performance resulting from these dependencies outweigh the disadvantages of tight binding between the two layers: we do not anticipate using any protocol other than the X.25 packet layer on top of this link layer.

The link layer implements X.25 LAP as defined for Datapac in [8]. It deviates from that definition in some aspects. As mentioned above, RR frames are only generated in response to polls, and RNR frames are never sent. Furthermore, received RNRs are treated as if they were RRs. This simplification could result in a greater number of discarded frames, and subsequent retransmissions, than if RNRs were honoured. However, we have never actually received an RNR over our link to Datapac, and if the other end was another instance of our link layer, we would be guaranteed never to see an RNR!

Another liberty was taken with the LAP definition to permit two instances of the link layer to communicate without an intervening network. X.25 is defined as an asymmetric interface for connecting a host (DTE) to a network node (DCE). This asymmetry is present in LAP only in the different address values used by the DTE and DCE in frame headers; it guards against accidental loopback of the communication link. However, a loopback capability is valuable for development and debugging. A symmetric interface allows transparent loopback as well as back-to-back connection of two DTEs. To obtain these benefits, a technique described by Bradbury [5] was adopted, whereby the link layer dynamically adapts to DTE or DCE mode, depending on the actions of the communicating partner. Basically, the method takes advantage of the fact that LAP commands and responses are encoded with different values, making the address fields redundant. The redundant information is enough to identify the role (DTE or DCE) that the remote link layer is playing.

It would be easy and sensible to modify the Link Server to conform to the Balanced Link Access Procedure (LAPB) of the more recent, revised X.25 recommendation [11] as it becomes available on Datapac and other networks. Unfortunately, LAPB has more asymmetries than LAP (in spite of its name) and, as Bradbury points out, the heuristics for dynamic adaptation to DTE or DCE mode are

considerably more complex.

4.3 The Packet Layer

The packet layer of the X.25 team is made up of one **Channel Server** process for each logical channel and one shared worker process, the **Packet Distributor.** Figure 5 illustrates the relationship between these processes and the layers above and below for a three channel X.25 service. Each Channel Server maintains the state of its own logical channel, responding to packet arrivals from the Packet Distributor and I/O requests from higher-level clients. To transmit packets, the Channel Servers send write request messages to the Link Server in the link layer. The Packet Distributor spends most of its time awaiting a reply to a read request to the Link Server. When it receives a packet, it determines which Channel Server to notify based on the logical channel identifier field in the packet header. Packets with an invalid or zero channel number are handled specially by the Packet Distributor. For example, the Packet Distributor responds to a restart indication packet on channel zero by sending a restart confirmation packet back to the Link Server and notifying the highest-numbered Channel Server. The notification is passed down the line of Channel Servers using the **Forward** message primitive. The same forwarding technique is used to allocate logical channels: the highest-numbered Channel Server receives all client requests to establish a new virtual circuit and forwards them along to the first free Channel Server (much like a telephone "hunt group").

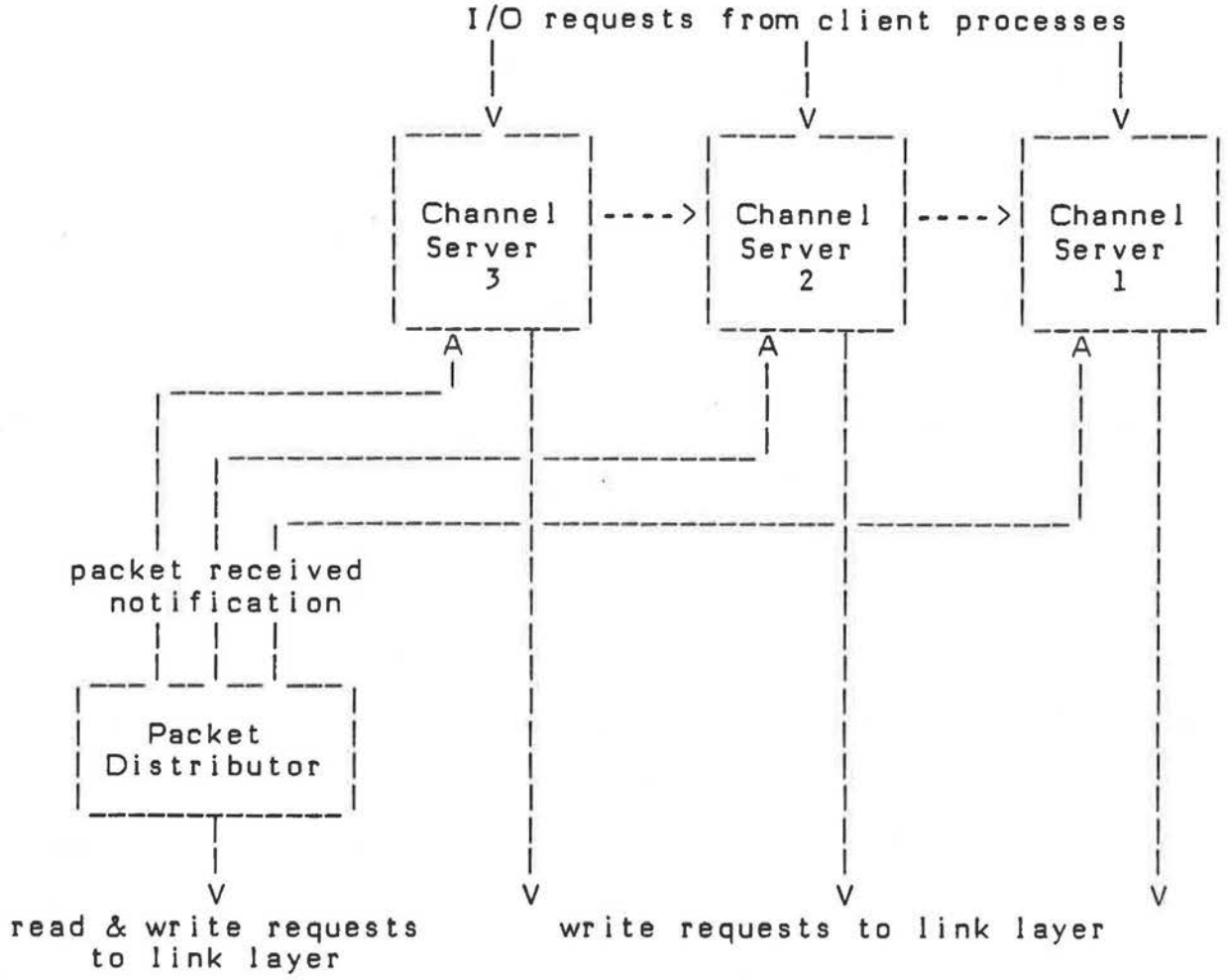

Figure 5. Process Structure of the Packet Layer

The packet layer functions could have been incorporated into the lower-level Link Server process; separate processes were chosen in order to make very clear the distinction between the two protocol layers. It would be hard to keep this distinction visible within the state machine structure of the Link Server. The clear separation eases the task of programming from the protocol descriptions, greatly enhances understanding by others, and facilitates the replacement of a layer (for example, the replacement of the link layer by a hardware implementation [30]). The cost of this separation is the overhead of message-passing and process-switching between the layers, a relatively small cost in the Verex environment.

Internally, the Channel Servers have the same finite state machine structure as the Link Server. Identifying the states and valid transitions was made easier by the state transition diagrams provided for the packet layer in the X.25 specification.

The FSM for each logical channel is embodied in a separate process in order to exploit possible concurrency. Because of swapping, the exchange of data packets with clients in other address spaces using the **Transfer-to** and **Transfer-from** primitives can sometimes involve disk operations. Providing separate Channel Servers allows packet traffic on one logical channel to overlap disk traffic for another.

The Channel Servers are all created at system initialization time, rather than dynamically in response to virtual circuit requests. Dynamic creation and destruction of Channel Servers would complicate the structure of the layer: a **Channel Server Manager** would be required to handle the coming and going of Channel Servers and the communication between Channel Servers would have to take process creation and destruction into account. Since the maximum number of logical channels is fixed at network subscription time, and since the X.25 service must have access to enough resources to support all of the channels in use at the same time, it is reasonable and convenient to statically allocate the maximum number of Channel Servers. The resources consumed by each additional Channel Server are stack space on the X.25 team (300 bytes in the current implementation), six full-size packet buffers in the team's buffer pool (267 bytes each for Datapac), and a process descriptor block (currently 56 bytes) in the kernel. After taking into account the other demands on the team's address space, these requirements limit the number of logical channels that can be supported in the current implementation to about 24, more than enough for such a small machine.

The use of multiple Channel Servers necessitates the demultiplexing function of the Packet Distributor. This function could have been performed within the Link Server, but since it is clearly a packet-layer function, the above argument for layer

separation applies. Moreover, the Packet Distributor process plays an important role as a relayer of messages from the Link Server to the Channel Servers. Without an intermediate process, message-passing between the two layers of servers could take one of two forms:

— The Link Server could send packet arrival messages directly to the Channel Servers. Since write request messages are sent in the other direction, this would create great potential for deadlock.

— The Channel Servers could send read request messages to the Link Server. Since packets can take indefinitely long to arrive, this would cause the Channel Servers to block indefinitely waiting for replies, rendering them deaf to other requests from clients. In particular, a Channel Server could not service a write request for a client while awaiting an incoming packet.

Therefore, the Packet Distributor allows full-duplex communication between the two layers of servers, while providing deadlock immunity.

The need for another worker, complementary to the Packet Distributor, to wait on write requests to the Link Server is obviated by having the Link Server reply immediately to write requests, queueing the outbound packets for eventual transmission. From the point of view of the Channel Servers, transmission is instantaneous. Unfortunately, this reduces the opportunities for piggybacking of acknowledgements: when a Channel Server has an acknowledgement to send, there is never a data packet waiting to go out; instead, they queue up, out of reach, in the link layer. Also, unlike the link layer, the packet layer cannot simply withhold all acknowledgement packets and depend on retransmissions and higher-level traffic, because the packet layer does not retransmit and there are many possible higher levels. However, one simple scheme is employed to provide occasional piggybacking: an acknowledgement for an incoming data packet is withheld until the next I/O

request is received from the client; if the request is a write, the acknowledgement is piggybacked on the outbound packet; if it is a read, the acknowledgement is sent by itself. This trick works well if the client's application is half-duplex in nature, such as a command/response terminal session or remote procedure invocation. It does not help at all for applications that receive many packets before generating any response, such as bulk file transfers.

There are significant advantages to placing the packet layer processes on the same team as the link layer. Most important is the reduction of expensive data copying between address spaces. As indicated in the preceding discussion, the two layers share a common pool of packet buffers and simply exchange pointers to buffers; data is only copied once to or from the clients' address spaces and once in or out by the frame level driver. The two layers also benefit from shared code: processes in both layers use a common set of procedures for language support, buffer pool management, queue manipulation, and modulo arithmetic. This sharing can be extended to multiple X.25 links by replicating both layers within the single team, subject to address space limitations.

The implementation of the packet layer conforms to the 1976 version of X.25 as specified in [8]. There is no support for permanent virtual circuits nor for the more recently defined datagram or "fast select" facilities [11], although any of these could easily be added if desired. Like the link layer, the packet layer processes never transmit RNR packets and incoming RNRs are treated like RRs. Reject (REJ) packets are neither generated nor accepted, as specified for Datapac.

Consideration of Bradbury's technique for DTE/DCE adaptation at the packet layer [5] reveals a problem with the multiple Channel Server structure: the method requires knowledge of the current state and history of each logical channel, knowledge that is distributed among the separate Channel Servers. This distribution of state would also be an obstacle to the implementation of a datagram-based

internetwork protocol, such as IP [23] or Pup [3], on top of X.25 virtual circuits: the heuristics for deciding when to establish and when to tear down virtual circuits depend on the number of circuits in use, their destinations, and demand by competing clients. It would require much message traffic to collect that information from the distributed Channel Servers, and it would be impossible to obtain a consistent "snapshot" of the total state, due to their independent execution.

Recall that the motivation for separate Channel Servers was to exploit concurrency between packet traffic and disk transfers with swapped clients. Unfortunately, the behaviour of the Packet Distributor can compromise this concurrency: if it sends an incoming packet notification to a Channel Server which is blocked awaiting a disk transfer, the Packet Distributor must also wait, effectively cutting off incoming traffic for other channels. Furthermore, if the X.25 communication link is slow compared to the disk transfer time — the usual case — or if the probability of a client being swapped out is low, any performance improvement due to concurrent execution of Channel Servers would be negligible.

In retrospect, it appears preferable to combine the Channel Servers into a single process. This would enable the use of algorithms that depend on global packet layer state. As a minor benefit, it would also raise the ceiling on number of logical channels, since a separate execution stack would not be required for each channel. If the X.25 team were to be used in a heavy swapping environment, concurrency could still be exploited by employing a small number of worker processes to perform the actual transfers across address spaces.

4.4 The Transport Layer

X.25, and our implementation of X.25, fulfils the requirements of the bottom three layers of the seven layer ISO Reference Model for Open Systems Interconnection. Within that model, the X.25 service would be expected to support layer four, the transport layer. Many of the functions required of the transport layer are similar in nature to those of the packet and link layers, and therefore the same multi-process structuring methodology can successfully be applied to the design of transport layer software.

Since the packet and link layer software share the same address space to avoid extra data copying, it is tempting to incorporate the transport layer into the X.25 team for the same reason. However, there are several arguments against doing so:

— There is presently no widely accepted transport layer protocol. The carriers that provide X.25 networks are not currently compelled to adopt a standard transport protocol, since transport is an end-to-end protocol of concern only to software in the customers' computers. This may change as carriers start providing higher-level services such as inter-machine electronic mail.

— For some applications, a transport layer is unnecessary or unwanted. For example, the X.29 remote terminal protocol [9] is defined for use directly on top of X.25. Since X.25 is a useful stand-alone service it is best packaged independently of any higher layer protocols, able to support a number of applications and communication architectures.

— One of the major purposes of the transport layer is to provide application processes with a network-independent interface to multiple networks with, perhaps, different low-level protocols. A transport layer team that is separate and independent of any lower-level protocol software would best accommodate a variety of networks.

— The buffer handling requirements of a transport service are unlikely to be compatible with those of the X.25 team. For example, the transport protocol might require its own retransmission queue to obtain reliable end-to-end data transfer, necessitating a more complex buffer sharing regime. The buffer size might have to be increased if the transport service provides fragmentation/reassembly of large data blocks. The complications introduced by these incompatibilities are easily avoided by copying data between transport layer buffers and X.25 buffers. The cost of copying across address spaces is, one hopes, not much greater than copying within an address space.

In general, protocol layers should be separated into different teams or spaces unless they are components of a coherent protocol "family", unlikely to be used individually. This is the case with the packet and link layers of X.25. The X.25 family does not (yet) include a transport layer protocol.

4.5 Buffer Management

To minimize data copying, all the processes on the X.25 team share a common pool of packet buffers. Any block of data is copied only once into the team's space and once out; the X.25 processes simply exchange pointers to buffers as the data makes its way through the layers.

The buffer pool is implemented as a linked list of free buffers, pointed to by a global variable. The processes obtain and discard buffers as needed by directly manipulating the free buffer list. Concurrent access by multiple processes raises the possibility that the list may become corrupted by unsynchronized modifications. This is avoided by taking advantage of a property of Verex process scheduling called **relative indivisibility:** a process executes indivisibly with respect to other processes of the same or lower priority on the same team until it blocks. In other words, within a team, a process can be preempted only by higher priority processes.

On the X.25 team, only the Frame Reader and Frame Writer have higher priority than other processes, in order to service the link device promptly. By arranging that those two processes never touch the free buffer list, all the others are free to manipulate it without danger of mutual interference.

Relative indivisibility provides a second mechanism for mutual exclusion, in addition to the use of message-passing. However, its benefits are minor — it avoids the overhead of message-passing for synchronization — and its use can always be replaced by a suitable process and message structure. For example, the X.25 free buffer list could be placed under the control of a single process which alone manipulates the list in response to request messages from the other processes. The position of the Link Server in the process structure of the team makes it a suitable candidate to provide this extra service.

All the buffers are the same size: large enough to handle a maximum-size frame. Unfortunately, much of the traffic consists of tiny control messages, such as link layer and packet layer acknowledgements. More efficient use of buffer memory could be made by supporting two sizes of buffers — large and small — or using buffer fragments that are chained together to hold large frames. As usual, such increased space efficiency would be bought with increased execution time: more processing would have to be done to select buffer sizes, follow chain pointers, etc.

Enough buffers are created at compile time to handle the observed worst-case demand. If ever the buffer pool is exhausted, the X.25 team halts execution, destroying all connections. It must then be recompiled with more buffers. Obviously, it would be preferable to use fewer buffers and to survive the occasional shortage during periods of unusual demand. However, when a process finds that the buffer pool is empty it is not sufficient for it to just wait until another process frees a buffer — the other processes may require traffic (such as acknowledgement packets) from the blocked process before they will free any buffers. To avoid such deadlock,

it would be necessary to free up some inessential buffers. Since buffers are distributed among separate processes, considerable synchronization and communication would be required to identify the disposable buffers and to transfer them to the needy process(es). This problem has not been successfully addressed in the current design. The only virtue of the straightforward scheme adopted is its simplicity as reflected in the ease of programming, speed of execution, and small size of the Link and Channel Servers — there is no code to handle buffer shortages.

5. Evaluation

The X.25 software as described was implemented by the author part-time over a period of four months and represents approximately two months of effort. Much of the development time was spent on the Frame Layer driver because of difficulties with the available communication hardware. The packet layer was programmed after the link layer and took about half as long, with the benefit of experience. The author applied the same structuring methodology to the somewhat simpler X.29 protocol to produce a virtual terminal program in little more than a weekend. (It was needed to test the X.25 software!)

The X.25 packet layer comprises 1268 lines of Zed language source, the link layer is 1005 lines, and procedures shared between the two layers contribute another 370 lines. The frame layer driver in the kernel includes 190 lines of Zed and 555 lines of Assembly Language. The large amount of Assembly Language reflects the shortcomings of the hardware, and includes software CRC calculations. The use of a high-level, portable programming language and concern with portability during programming have made all but the frame layer software easily transportable to other machines.

On the TI 990/10, the software translates into 17,066 bytes of program code (1206 in the kernel), 1792 bytes of stack and data space per link (72 in the kernel), and 2248 bytes of stack and data space per logical channel (including buffers). Thus, a single-link, three-channel X.25 service requires 24,324 bytes on the X.25 team and 1278 bytes in the kernel.

The software has been in use for over a year, supporting incoming and outgoing terminal (X.29) connections over Datapac. It has proven to be extremely reliable: it has failed only once by running out of buffers. (The network has failed much more often.)

Unfortunately, little performance data have been collected on the software, due to lack of a suitable testing configuration. Our slow (1200 bits per second) network connection prevents significant loading of the software: even when handling heavy traffic on multiple channels, no degradation of overall system performance can be perceived. A one-way file transfer to a faster receiver attains an effective data rate of 1117 bits per second — 93% of the link capacity. This reflects the efficiency of the protocol more than the software.

The X.25 software design has been used by others as a model for different communication protocols. In addition to the virtual terminal program mentioned above, a host-side X.29 service has been implemented [27] and a Byte Stream Protocol server for the Cambridge Ring local-area network [20] was constructed in a month by two undergraduate students with no previous experience in implementing communication software. This is a measure of the understandability and maintainability of the software and the generality of the structuring methods.

Very little has been published of structuring methods for X.25 or other protocol software. In one of the few relevant papers, Bochmann and Joachim [2] describe an implementation derived from a formal specification of X.25 and based on the multi-process facilities of Concurrent Pascal. They decompose the protocol into a logical structure of modules communicating via messages. Their module structure is very similar to our collection of processes — some modules are finite state machines and some are workers performing I/O or multiplexing functions. However, they then go on to describe a technique for translating the logical structure into a physical realization using Concurrent Pascal processes and monitors. The resulting

implementation is considerably more complex and opaque, if the difference in their diagrams can be taken as evidence. The authors say they would have been happier with language facilities (or, presumably, operating system facilities) that more closely matched their logical structure.

Cohen [16] describes the implementation and performance of X.25 and X.29 software using the RSX-11M operating system on a PDP/11 computer. RSX-11M is a typical real-time system with a large collection of facilities for multi-process, real-time programming. Cohen used these facilities to build a configuration of separate processes supporting the various layers of protocol and communicating via messages. His only comment on those facilities was that early versions of his software tended to use too many of them or use them in an inefficient or incorrect manner.

The facilities for multi-process structuring provided by Verex have proven to be adequate and desirable for communication software. The main drawback of our methodology is that the Verex environment is not universally available. However, there are other message-based systems which differ in various degrees from Verex, which could support software like that described. Lack of shared address spaces, non-blocking message-passing, different scheduling policies, etc., would effect some aspects of the design, but not all.

The cost of interprocess communication on some systems might require combining the Link Server and Channel Server functions into a single process in order to obtain reasonable performance. As pointed out in Chapter 4, the motivation for separating the layers across processes was modularity rather than concurrency. It may be possible to maintain the visible separation of the layers within a single process by use of language facilities for module definition or other coding conventions. The tighter binding of the layers would present other opportunities for improved performance: access to the link layer queues by the packet layer would

allow more packet-level piggybacking of acknowledgements and facilitate high-priority handling of interrupt and reset packets. Recovery from buffer shortages would be considerably simpler with only one process to deal with.

The mechanisms described in Chapter 3 for handling incoming calls and for signalling interrupts to clients seem awkward and heavy-handed, relative to the rest of the design. Both of these situations require communication initiated by a server (the X.25 team), directed to a client, unlike the normal client-initiated interactions which are served so well by the remote procedure call style of message-passing. It is not clear whether this indicates an inadequacy of the interprocess communication primitives or of the design.

As demands on the X.25 software increase, changes will be needed. In particular, more sophisticated buffer management algorithms will be required to make better use of memory as the number of logical channels increases. The newer features of X.25, such as LAPB and datagram support, should be incorporated into the software as the need arises. As higher speed links are adopted, performance measurements will be needed to focus optimization efforts.

It should **not** be necessary to change the software significantly to take advantage of multiprocessor or distributed computer architectures. The use of processes for the protocol layers and messages for communication between the layers should allow easy distribution of all or part of the X.25 software to separate machines. An existing example is a multiprocessor version of Verex [4] which gives the X.25 team its own dedicated processor with no changes to any software outside of the kernel.

In conclusion, it has been shown that software to support X.25 can be implemented quickly, cleanly, and correctly. Multi-process structuring methods applied to layered specifications of finite-state machines can satisfy the

functionality and performance demands of not only X.25 but many modern communication protocols.

Bibliography

1. Bochmann, G.V., "Finite state description of communication protocols", Computer Networks 2, 4/5 (October 1978), 361-372.

2. Bochmann, G.V. and Joachim, T., "Development and structure of an X.25 implementation", IEEE Transactions on Software Engineering 5, 5 (September 1979), 429-439.

3. Boggs, D.R., et al, "Pup: an internetwork architecture", IEEE Transactions on Communications 28, 4 (April 1980), 612-623.

4. Boyle, P.D., "The design of a distributed kernel for a multiprocessor system", University of British Columbia, Department of Computer Science, M.Sc. Thesis, June 1982.

5. Bradbury, C., "X25 asymmetries and how to avoid them", ACM Computer Communication Review 8, 3 (July 1978), 25-34.

6. Brinch Hansen, P., RC4000 Software Multiprogramming System, A/S Regnecentralen, Copenhagen, 1969.

7. Bunch, S.R. and Day, J.D., "Control structure overhead in TCP", Proc. IEEE Trends and Applications: 1980, Computer Network Protocols, Gaithersburg, Maryland, May 1980, 121-127.

8. Computer Communications Group, "Datapac Standard Network Access Protocol Specification", Trans Canada Telephone System, Ottawa, Ontario, March 1976.

9. Computer Communications Group, "Datapac Interactive Terminal Interface (ITI) Specification and User's Manual", Trans Canada Telephone System, Ottawa, Ontario, October 1978.

10. CCITT Recommendation X.25, "Interface between DTE and DCE for terminals operating in the packet mode on public data networks", March 1976.

11. CCITT Draft Revised Recommendation X.25, "Interface between DTE and DCE for terminals operating in the packet mode on public data networks", February, 1980.

12. Cheriton, D.R., "Distributed I/O using an object-based protocol", UBC Computer Science Technical Report 81-1, University of British Columbia, January 1981.

13. Cheriton, D.R., The Thoth System: Multi-Process Structuring and Portability, Elsevier North-Holland, New York, 1982.

14. Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R., "Thoth, a portable real-time operating system", Communications of the ACM 22, 2 (February 1979), 105-115.

15. Cheriton, D.R. and Steeves, P.J., "Zed language reference manual", UBC Computer Science Technical Report 79-2, University of British Columbia, September 1979.

16. Cohen, N.B., "Implementation and performance of an X.25 packet network interface machine", Technical Report T-78, Computer Communications Networks Group, University of Waterloo, Waterloo, Ontario, September 1978.

17. Haverty, J.F. and Rettberg, R.D., "Interprocess communications for a server in UNIX", Proc. IEEE Computer Society International Conf. on Computer Communications Networks, September 1978, 312-315.

18. Hoare, C.A.R., "Communicating sequential processes", Communications of the ACM 21, 8 (August 1978), 666-677.

19. ISO/TC97/SC16, "Data Processing - Open Systems Interconnection - Basic Reference Model", Document N537 Revised, December 1980.

20. Johnson, M.A., "Ring byte stream protocol specification", Computer Laboratory, Cambridge, April 1980.

21. Kernighan, B.W. and Ritchie, D.M., The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

22. Lockhart, T.W., "The design of a verifiable operating system kernel", UBC Computer Science Technical Report 79-15, November 1979.

23. Postel, J., Editor, "DoD Standard Internet Protocol", ACM Computer Communication Review 10, 4 (October 1980), 12-51.

24. Rashid, R.F., "An interprocess communication facility for UNIX", Technical Report CMU-CS-80-124, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, revised June 1980.

25. Ritchie, D.M. and Thompson, K., "The UNIX timesharing system", Communications of the ACM 17, 7 (July 1974), 365-375.

26. Rybczynski, A., Weir, D., and Palframan, J., "Questions and answers on X25 and characteristics of intra-Datapac virtual circuits", Computer Communications Planning, Trans Canada Telephone System, Ottawa, Ontario, March 1978.

27. Scotton, G.R., "An experiment in high level protocol design", University of British Columbia, Department of Computer Science, M.Sc. Thesis, December 1981.

28. Vuong, S.T. and Cowen, D.D., "Automated protocol validation via resynthesis: the CCITT X.75 packet level recommendation as an example", Technical Report CS-80-39, Computer Science Department, University of Waterloo, Waterloo, Ontario, revised May 1981.

29. Wecker, S., "A table-lookup algorithm for software computation of cyclic redundancy check (CRC)", Digital Equipment Corp., Maynard, Mass., January 1974.

30. Western Digital Corporation, "LSI packet network interface WD2501/11, short form data sheet", Preliminary Specification, Irvine, California, June 1981.