NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Design and Validation of an XTP simulator

Jason Xiao-Guang Chen

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

ISBN 0-315-51353-5

ABSTRACT

Design and Validation of an XTP Simulator

J.X.G. Chen

This thesis presents a desigtn and validation of a simulator for a transport and network level protocol the Xpress Transfer Protocol (XTP). XTP was designed to provide a highly efficient transport service.

The tool on which the XTP simulation is based is the Local Area Network Simulation Facility (LANSF). The chosen simulation LAN environment was the ETHERNET. LANSF was modified to support virtual circuit simulations.

The conceptual model of XTP was designed, and used to structure the processes within the LANSF simulation

Subsequent analysis of XTP performance shows that it can provide high throughput to the transport users for file transfer operation. Its bandwidth for short transfers is limited by the necessity of carrying the overhead of packet headers, packet trailers and acknowledgement packets. However, it is no worse than most other protocols e.g. TCP and TP4 in this respect.

To Janet

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincerest thanks to my thesis supervisor, Dr. J.W. Atwood, whose advice and knowledge have helped me initiate, pursue, and complete this research. Professor Atwood also read and criticized many versions of this thesis. The final version owes much to him; the mistakes are my own.

To all my friends and colleagues in the Department of Computer Science, thank you for your encouragement, thoughtful discussions, and help. I would like to specially thank Mr. Thomas Wieland who has patiently helped me to proofread my thesis drafts many times.

I am grateful for the financial assistance provided by Dr. Atwood and the Department of Computer Science.

I would also like to thank my parents for their patience and encouragement. Thanks are also due to my relatives who have consistently given me support and encouragement.

Finally, many thanks to my fiancee Janet, whose caring, support, and confidence never wavered.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (cont.)

LIST OF FIGURES (cont.)

LIST OF LISTINGS

CHAPTER 1

INTRODUCTION

"If the presence of electricity can be made visible in any part of a circuit, I see no reason why intelligence may not be transmitted instantaneously by electricity."

Samuel F.B. Morse

Along with the rapid development of distributed systems and high speed communication media, the demand for a more efficient transport layer protocol for high bandwidth media is increasing quickly. Many research activities have been devoted to this area. The Xpress Transfer Protocol (XTP) is one such example.

The following thesis presents a design and validation of an XTP simulator which can be used for performance analysis. The thesis consists of six chapters:

The second chapter is an introduction to communication protocol models, and the advantages and disadvantages of using these models for distributed system implementations. This chapter also compares two groups of protocols: general purpose protocols and special purpose protocols.

The third chapter introduces the background of the XTP protocol, and describes the operations of XTP, which include connection management, data transfer, termination management, multicasting, and datagram service.

The fourth chapter explains the structure of the Local Area Network Simulation Facility (LANSF), which has been

used in this thesis as the performance simulation tool for XTP. This chapter also points out the need to modify LANSF, in order to do the XTP performance simulation.

The fifth chapter presents our design of the XTP simulation program. This chapter also documents our assumptions on which the simulation and the simulation plan were based. Finally this chapter presents our simulation results.

The last chapter summarizes the results which we have obtained from the XTP performance simulation, and proposes future enhancements to our work.

CHAPTER 2

AN INTRODUCTION TO COMMUNICATION PROTOCOLS

"Mr. Watson, come here, I want you."

Alexander Graham Bell

"It's currently a problem of access to gigabits through punybaud."

J.C.R. Licklider

2.1 The Importance of Communication Protocols in Distributed Operating System Design

The computers forming a distributed system normally do not share primary memory, and so communication via shared memory techniques such as semaphores and monitors is not applicable [27]. Instead, message passing in one way or another is used. In order for processes to maintain communication between its components, the system must be able to deliver messages reliably and quickly from one machine to another. Therefore, achieving reliable and efficient message delivery has become an important issue in distributed system design.

There are many factors that affect how reliably and how quickly a system could transmit messages from one point to another. Some examples are:

- the physical characteristics of the communication media;

- the distance between interconnected machines;
- the behavior of electronic switching circuits;
- the data representation formats;
- the structure of the operating system;
- machine compatibility.

All these factors point out the need to set up some sort of conventions for machine-to-machine communications. The set of conventions which defines the rules governing the exchange of data between two machines is referred to as a protocol [26].

2.2 ISO and DoD Standards

Protocols are usually defined by logically separated layer models, because it is too complicated to specify them in a single layer. Figure 1 is a comparison of several popular protocol reference models.

Among them, the Open System Interconnection (OSI) reference model from the International Organization for Standardization (ISO) is the one most widely discussed in the research work referring to message passing. The main purpose of the ISO reference model is to provide a common basis for coordination of standards development in system interconnection. There are seven layers in the ISO OSI model. The following is a summary of the functionality of each layer [26].

Application: provides an accessing environment to the user.

Presentation: converts different data representations, e.g., from EBCDIC to ASCII.

Session: provides for the establishment of a session between two communicating processes.

Transport: provides reliable and transparent data transfer between end points; provides end to end error recovery and flow control.

Network: provides upper layers with independence from data switching technology used to connect systems (routing functionality).

Data link: provides reliable data transfer across the physical link; sends blocks for synchronization; provides error control and flow control for the upper layers.

Physical: provides electrical, mechanical, functional, and procedural standards to access the physical medium, i.e., cable interfaces, signal encoding and decoding.

By using a standardized layer model, it is possible to connect systems with widely different operating systems, character codes, and ways of viewing the world [26].

Unfortunately, there are two major problems that prevent these layered model from being widely adopted for actual distributed system design:

<table>
<tr><th>Layer</th><th>ISO</th><th>DoD</th><th>ARPANET</th><th>SNA</th><th>DECNET</th></tr>
<tr><td>7</td><td>Application</td><td>Application</td><td>User</td><td>End User</td><td rowspan="2">Application</td></tr>
<tr><td>6</td><td>Presentation</td><td rowspan="2">Utility</td><td>Telnet, FTP</td><td>NAU Services</td></tr>
<tr><td>5</td><td>Session</td><td>(None)</td><td>Data flow control</td><td>(None)</td></tr>
<tr><td>4</td><td>Transport</td><td>Transport</td><td>Host-host</td><td rowspan="2">Transmission Control</td><td>Network Services</td></tr>
<tr><td rowspan="2">3</td><td rowspan="2">Network</td><td>Inter-network</td><td>Source to Destination IMP</td><td rowspan="2">Path Control</td><td rowspan="2">Transport</td></tr>
<tr><td>Network</td><td rowspan="2">IMP-IMP</td></tr>
<tr><td>2</td><td>Data Link</td><td>Data Link</td><td>Data Link Control</td><td>Data Link Control</td></tr>
<tr><td>1</td><td>Physical</td><td>Physical</td><td>Physical</td><td>Physical</td><td>Physical</td></tr>
</table>

Figure 1 Approximate Correspondence between the Various Network Hierarchies [28]

(a) All these layered protocol models do not describe multicasting operation, which is a very important feature of distributed systems, especially in the areas of decentralized naming [5], distributed scheduling, parallel computation [6], distributed transaction management [12] and replication [12].

(b) A substantial amount of overhead results if the implementation exactly follows these layered models.

The heavy overheads are mainly caused by:

(1) There are too many interactions between layers [24]. Figure 2 demonstrates how many interactions are needed among the transport, network, and data link layers, in order to send a single data packet. This figure is derived from the description given in [18]. There are a total of 22 interactions in the process of sending one single user packet! Note: This is a worst-case scenario, where a datapath is opened, the data are sent, and the datapath is then closed. If more data are to be sent than can be contained in a single packet (for example, in a file transfer operation), then the open and close overheads can be amortized over the several data packets.

(2) There are too many buffering operations among the layers, due to the clearly defined layer boundaries [24]. The more buffering operations, the heavier the communication process overhead.

(3) There are redundant functionalities among the layers. For example, cyclic redundancy check (CRC) [26] is

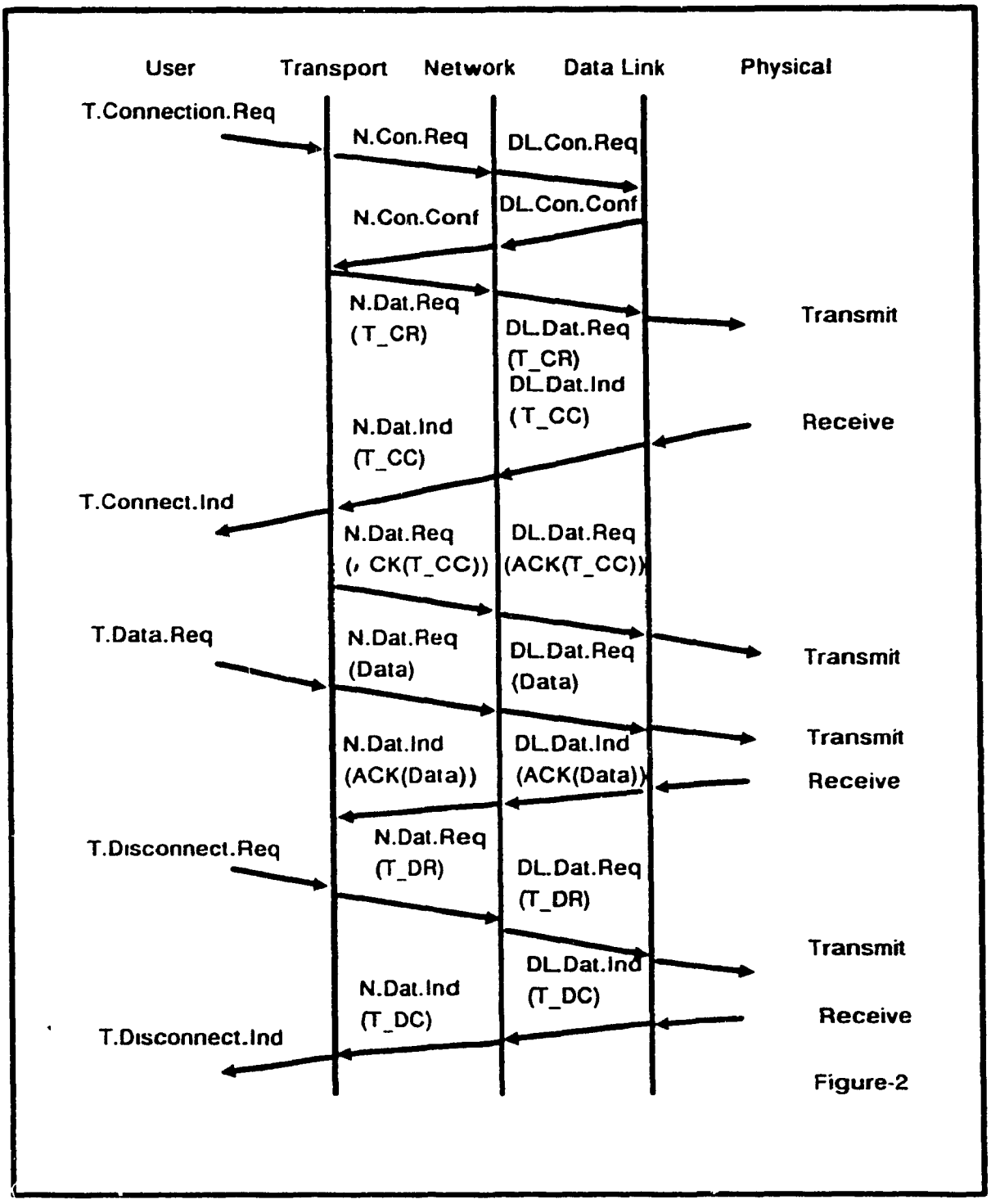


Figure 2 Interactions among transport, network, and data link layers

used by the data link layer, and a checksum scheme is also used by the transport layer, and sometimes, the network layer. Both of the techniques are used for error detection. An extra error detection means an extra parse over a packet.

(4) The complicated design of some layers such as the transport layer, which introduces heavy CPU usage in the implementation [24].

2.3 Transport Level Protocols

2.3.1 Significance of Transport Level Protocols for Distributed System Performance

Distributed systems require their communication protocol to provide reliable and fast end-to-end message delivery service. As described in [28], the transport layer protocols are responsible for end-to-end message delivery. Thus, much distributed system protocol development effort has been put into transport level protocol design.

Figure 3 summarizes the main functions of the transport protocols, and the mechanisms used by most of the designers.

Most of the existing transport protocols can be classified into two groups: general purpose transport protocols and special purpose protocols. The following two sections will compare these two groups in detail.

2.3.2 Traditional Designs of Transport Layer Protocols

General purpose transport layer protocols are mainly developed for inter-system communications [4], which

Function	Mechanism
Connection management	Handshake-based
	Explicit timer-based
	Implicit timer-based
Error recovery	Checksum for error
	Explicit ACK
	Implicit ACK
	Sequence number
Flow control	Explicit sliding-window
	Implicit sliding-window
	Assumption on sender's transmitting rate & discard overflow
	Transmission rate based
Message size	Fragmentation and reassembly

Figure 3 Transport layer functionalities and implementation techniques

concerns activities such as remote terminal access and long file transfer in a wide area network. They are designed based on the assumption that the communication will take place in an environment in which the lower layers have high error rate and the behavior of the environment is unpredictable. An example of such an environment would be using packet radio communication in the hills around Berkeley (the development environment assumed by the TCP design group).

For connection management, general purpose protocols usually use multiple packet exchanges to set up a virtual circuit connection, and to tear down the connection. A timer is usually started after each handshake packet is sent. If no expected packet has arrived by the time the timer expires, the previous handshake packet will be retransmitted.

For error detection and recovery, general purpose protocols use checksums to detect corrupted packets, and use sequence numbers to detect duplicated or out-of-sequence packets. A go-back-N or selective-repeat scheme is employed for error recovery; go-back-N is favored for multicast operation, and selective-repeat is favored for long delay virtual circuit transmission. Figure 4 shows how go-back-N and selective-repeat operate [26].

For flow control, general purpose protocols use an explicit sliding window strategy which means the available

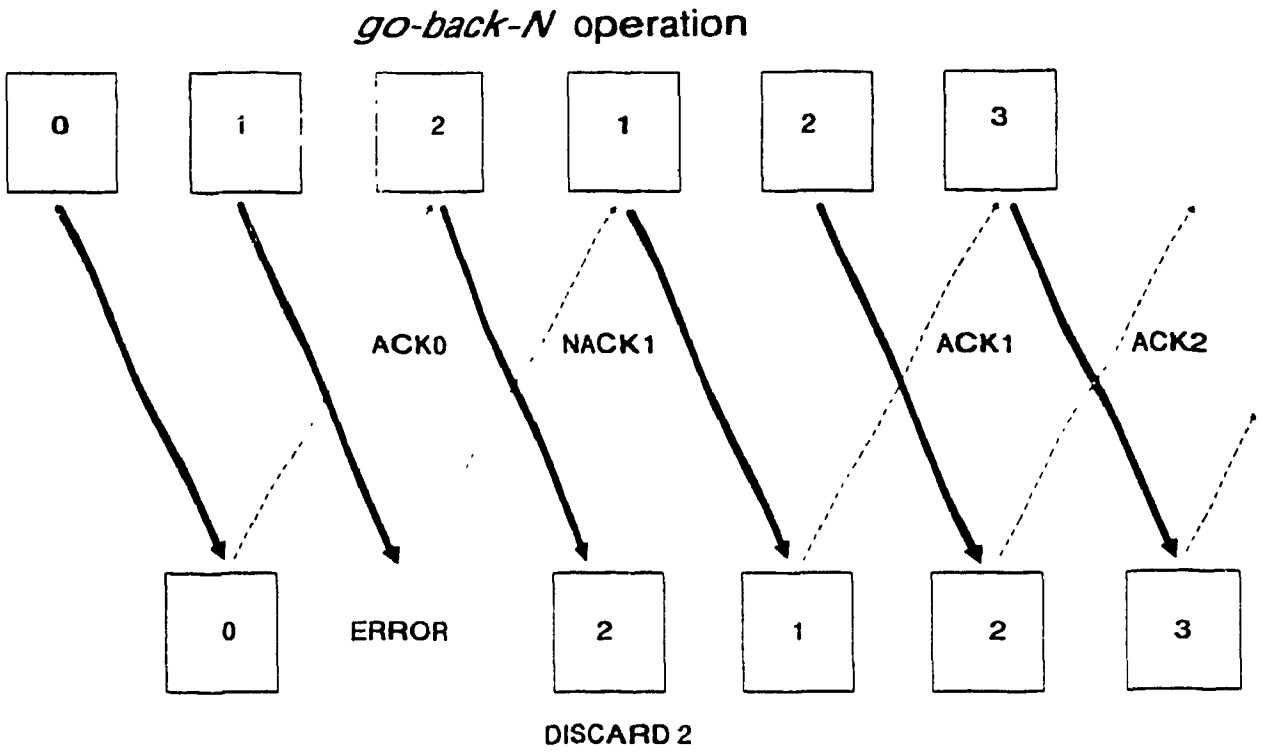


Figure 4 Error recovery methods

buffer space is updated with the arrival of each acknowledgment packet.

For message fragmentation, general purpose protocols use fixed packet header and fixed user data segment, or variable size in control packets and fixed size in user data packets.

We will now discuss two well-known general-purpose transport protocols.

2.3.2.1 The Department of Defense's Transmission Control Protocol

Transmission Control Protocol (TCP) was developed for the U.S. Department of Defence (DoD) in 1981 [13], and later became a DoD standard for communication. TCP is probably one of the most popular transport protocols, and it has been adopted by many experimental distributed systems, because of its availability on the Internet and its relationship with the BSD version of the UNIX operating system.

Figure 5 shows the minimum packet exchange needed to transmit one packet of user data. There are three phases in each transaction: connection establishment, transmission, and disconnection.

To send a message, a client must set up a virtual circuit connection first, before the message can be sent. The virtual circuit should be torn down when it is no longer needed.

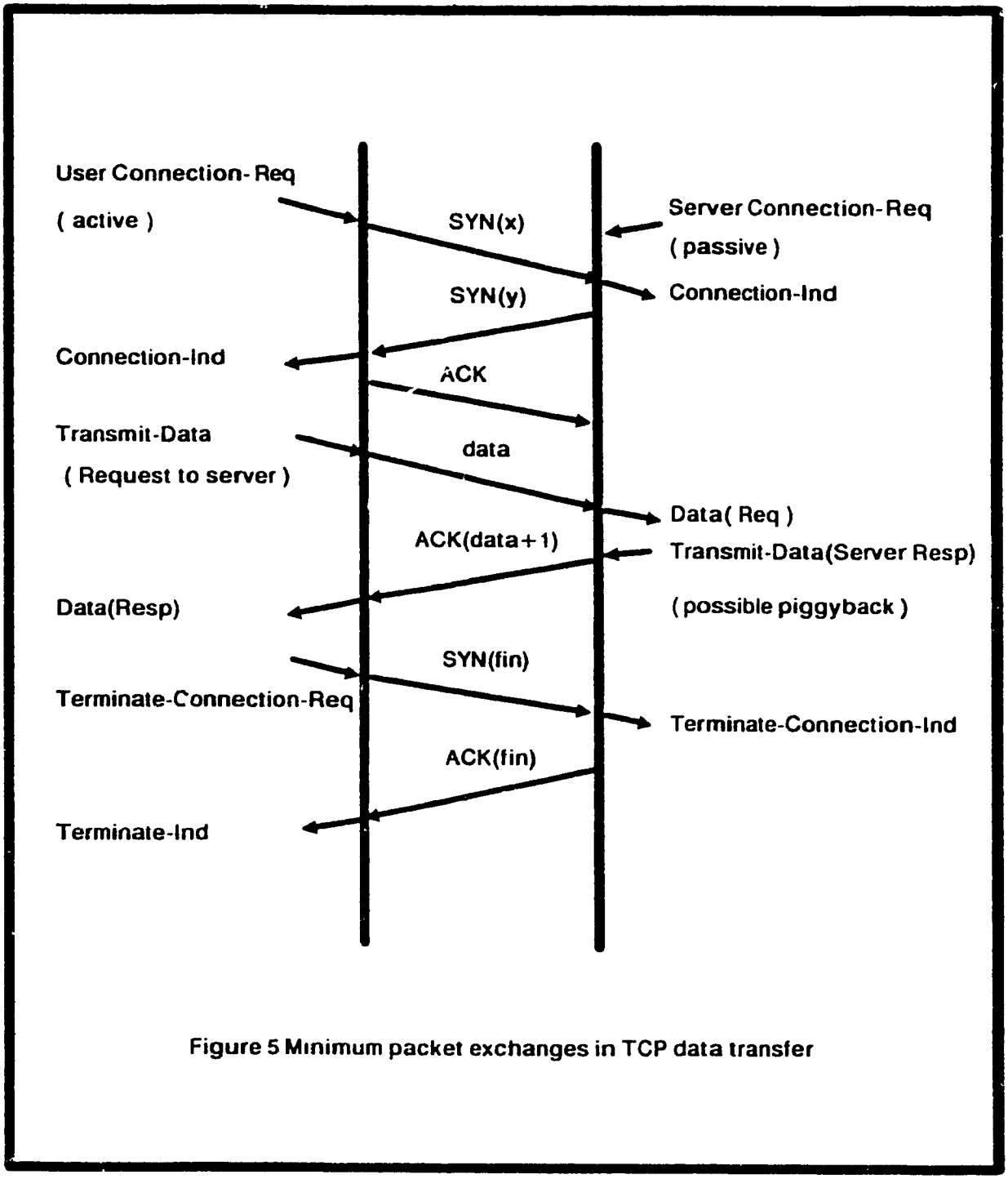


Figure 5 Minimum packet exchanges in TCP data transfer

When a send message request is passed to TCP from a user, TCP will fragment the message into TCP packet(s). Then TCP will send a "connection request" packet (SYN) to the target site; in the meantime, a wait timer is started at the sender site. If no "connection confirm" packet arrives before the expiration of the wait timer, TCP will retransmit the "connection request" packet. If the "connection confirm" packet arrives before the wait timer expires, the sender site will transmit an "acknowledgment packet" which acknowledges the receipt of the "connection confirm" packet to the receiving site. The process described above is usually referred as the three way handshake. During the handshake, transfer and resource allocation are negotiated between sender and receiver. Theoretically, TCP can safely exchange a request and a reply using five packets (without disconnection); many implementations, however, require nine [24]. Once the virtual circuit is set up, the actual transmission of the user data will take place. An "acknowledge packet" must be sent to the sender after the receiver has received a certain amount of data. When the user wishes to terminate the transmission, TCP will send a "disconnection request" to the receiver site. If it receives a "disconnection confirm" packet from the receiver, it will shut down the virtual circuit.

TCP also provides error detection, error recovery, and flow control functions for the user. Checksums and sequence numbers are used to detect corrupted, duplicated, and out-

of-sequence packets. In some of the TCP implementations, selective-repeat is used for long distance file transfer and go-back-N is used for short distance communication, simply because the cost of using go-back-N for long distance communication is too high. However, there is some overhead caused by using selective-repeat, since the receiver site must have the ability to store out-of-sequence packets and to restore the sequence later. These requirements make the receiving protocol more complicated. For flow control, TCP uses an explicit sliding window technique. The main shortcoming of the explicit sliding window is the possible occurrence of the "silly window syndrome" [10] which is caused by the window size becoming as small as one byte; in this situation, only one byte is allowed to be transmitted in each packet.

User messages are fragmented into TCP packets with fixed format headers; this introduces additional overhead in packets, because some control fields are not necessary for data packets.

2.3.2.2 The ISO TP4

ISO developed Transport Protocol class 4 (TP4) in 1984 [18]. TP4 design is based on assumptions that are similar to TCP's. As a result, TP4 design is very much like TCP's, the only difference being that TP4 uses variable size control packets in an attempt to cut down on overhead bits in the packet format. The variable size control packets also make the receiver side more complicated, and therefore

reduce the throughput. Figure 6 shows the minimum packet exchanges to send a request. There are very few actual implementations of TP4 in the current market. As a result, there is no distributed system adopting TP4.

In general, TCP and TP4 are very robust in a poor communication environment as usually exists in a long haul network. Once the virtual circuit is set up, the data transfer becomes very efficient [24].

On the other hand, TCP and TP4 are extremely inefficient when they are used for exchanging small amounts of data as is frequently the case in request-reply style communications. The main causes are:

- excessive packet exchange in virtual circuit establishment;
- excessive packet exchange in virtual circuit disconnection;
- possible silly window syndrome in flow control;
- no reliable real-time arbitrary-sized datagram service;
- requirement for three phases per transaction.

Some experimental distributed systems use a modified version of the general purpose protocols in order to gain better performance [1,19,34].

2.3.3 Special Design of Transport Layer Protocols

With the development of distributed systems, the use of communication has been shifted significantly to intra-system

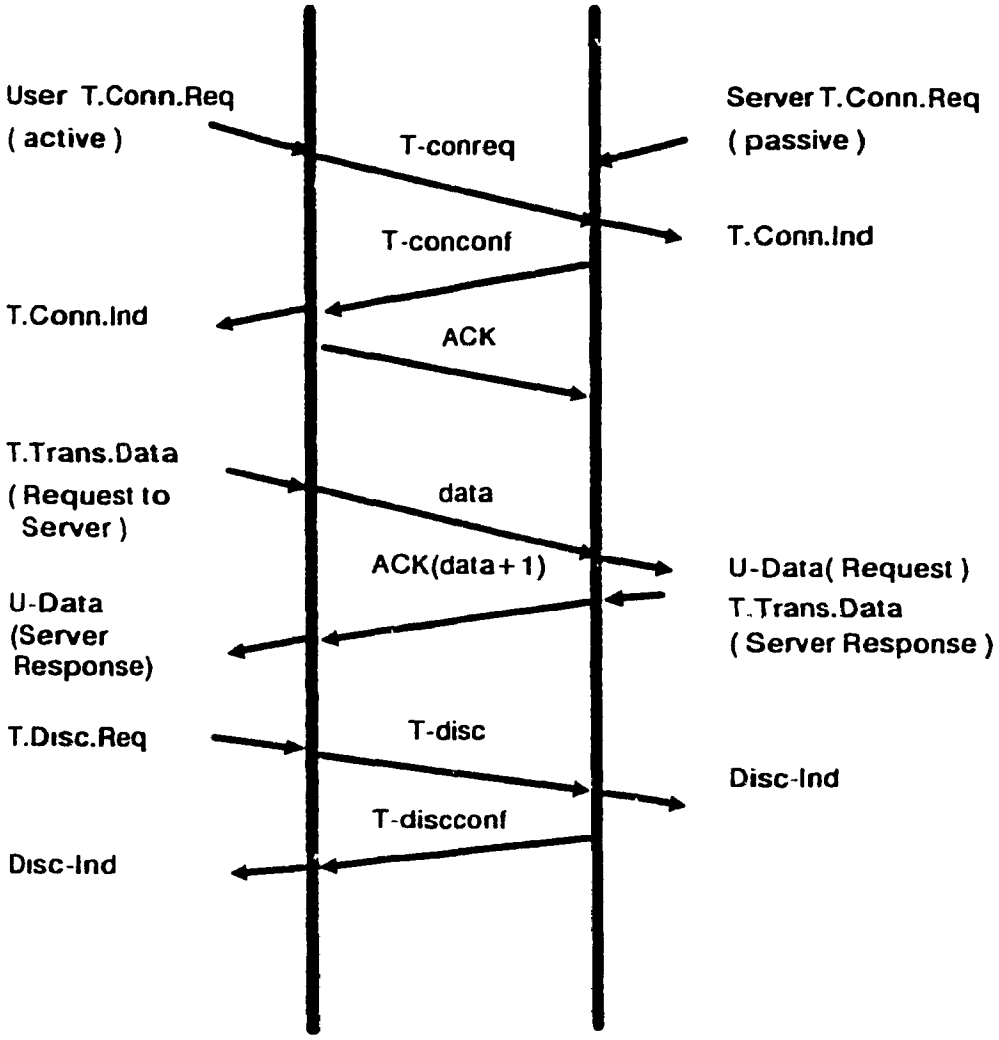


Figure 6 Minimum packet exchanges in TP4 data transfer

communication such as remote procedure calls, multicast, and real-time datagrams [4]. Most of the intra-system communication can be characterized as a short burst of packets that carry the requests and replies.

From the previous discussion of general purpose transport protocols, we can see that they are inadequate for request and reply style communication. Thus many of the experimental distributed system implementations have adopted a different type and much simpler protocol for their communication [27]. These newer protocols are classified as special purpose protocols.

Special purpose protocols are designed to achieve high performance for distributed systems. Their design assumes that the system is based on a local area network, and that the behavior of such a configuration is predictable. With these assumptions in mind, many mechanisms that are used to cope with an unpredictable environment are removed.

In the following we present two examples of special purpose transport protocols. Our criteria for selecting the examples are:

- The protocol must be designed specifically for distributed systems.
- It must be designed from scratch.
- It must be implemented.
- It must have performance statistics available.

2.3.3.1 The Versatile Message Transaction Protocol (VMTP)

VMTP was developed at Stanford University and used in the V distributed system. VMTP is basically a request-response protocol. The designers of VMTP assume that the protocol would be used in a local area network or tightly-coupled cluster of local networks [3].

A VMTP message transaction is initiated by a user sending a request message to a server entity, and terminated by the server sending back a response message [4].

The handshake mechanism for virtual circuit establishment is not adopted by VMTP; instead, a much simpler, timer-based connection set up method is used:

When a client wishes to request a particular type of service, he or she formats a VTMP control block which specifies the name and address of the server, and passes the control block to the VMTP entity. VMTP will fragment the request message into several fixed-length VMTP packets if the message is too long. Then, it will send all the request packets to the server site in a short burst. A wait timer is also started right after the sending action. The wait timer is set to the interval of a packet round trip time plus the expected server processing time. If the sending VMTP entity does not receive any response packet from the server site before the wait timer expires, it will retransmit the previous request again. VMTP will declare network failure after six unsuccessful tries.

At the server site, the server is always listening. Once VMTP receives a new request, it will set up a virtual circuit associated with the request, and will pass the request to the server process. At the same time, the server site VMTP entity will also start a wait timer which will expire in the length of expected response time. If there is no response from the server by the time the wait timer expires, VMTP will discard the virtual circuit to prevent possible duplicated requests, because the reply would not able to reach the client site before the wait timer on the client site expires.

There is no specific packet exchange needed for the termination of the virtual circuit. A timer called T-stable timer is started when the virtual circuit is set up. The time interval of the T-stable timer is six packet round trip times plus six expected response times. If the virtual circuit is not active in the period of T-stable, it will be automatically torn down by the VMTP entity, and its identifier will be used by another client.

There are a number of control bits in the VMTP packet header that allow the sender entity to control when the receiving site should return an acknowledgment. In this way, one acknowledgment packet can acknowledge more than one packet. If the number of the request packets is small, VMTP could set no acknowledgment. In this case the response from the server will be the acknowledgment. Using replies for acknowledgment purposes is usually referred to as "implicit

ack". VMTP only allows one outstanding request per client; as a result, there is no need for multiconnection management for each client, which makes the receiving site simpler.

VMTP uses checksums and sequence numbers for error detection. For long transmissions, selective-repeat is used for error recovery [4].

The VMTP Packet header is smaller than that of most of the general purpose protocols because of the reusable virtual circuit identifiers. Figure 7 shows the operation of VMTP.

In general, VMTP designers made maximum effort to reduce the number of packets required for virtual circuit connection management, error control, and flow control. Thus VTMP has better performance than most of the general purpose protocols in a LAN environment.

However, there are also inefficient aspects in VMTP. VTMP does not perform well when the client sends a large number of requests within a time interval which is slightly greater than T-stable, because the virtual circuit would have to be set up with each request. VMTP is also difficult to tune, because there are many timers, especially the round trip timer and the expected response timer. This problem of tuning makes protocol adaptation and open connection difficult. VMTP is also not suitable for long file transfers, because VMTP only allows one outstanding request per client, and the maximum user data length in each request control block is 32 Kbytes.

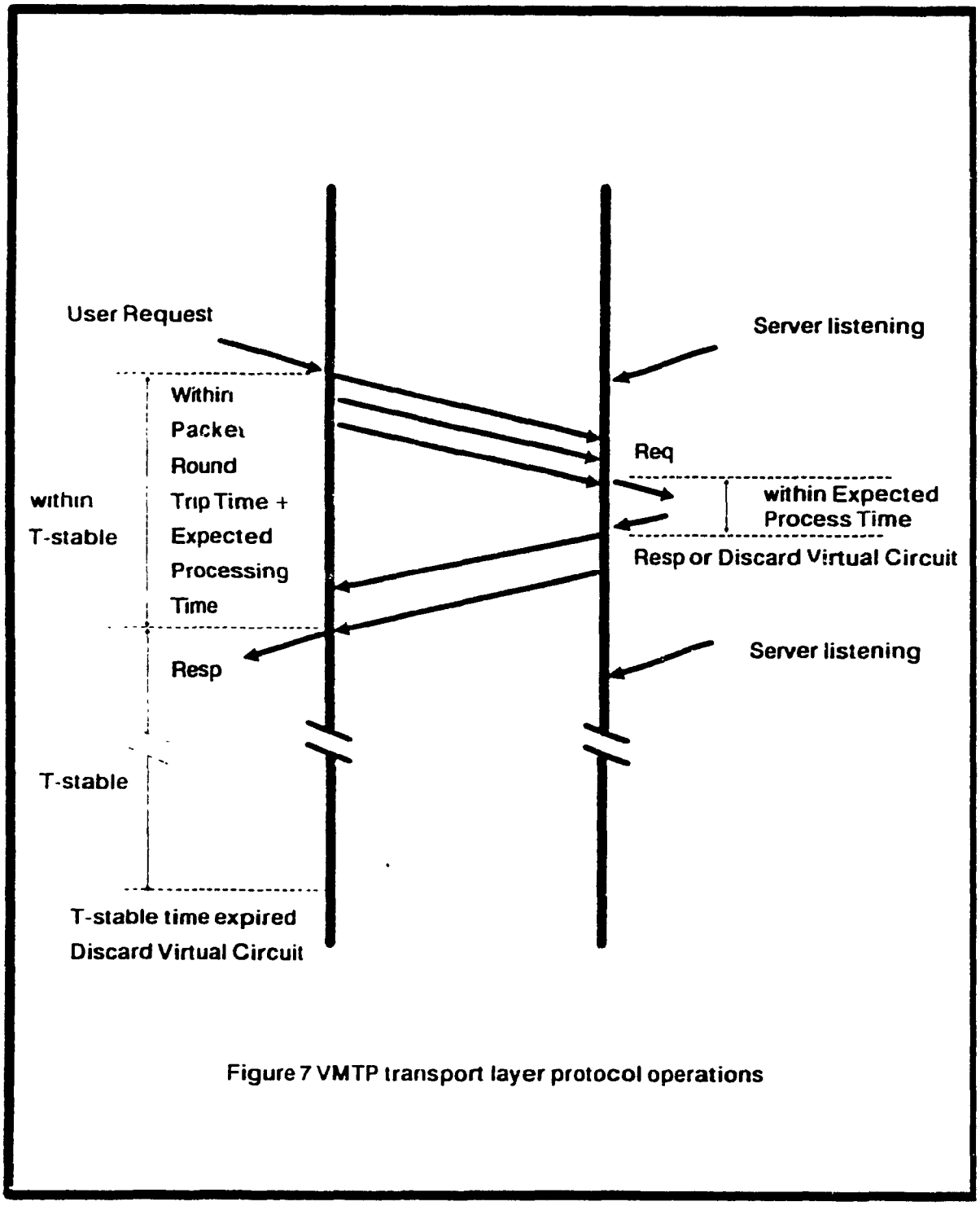


Figure 7 VMTP transport layer protocol operations

2.3.3.2 The AMOEBA transport protocol

The AMOEBA distributed system was designed and implemented at the Vrije University in Amsterdam [30].

To achieve high performance, the AMOEBA transport protocol has been kept very simple. Figure 8 shows a request-reply operation in the AMEOBA protocol.

There is no virtual circuit in AMOEBA's protocol. When a client wants to send a request for the first time, a packet containing the server's port number is broadcast over the network. The kernel running the server responds with a packet containing its physical network address. The client caches this information so that it may use it as a hint in subsequent transactions to the same server [30].

Next the client sends the request packet, or a sequence of packets if the request does not fit into one packet, to the server using the acquired physical address. A retransmission timer is started to recover from network failures.

In the case where the reply is not generated quickly enough, the server sends back an acknowledgment to prevent the client from retransmitting the request [30].

There is no network level routing since a broadcast technique is used.

For flow control, the AMOEBA protocol simply discards the overflow packets, because the transmission rate is known in the design. For error control, checksums and sequence numbers are used.

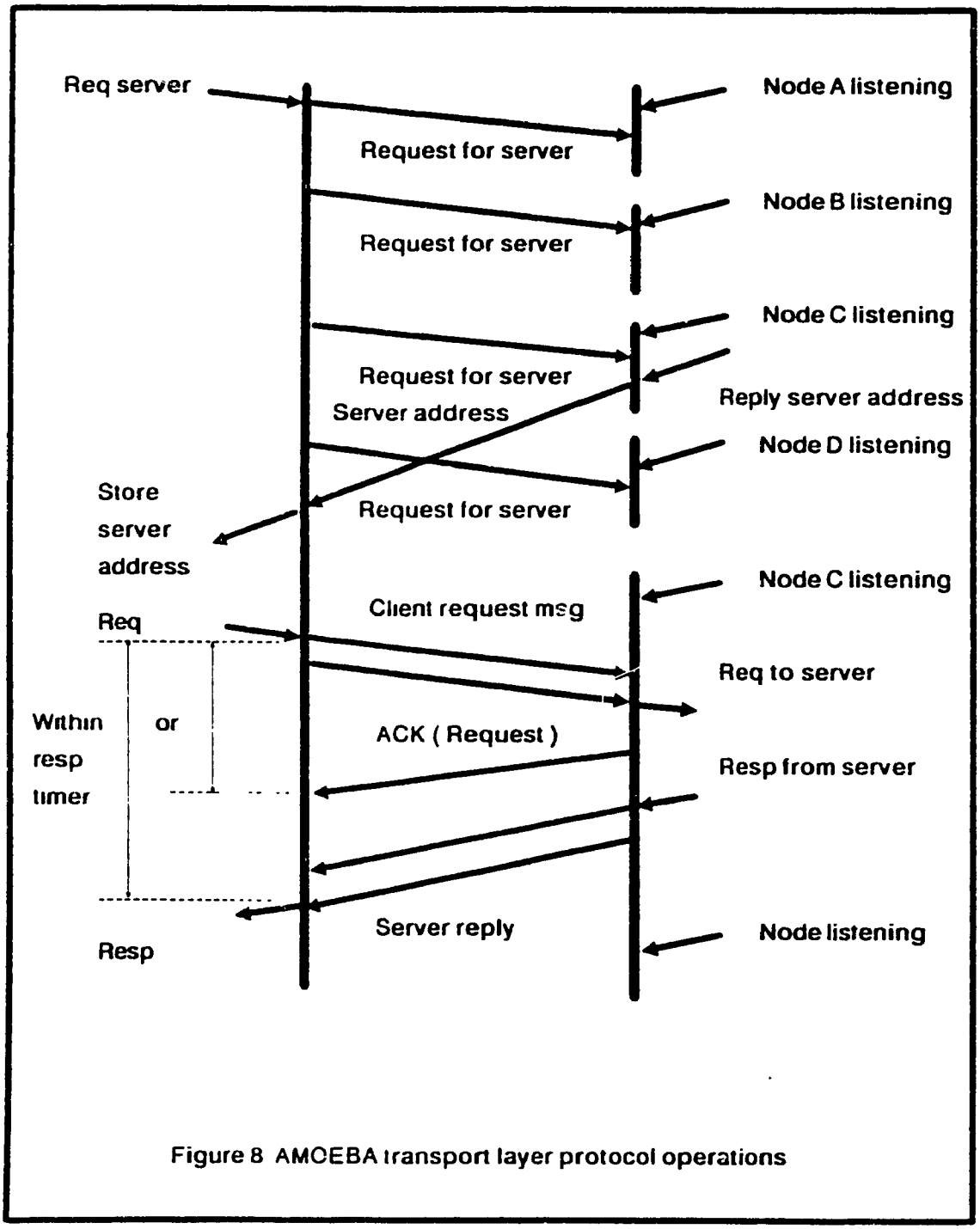


Figure 8 AMOEBA transport layer protocol operations

The AMOEBA protocol is extremely simple, which leads to high performance in a stable local area network environment [30].

However, the AMOEBA protocol is not suitable for long file transfers, because there are no virtual circuits, and there is no adequate flow control. There is also the problem of connecting to other systems.

To summarize the main design characteristics of special purpose protocols, they achieve high performance in intra-system communication by using a minimum number of packets for connection management or a connectionless protocol, and by using one-to-many acknowledgment or implicit acknowledgment for error and flow control.

2.3.4 Performance Comparisons from the Literature

To compare the performances of the general and special purpose protocols in distributed systems, we selected a number of systems. The criteria for our selection are:

- The system must use an ETHERNET for communication.
- The system must be configured with similar hardware.
- The system must be designed as a full system.
- The protocol used by the system must be known.
- The performance statistics must be available.

Although there are many reports on experimental distributed systems, most of them do not provide performance figures, e.g., [1,19,22]. Some systems are tightly coupled multiprocessor distributed systems, e.g., [29]. For some systems, information is provided about their protocol and

performance figures, but the protocols are just variations of the general purpose protocols, e.g., [23]. Some systems are designed for a special purpose such as [17]. Some systems use different medium access methods, e.g., [22]. Most of the papers provide performance information on intra-system communication of their systems, but not on long file transfer performance.

The systems that meet the selection criteria are:

- DUNIX and SPRITE for general purpose protocols
- V and AMOEBA for special purpose protocols.

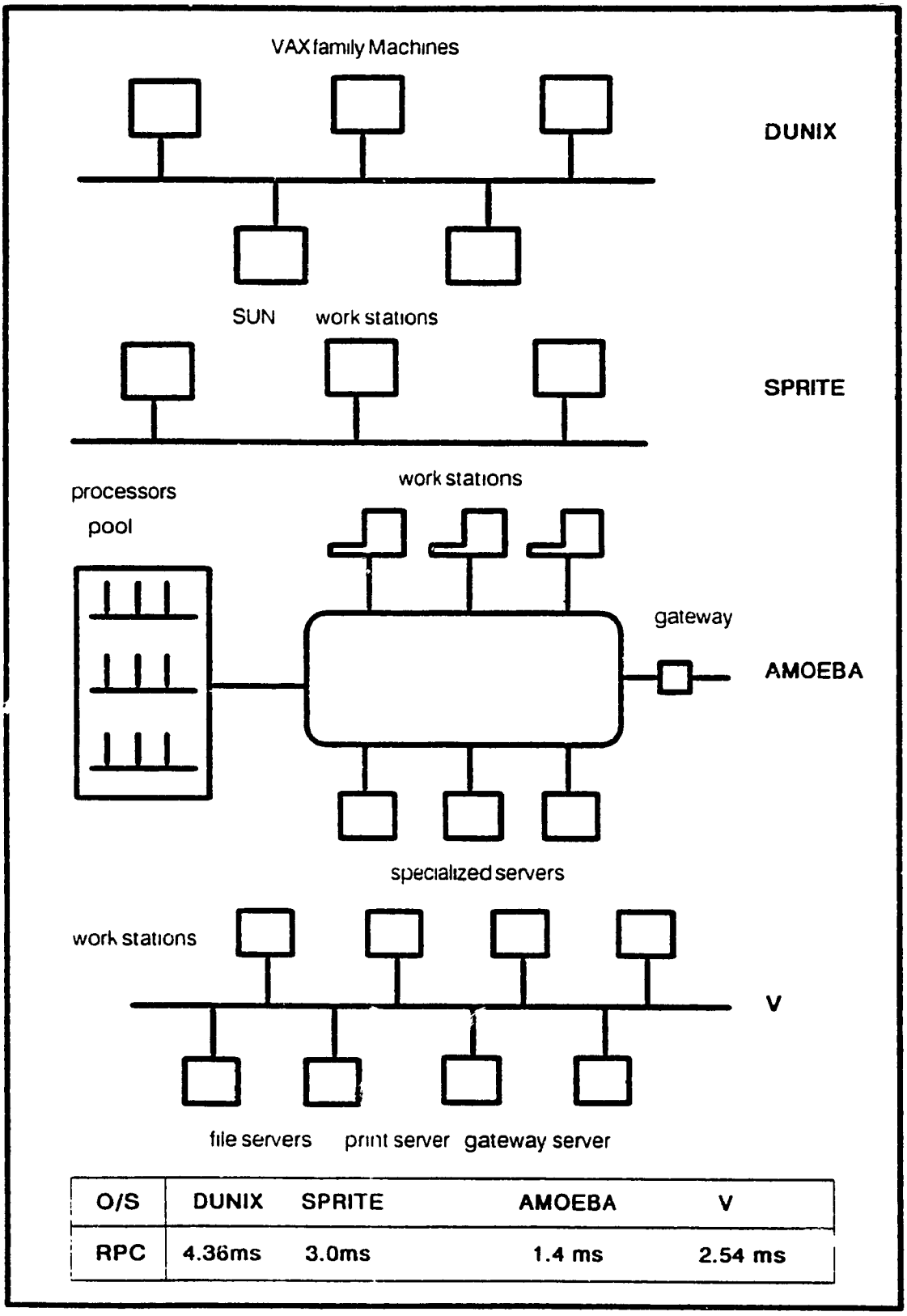
DUNIX was developed by Bell Communication Research. DUNIX currently runs on the DEC VAX family of computers and uses the Ethernet for inter-computer communication. The protocol is adapted from DECnet [20].

SPRITE was developed by UC Berkeley. SPRITE runs on a number of SUN stations that are connected by Ethernet. The communication protocol in SPRITE is TCP [32].

The configuration of the V distributed system is similar to SPRITE: it is also a number of SUN workstations connected by Ethernet. The communication protocol is VMTP [3].

AMOEBA currently runs on Motorola 68020, MicroVax II, and National Semiconductor 32032 processors using Ethernet and Pronet LAN [30]. The transport protocol is described above.

Figure 9 shows the system configurations of the four examples and their performance on remote procedure calls.



O/S	DUNIX	SPRITE	AMOEBA	V
RPC	4.36ms	3.0ms	1.4 ms	2.54 ms

Figure 9 System configurations and RPC performance

From the comparison, we can see that special purpose protocols do perform better in intra-system communication than general purpose protocols. On the other hand, we have noted above that general purpose protocols are better suited for long file transfers. We also know that none of the above four protocols provides reliable datagram and mulitcast services to their user.

Currently much effort has been put into improving general purpose protocols. Examples of such efforts are [2,11]. Also there are many activities on the area of designing new transport protocols that are suitable for inter-system and intra-system communications. The next chapter outlines one such effort.

CHAPTER 3

AN INTRODUCTION TO XTP

"Small is beautiful."

Schumacher's dictum

3.1 Background

The newly designed Xpress Transfer Protocol (XTP) aims to combine the advantages of both the general-purpose protocols and the special-purpose protocols [7,8,9].

The XTP protocol has been designed under the leadership of Dr. Greg Chesson. The protocol provides high efficiency for bulk transport, real-time datagrams, and traditional stream services. It also has flow/error/rate control, accommodation of multiple addressing schemes, message boundary preservation, out-of-band signalling, and a reliable multicasting service [8]. XTP has the functionalities of the transport and network layer in the ISO reference model. The protocol is designed to meet the demands of the VLSI execution environment [7,8,20]. The XTP network layer could also operate as a bridge or routing gateway. The core of the XTP protocol is a minimal mechanism, or the so-called lightweight transport [8]. The work reported in this thesis is based on version 3.3 of the protocol definition [8]; occasional reference will be made to new features defined in version 3.4 [9].

3.2 The Operation of XTP

For the purposes of helping the reader to understand

our simulation, we are only going to discuss those XTP operations that have significance to our simulation, rather than the whole detailed definition of the protocol. Those who are interested in the complete definition should refer to [8]. Figure 10 is a summary of the XTP packet structure.

In the general sense, there are two types of XTP packets:

(1) Control packets, which are used by the XTP protocol to send control and management information between the XTP peer entities. XTP control packets do not carry any user data.

(2) Information packets, which are used to carry user data.

Every XTP packet has the same header and trailer; both are sixteen bytes long. The control segment is 160 bytes long. In our simulation, the user information segment is varied from 6 to 1442 bytes long, which fits into the ETHERNET user data segment.

The XTP packet header has six fields:

(1) The options field which has sixteen bits. Table 1 is a summary of the meaning of the bits in the options field (the bit count starts from the least significant bit, which is bit 0 in this case). Here, we are not going to describe the details of the options field, because it does not relate to our current simulation. A more detailed definition of the options field can be found in [8].

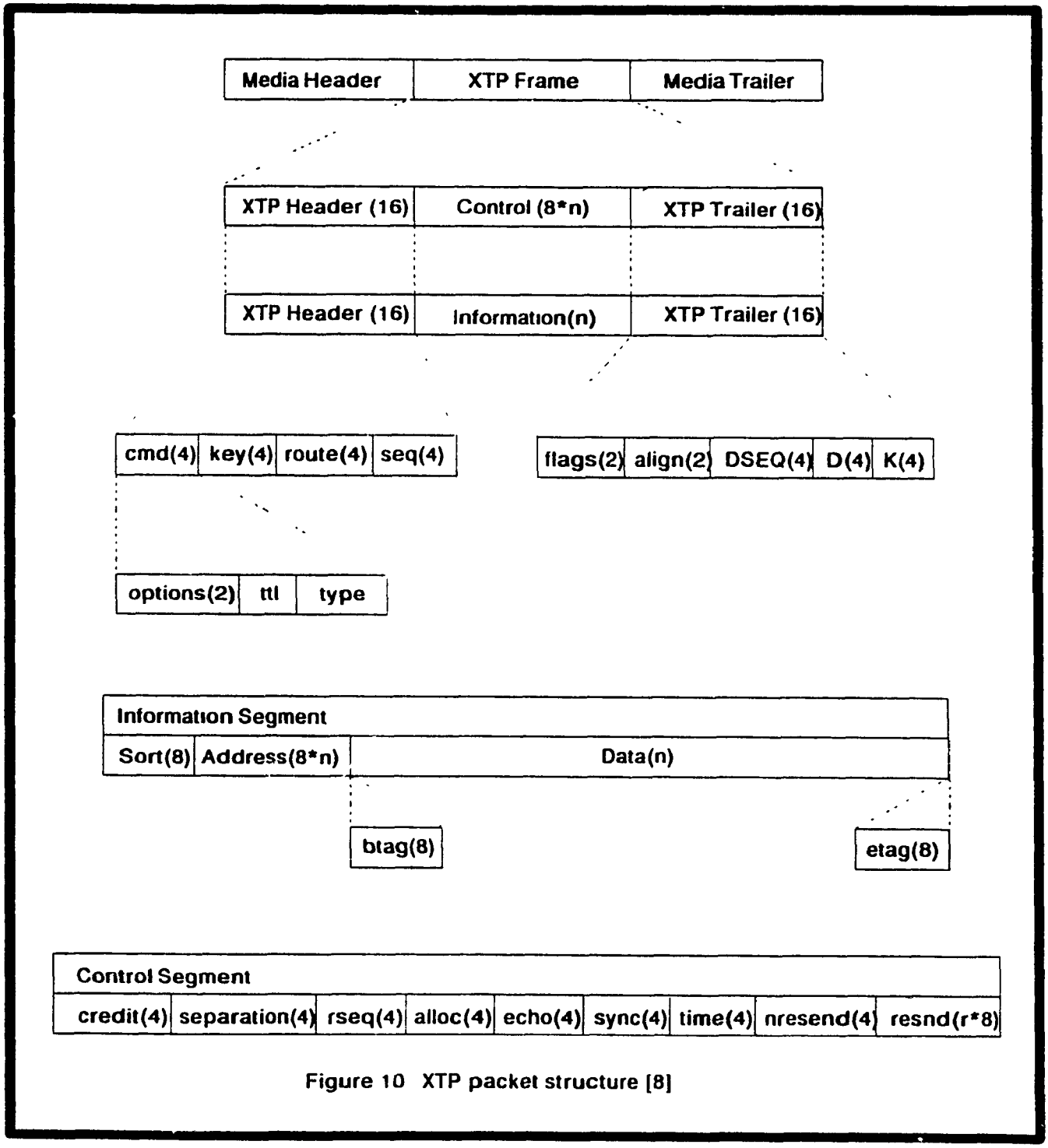


Figure 10 XTP packet structure [8]

BIT POSITION	MEANING
0	SORT field exists in the data segment
1	BTAG field exists in the data segment
2 - 7	Not used
8	Check function disabled
9	Multicast mode enabled
10	Reservation mode enabled
11	No-error-check mode enabled
12	direct addressing mode enabled
13	32-bit mask enabled
14	64-bit mask enabled
15	Little-endian byte order used

Table 1 Bit meanings for the XTP option field

(2) The ttl field records the total lifetime allowed for a packet in a network.

(3) The type field consists of eight bits, shown in Table 2.

BIT POSITION	MEANING
0-3	Packet type
4-6	Version number
7	Little-endian bit order used

Table 2 Bit meanings for the XTP type field

(4) The route field is used for routing information.

(5) The key field is used for context association.

(6) The seq is the packet sequence number.

The XTP protocol has nine packet types:

(1) DATA - data packet;

(2) CNTL - control packet;

(3) FIRST - the first packet;

(4) REJ - error indication packet;

(5) PATH - path threading packet

(6) DIAG - diagnostic packet;

(7) MAINT - network maintenance packet;

(8) MREPLY - multicast reply packets;

(9) MGMT - management packet.

The XTP packet trailer has five fields:

(1) The flags field contains a number of command flags. Table 3 shows the definitions of the trailer flags [8].

BIT POSITION	MEANING
0	The last packet indication
1	The end of message indication
2	The end tag indication
3-10	Not used
11	The data checksum present
12	The write site closed indication
13	The read site closed indication
14	The end of burst indication
15	The status request indication

Table 3 XTP packet trailer flags

(2) The align field records the number of padding bytes present.

(3) The dseq denotes that the sender requests that status be sent after the data have been copied to user space.

(4) The dcheck is the checksum calculation on the information segment.

(5) The kcheck is the checksum calculation on the header and trailer.

XTP control packets do not carry any user data; each of the control packets has ninety-six bytes of control segment within which there are eight control fields plus sixteen retransmission pairs. The following is a summary of the functionality of each field (from [8]):

(1) Credit field; unlike the concept of credit that is used in the explicit sliding window scheme, credit in XTP is used to control the maximum number of output bytes per burst allowed for each virtual circuit in each node. The credit serves the function of enforcing fairness among the senders in a node.

(2) The separation field specifies the pause space between packets that are going on the same route.

(3) The rseq field states the highest sequence number that the receiver has received in consecutive form. In other words, there is no packet missing up to the rseq number.

(4) The alloc field indicates the maximum number of bytes that the receiver is willing to accept. The

alloc function is the same as the window size in the sliding window scheme.

(5) The echo field returns the same value that the sender has specified in the sync field in the previous control packet. Once XTP receives a control packet that requests the current status of the receiving site, the XTP receiver will report it s current status and will copy the sync value from the request control packet to the echo field of the return control packet.

(6) The sync field contains an arbitrary value that the sender can put into a request control packet. The sender, then, will check each value in the returned control packet to ensure that the returned control packet corresponds to the most recent request.

(7) The time field states the received time of the control packet, this is used by the sending site to determine the round trip time.

(8) The nresend field specifies the number of retransmission pairs used in the control packet.

Each of the above eight fields is four bytes long.

Following the fields is the resend pair array which specifies the received packets from whose complement the sending site can determine the missing packets. Note: the function of the resend pairs described here is from [9]. The error recovery process will be discussed later.

3.2.1 Connection Management

XTP, unlike TCP and TP4, uses a single packet to establish a virtual circuit connection. When a user passes a message in a XTP control block format to XTP, XTP will check if there is a context which has already been created associated with the destination. If there is no such context, XTP will create a context record, otherwise XTP will start to send the message immediately. To send a message which does not have a previous established virtual circuit, XTP will start by formatting an XTP packet, which may contain the entire message. XTP will then set the packet type to indicate that this is a "first" packet. At the reception site, there must be a user who is willing to receive the incoming message, and to receive it the user passes a control block with receiving buffer to XTP. Once XTP receives a control block that requests a passive listen, XTP will create an idle context which is associated with the listening address. When XTP receives a packet with the first packet bit on, XTP will search its data base to attempt to match to a context that is waiting for the received packet; XTP will then set the passive listen context to active and thereby a virtual circuit is set up. Figure 11 shows the process of virtual circuit establishment.

3.2.2 Data Transfer Operation

XTP fragments user messages into smaller packets depending on the media. Once a transmission on a virtual

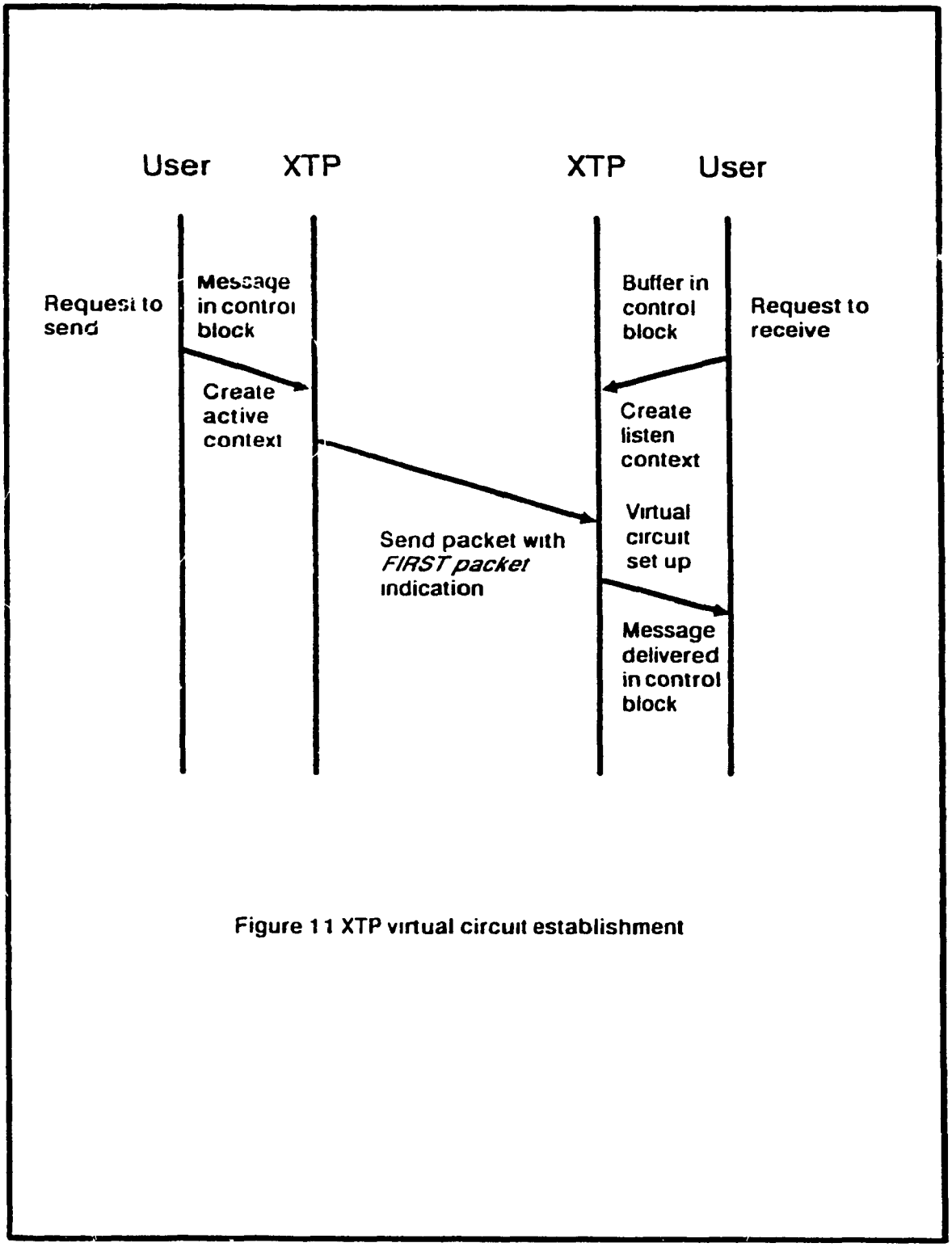


Figure 11 XTP virtual circuit establishment

circuit is started, XTP will send user data up to the limit given by the alloc or credit value, whichever is the smaller one. If one of the limits is reached, XTP will turn the SREQ bit on in the last packet going out. When the receiver receives a control packet with the SREQ bit on, it immediately sends back a control packet with its current receiving status. On the other hand, if a data packet has the SREQ bit on, the receiver will deliver the data to user space before generating a status control packet.

3.2.2.1 Flow Control

Initially, XTP uses the default value for the receiver allocation space (alloc). Once the receiver has copied data to the user's buffer and the user has passed a new buffer to XTP for further reception of data, XTP at the receiving site will update the alloc value. Then on the request for status from the sender, the XTP receiver sends a control packet with the new alloc value.

For every packet that XTP receives, the sequence number of the packet is checked against the alloc and rseq values. If the packet's sequence number is greater than the alloc value, which means that the receiving user does not have enough space to receive the packet, the packet is discarded silently. If the packet's sequence number is smaller than the rseq value, which means the packet is a duplicated one, it is also discarded silently. If the packet sequence number passes both tests, and it is the same as the rseq value, XTP

will copy the data in the packet to user space and will signal the user the arrival of the new data. The XTP receiver, unlike TCP and TP4, does not report its window size (alloc) on a fixed period, rather it only reports the window size when the sender requests it. Figure 12 illustrates the operation of flow control in XTP.

3.2.2.2 Error Recovery

If a packet sequence number is greater than the rseq value and is smaller than the alloc value, this means that there were packets missing during the transmission, or the previously received packet did not pass the checksum. The XTP uses a selective-repeat method for its error recovery in non-multicast operation. Once an error is detected, XTP records the sequence number of the new packet in the resend pair. (The XTP protocol definition 3.3 specifies that the "resend pairs" records the missing data segment, but in the revision 3.4 the "resend pairs" are used to record the received data segment.) For example, if the expected sequence number (rseq) is 100, and the newly received packet sequence number is 200, and the packet length is 100 bytes, then the resend pair will be recorded as 200, 300. The value of 300 is calculated by adding 200 to the packet length which, in this case, is 100. Also the nresend value will be incremented by one. XTP only allows for one outstanding reject (REJ) packet per virtual circuit. If there is no outstanding REJ packet, then XTP sends a REJ packet to the sender upon the detection of the error. If the sequence

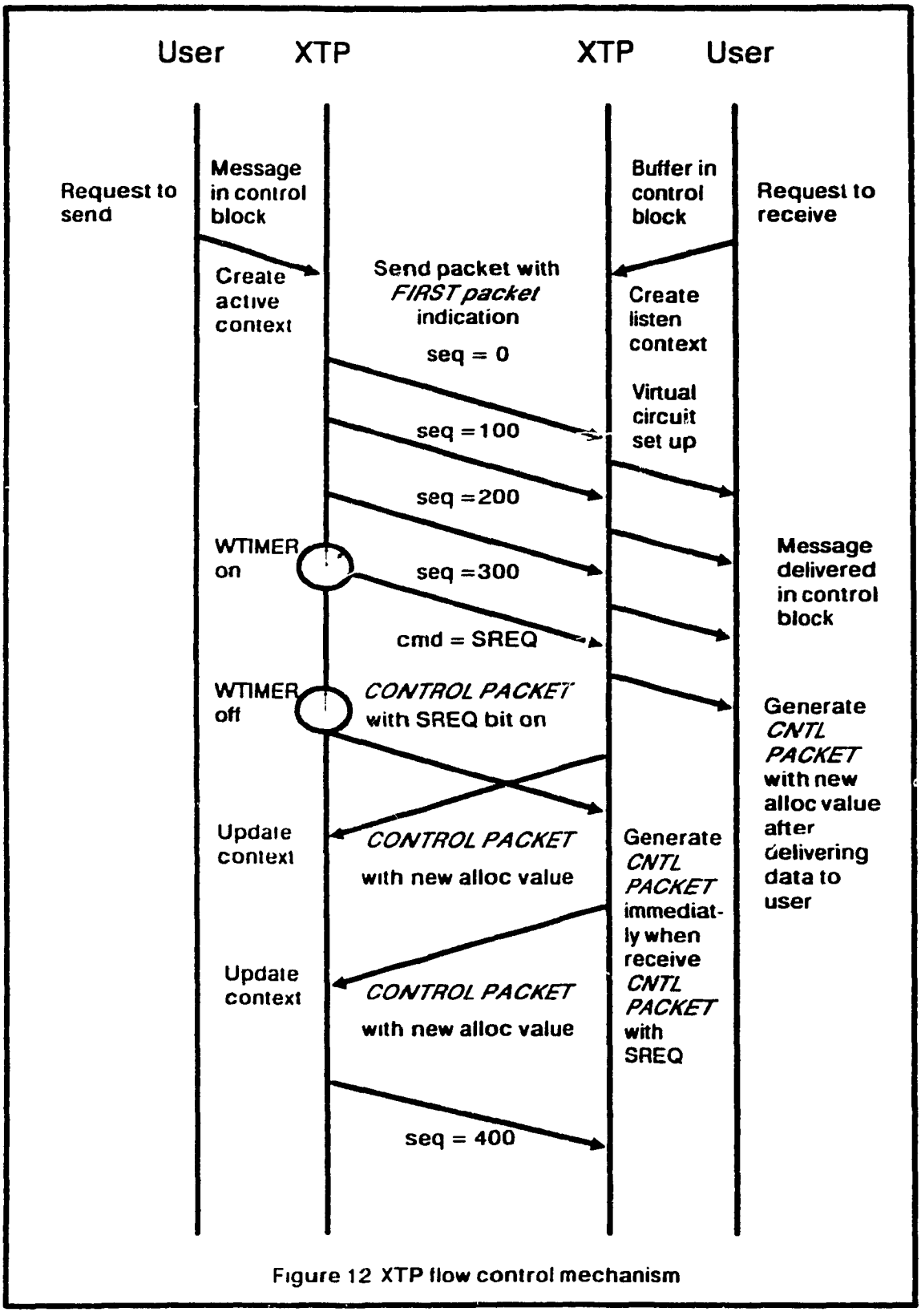


Figure 12 XTP flow control mechanism

number of the newly received packet is equal to rseq, and if there is a resend pair for this packet, then the rseq will be set to the end value of the resend pair, and nresend will be decremented by one. Using the previous example, upon receiving a packet with the sequence number 100, the rseq will be set to 300, and the nresend value will be 0. Figure 13 shows the operation of error recovery in XTP.

3.2.2.3 Separation Control

XTP employs a separation control mechanism for congestion control. The difference between the separation control and flow control is that the separation is used to control the number of bytes that a sender can send to the same route in a fixed period, and the flow control is used control the number of bytes that a user can send to the receiver. XTP performs the separation control on a per route basis in each node: XTP pauses for the length given by the separation value between packets that go on the same route. The separation control allows XTP to be used as a gateway protocol. AMEOBA and VMTP do not have any rate control mechanism, thus their usage is limited to LAN based applications.

3.2.2.4 Credit Control

XTP uses credit control to provide fair service among the virtual circuits. A virtual circuit, in XTP, is allowed to send up to the maximum number of bytes specified by the credit field in the virtual circuit context. For every

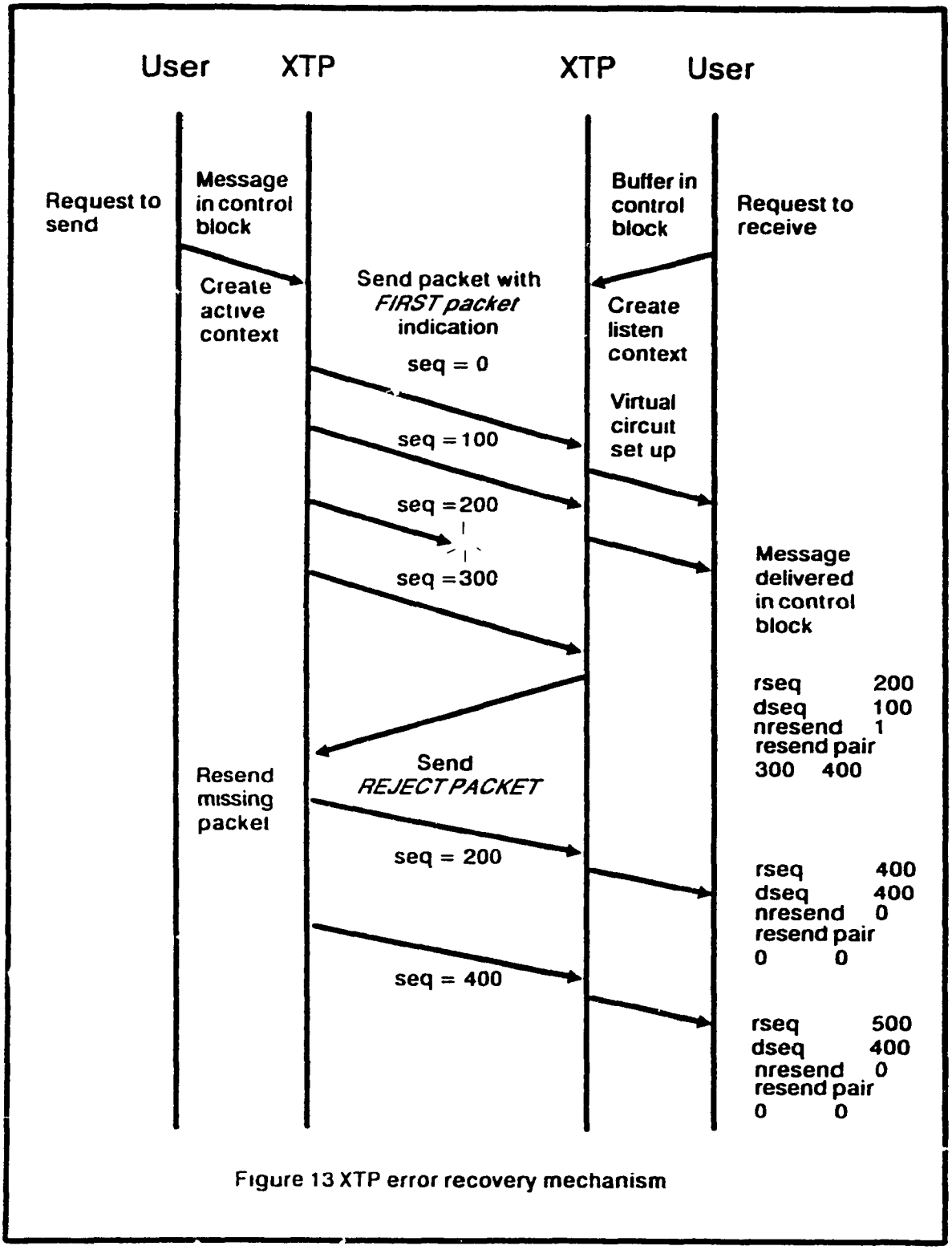


Figure 13 XTP error recovery mechanism

packet sent, XTP decrements the credit field by the bytes sent. Once a virtual circuit reaches its credit limit, it must wait until a new credit value is assigned to it. The credit field of every active virtual circuit is updated every sixtieth of a second. Neither AMEOBA nor VMTP provide any fairness control to the users, so it is possible that users could be starved indefinitely.

3.2.3 Termination Management

There are two ways to terminate an XTP connection:

(1) The user at the sending site passes a control block holding a write close command (WCLOSE) to XTP. When it receives WCLOSE, XTP sends a packet with the WCLOSE and SREQ bits being turned on in the trailer to the receiving site. If the receiving site does not require any retransmission, it will generate a control packet which reports the status and indicates the receiving site close (RCLOSE); then the receiving site will release the context. Otherwise, the sending site will perform the necessary resend; then it will try the WCLOSE process again.

(2) If the user at the receiving site decided that the incoming data were not interesting, he or she could pass a control block holding the read close command (RCLOSE) to XTP. XTP will send a control packet with the RCLOSE bit turned on to the sender. The sending site XTP will inform the user that the receiving site does not want to receive any more data and then the XTP will discard the virtual circuit context. Figure 14 shows two types of closing

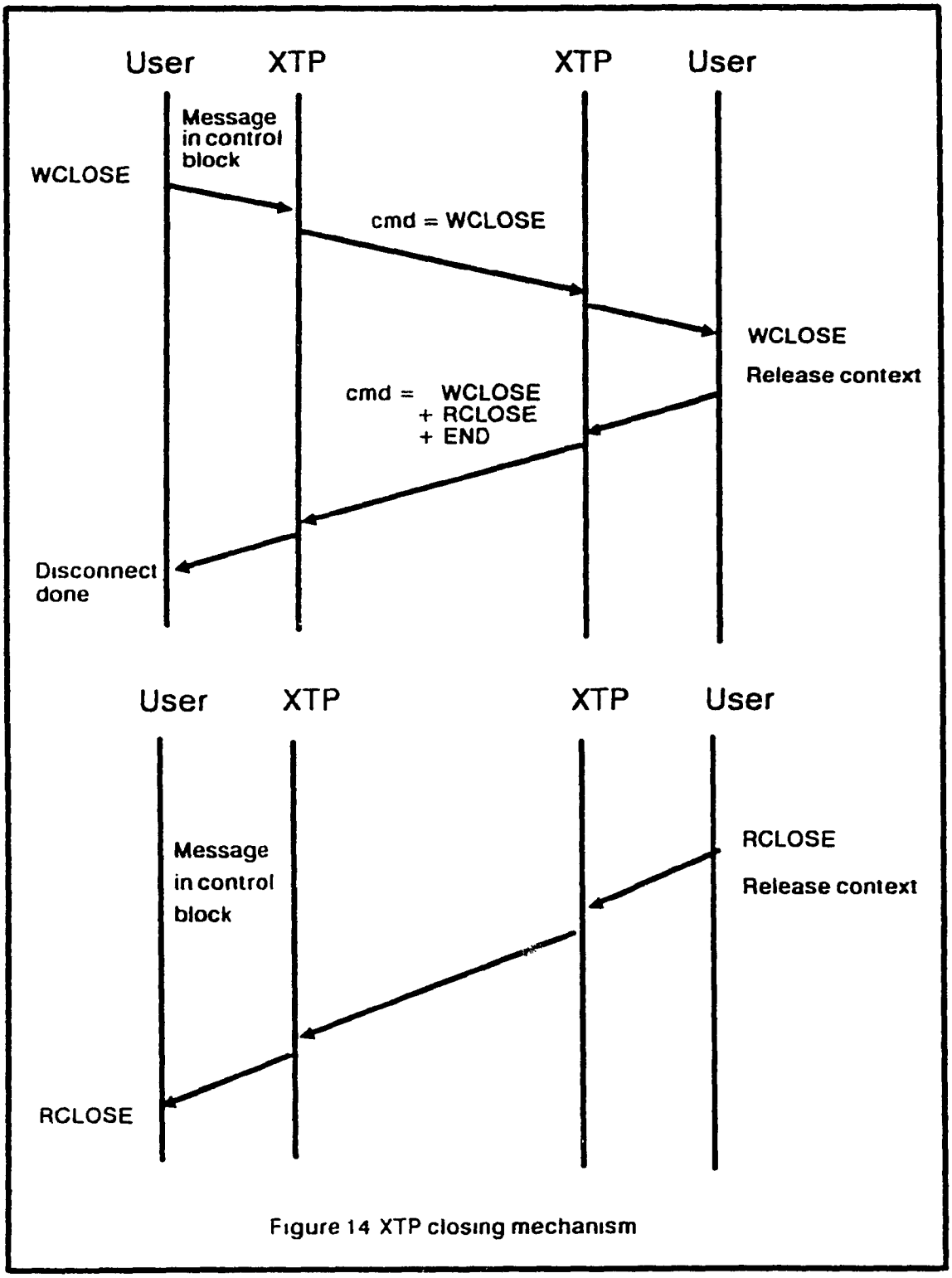


Figure 14 XTP closing mechanism

process.

The advantage of the XTP closing processes is that they avoid the problems associated with VMTP's closing protocol, while at the same time maintaining the simplicity.

3.2.4 XTP timers

The XTP protocol uses four timers, all of which are managed by the sender site. The following is a summary of the description of the four timers [8]:

(1) Wait timer (wtimer), which is started each time a packet is sent with status request bit turned on. If no expected control packet arrives by the time the wtimer expires, XTP will resend the request control packet. In the current implementation, the wtimer interval is thirty-two milliseconds. When a status report control packet arrives, the wtimer which is associated with the status request will be cancelled.

(2) Context life timer (ctimer), which is started when a new virtual circuit context is created. The ctimer expires every sixty seconds. When the ctimer expires, XTP checks its receive packet count. If the counter is zero, the XTP will try up to four times to request status from the receiver site. If all four tries fail, the XTP will inform the user that there is a possible problem with the network or the communication software. If the packet receive counter is greater than zero, XTP will reset the counter to zero and will restart the ctimer again.

(3) Rate control timer (rtimer), which is started after a packet is sent on a specific route. The length of the rtimer is implementation dependent. Before the rtimer expires, no other packet can be sent to the same route.

(4) Credit control timer (crtimer), which is started at the XTP booting time. Its length is 1/60 second. Whenever the crtimer expires, the XTP updates the credit field in every active virtual circuit context, and signals those virtual circuits that are blocked by the credit control to send.

3.2.5 Datagram Service

The datagram service of the XTP protocol is simply a short-lived virtual circuit connect [8], which is similar to the X.25 datagram service (the fast-select). Figure 15 demonstrates an XTP datagram operation. VMTP and AMEOBA TP do not have such a feature (reliable datagram).

3.2.6 Multicast Operation

XTP also provides a multicast service. Its multicast operation is similar to the virtual circuit operation where the virtual circuits are set up with the FIRST packet the sender broadcasts. The error recovery in multicast uses go-back-n. Multicast termination is same as the virtual circuit termination process. Figure 16 shows the multicast operation with error recovery. VMTP does not have any multicasting service, and multicasting in AMOEBA TP is not reliable.

3.2.6.1 No Error Operation

There is a NOERROR operation option that an XTP broadcaster could use in a multicast. This option could be used in a continuous data update environment in which missed or corrupted packets would not have much significance to the receiver. An advantage of the NOERROR operation is that the throughput could be higher, because the sender does not require any retransmission. Once the receivers have received a packet with the NOERROR bit on, they stop reporting errors back to the multicast originator; instead they report the errors to the receiving users.

3.2.6.2 Multicast Reply Packet

An XTP multicast receiver could use a multicast reply packet (MREPLY) to send user data back to the multicast originator without establishing a new virtual circuit. The MREPLY packets are acknowledged by using the resend-pairs in a control packet. XTP only allows one outstanding MREPLY packet per user.

3.2.6.3 Damping Operation on Control Packets

In normal multicast operation, responding control packets are controlled with a damping technique to suppress multicast storms which can created by every node in the network broadcasting the stutas report packet at the same time. Figure 17 demonstrates the damping operation.

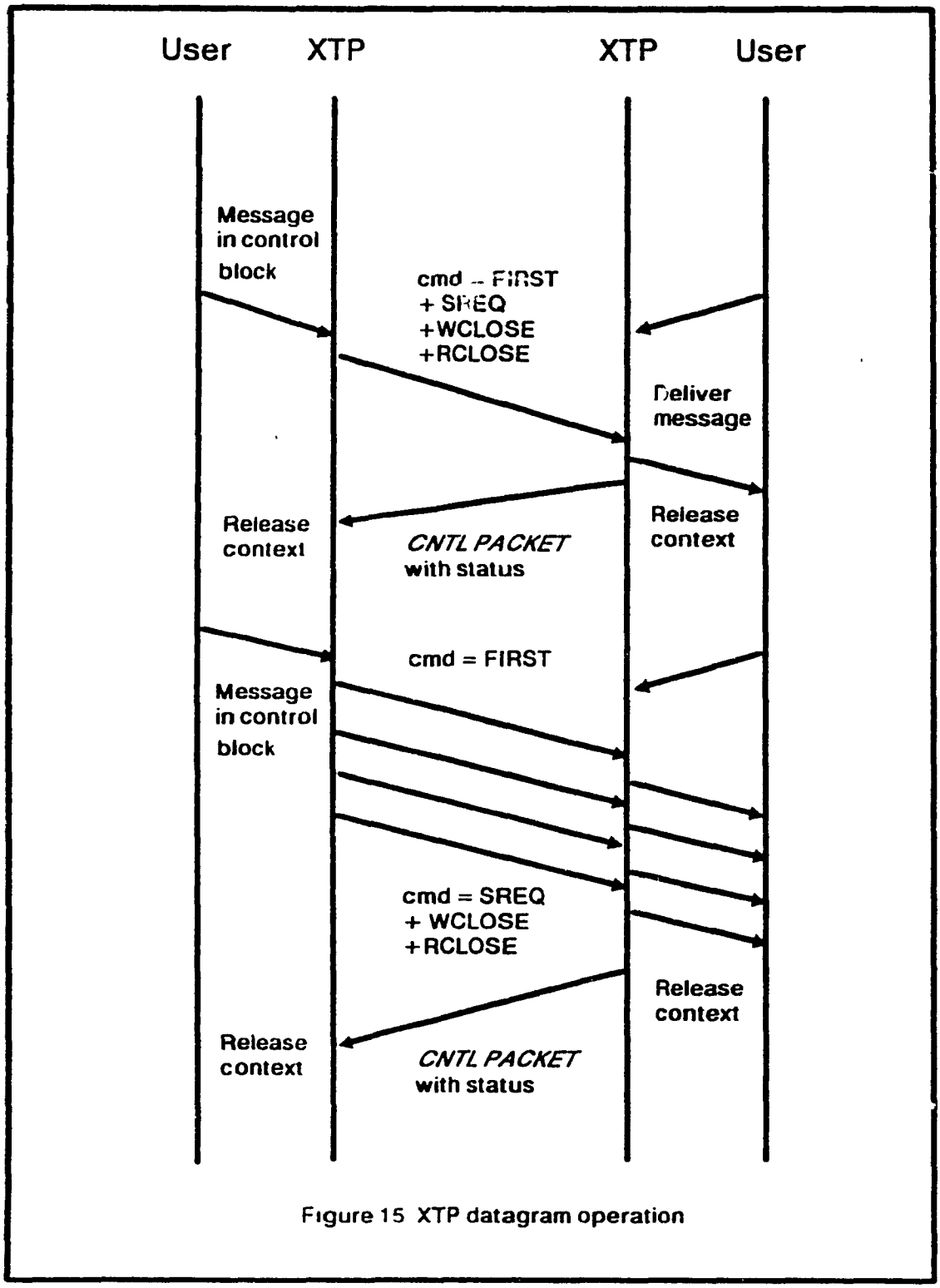


Figure 15 XTP datagram operation

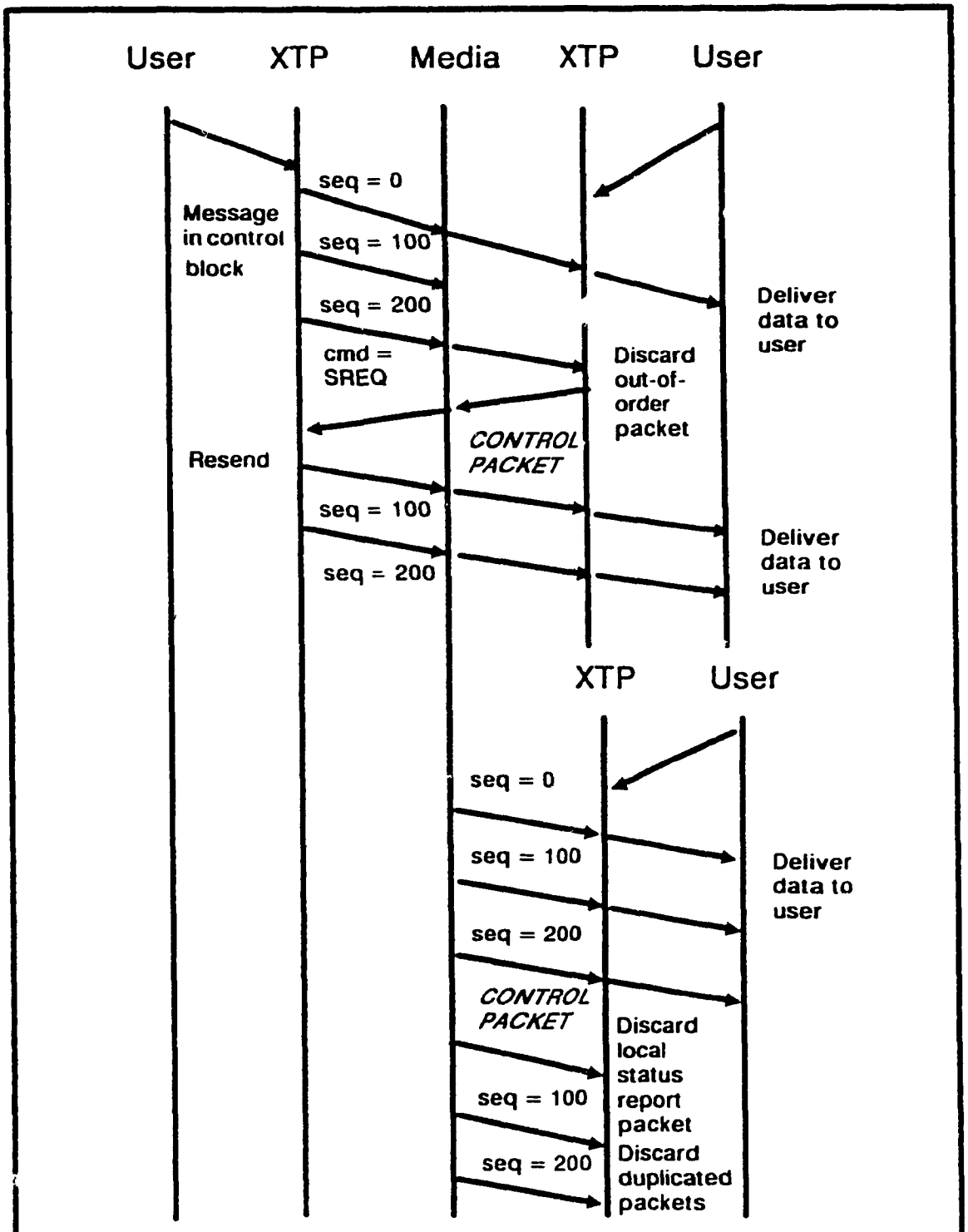


Figure 16 XTP multicast operation

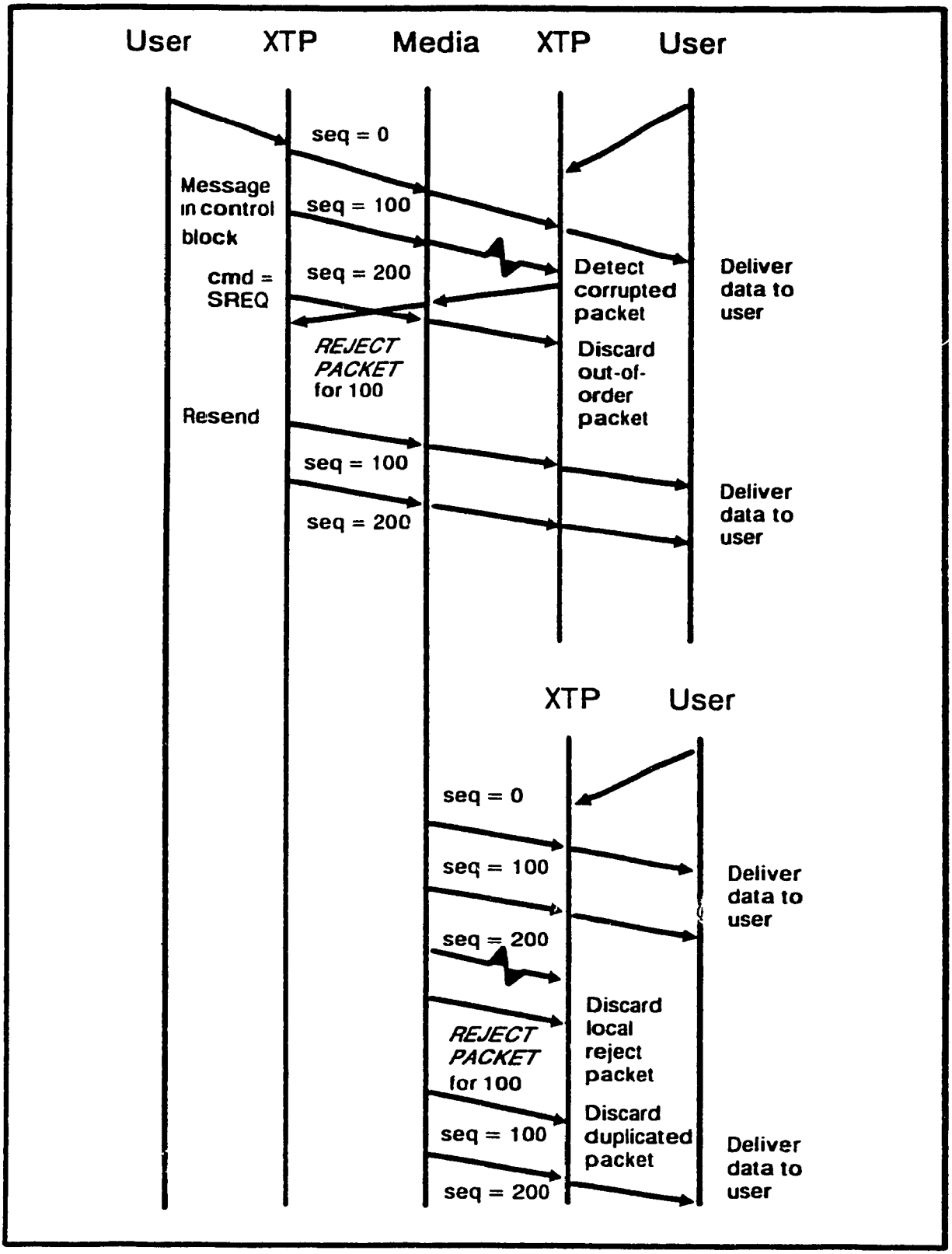


Figure 17 XTP damping operation

CHAPTER 4

AN INTRODUCTION TO LANSF AS A SIMULATION TOOL

"Recent advances in computer and communication systems have resulted in demands for new tools for their analysis. Mathematical modelling techniques have so far proved inadequate in dealing with these systems, and simulation seems to be the only viable alternative."

J. Misra

This chapter introduces the Local Area Network Simulation Facility (LANSF) as the tool which we used to study the performance of XTP. In the following, we summarize the key concepts of LANSF, to help the reader understand our simulation design. We also compare the LANSF design concepts to those of ESTELLE.

4.1 Background

The Local Area Network Simulation Facility (LANSF) is a software simulation modelling package which is being developed by Mr. Pawel Gburzynski and Mr. Piotr Rudnicki at the University of Alberta. This package is used for communication network performance investigations [16].

4.2 The LANSF Simulator Structures

4.2.1 Time

Time in LANSF is discrete, which means that there is an indivisible time unit (ITU), and two moments in real time that differ by less than one ITU are assumed to take place

at exactly the same time. The ITU in LANSF is represented as a 155 bit non-negative integer [16]. Therefore it is possible for LANSF to simulate very small fractions of a second (10 to the power of -12). Besides the ITU unit, LANSF also provides the concept of virtual seconds for better readability of the simulation results. A virtual second is represented by a number of ITUs specified by the individual user. In our case, one virtual second is equal to ten million ITUs, because the ETHERNET transmission rate is ten million bits per second.

The simulation of a global clock is achieved by processing an event in every station, before incrementing one unit of ITU.

4.2.2 Stations

The second concept in LANSF design is stations, which possess links and ports. A station is a representation of a physical machine in a network; a link is a medium which could be used by a number of stations for exchanging messages (packets); and a port is used by a station to connect to a link. In a way, stations are similar to modules, links are similar to channels, and ports are similar to interaction points in ESTELLE. Figure 18 shows some of the possible network configurations that LANSF is able to simulate.

The station attributes in LANSF are defined using a structure in the C language. LANSF allows the user to add additional elements to the station attributes (how to add

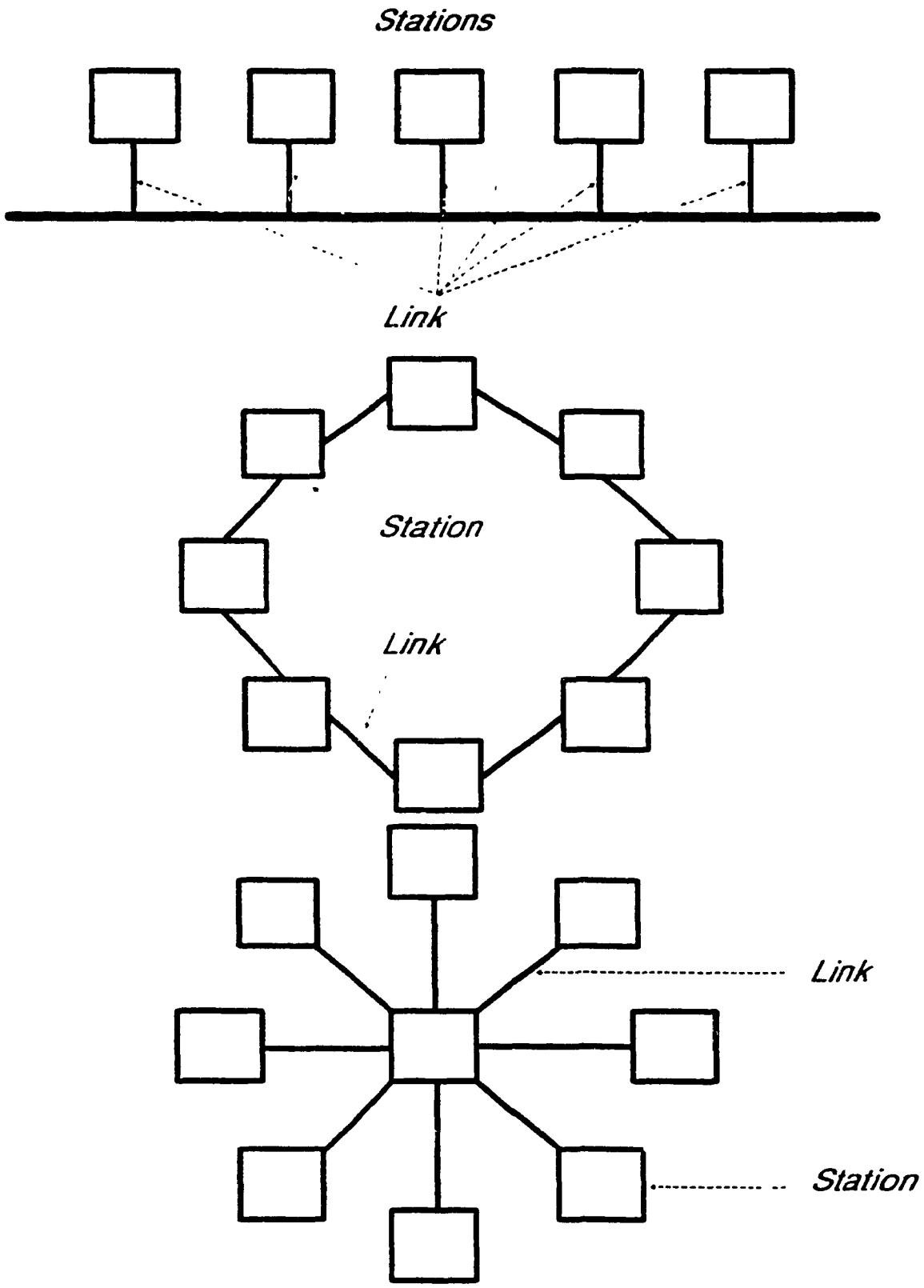


Figure 18 Some simulated network configurations [15]

them is explained in appendix A). Figure 19 is the station structure with the added elements in our simulation. The link and port structures are omitted here because users are not supposed to change them.

4.2.3 Processes

A station may have a number of user-defined processes which are similar to system activities in ESTELLE. The station's operations are driven by events generated by the so-called servers, where each server is a logical process external to the stations, and is responsible for generating events of a particular type [16]. There are four servers existing in LANSF: port, timer, client, and signal.

The port server basically simulates the interaction points that the stations could use to send, to receive, and to detect possible collision.

The timer server essentially is an alarm clock. There are two types of timer servers: one uses ITU's as delay units, and another uses virtual seconds as delay units. The one which uses virtual seconds is more efficient in term of using CPU, because it does not involve the complex calculation of ITUs (155 bits).

The client server generates messages with specific type, length, and inter-arrival time to the stations. Figure 20 is the structure diagram of the LANSF message [16].

The signal server generates a number of types of signals to the station processes.

Each process could be seen as a finite state machine

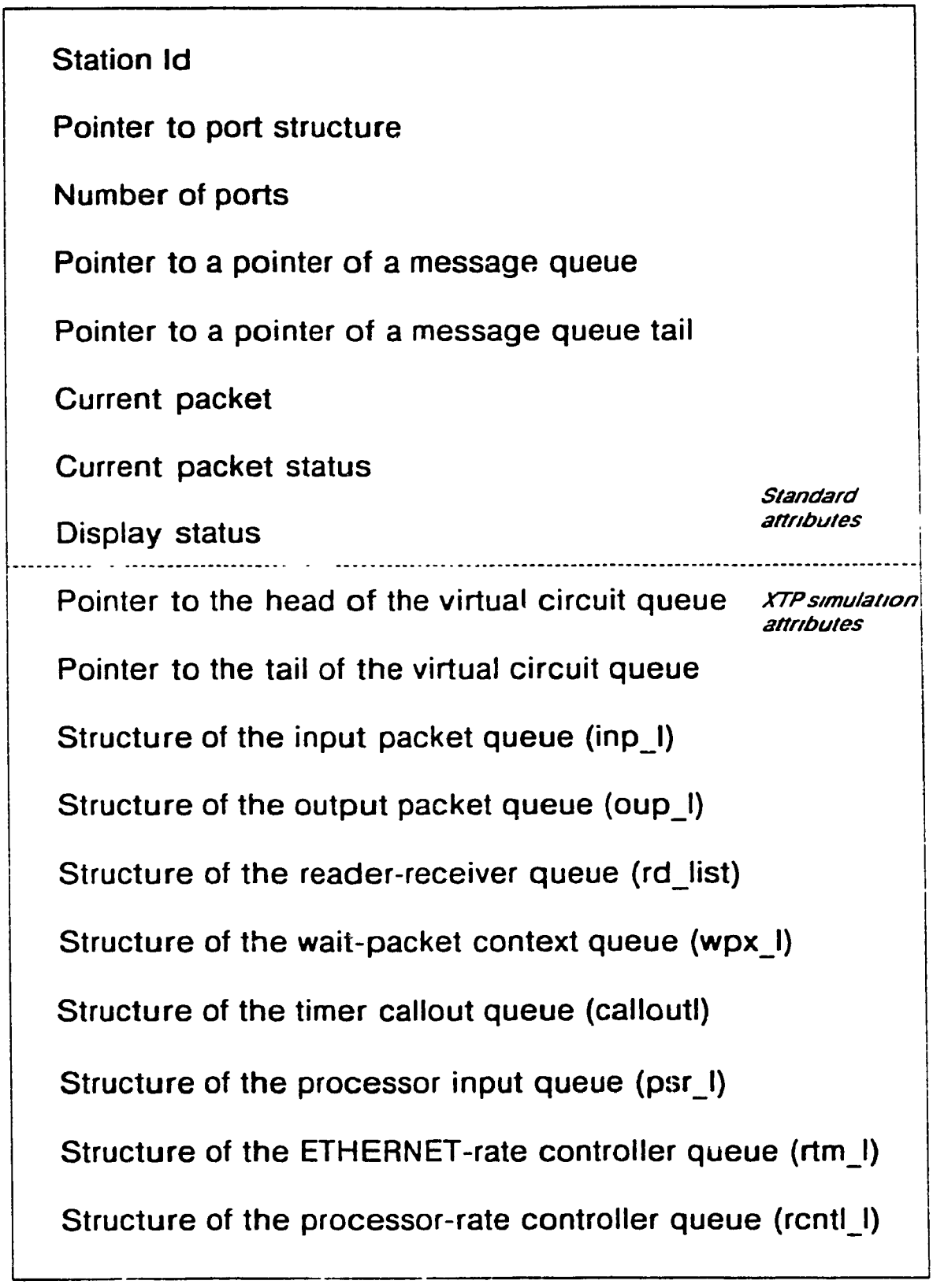


Figure 19 XTP simulation station attributes

(FSM) in which each state is written in the form of a wait function call and each action fired from a state is written in the form of a case statement in the C programming language. A process initially waits for a starting event. For example, the XTP writer process initially waits for the event of message arrival from the simulated client. When an event occurs, the simulator schedules the appropriate processes to run. A process usually returns to a new wait statement after it has handled the previous event.

4.2.4 The Simulated IPC Mechanism

The signal server basically provides a handshake mechanism for inter-processes communication inside a station. However, the signalling mechanism could not be used to pass multiple values, because the signal in LANSF is an integer value. The signal mechanism could also be used between stations.

The main means of communication between stations is to use LANSF packets. There are two types of packets: one type is the standard packet that LANSF uses to calculate link performance statistics; another type is the non-standard packet that is mainly used by the user for control packet purposes. LANSF does not include non-standard packets in its performance statistics. LANSF automatically fragments messages into standard packets. Each standard packet has a contents field which the user could use for passing protocol information such as a sequence number. Figure 21 shows the LANSF packet structure which also includes our protocol

information. In order to create control packets, LANSF originally provided a function make_packet() which only generated a LANSF non-standard packet without the contents field. We rewrote this function so that it generates non-standard packets that have a contents field.

Overall, LANSF provides minimal features for complex protocol performance simulations. As a result, the implementor of complex protocol simulations has to construct the protocol in great detail (almost same as the actual implementation), in order to obtain a proper simulation result.

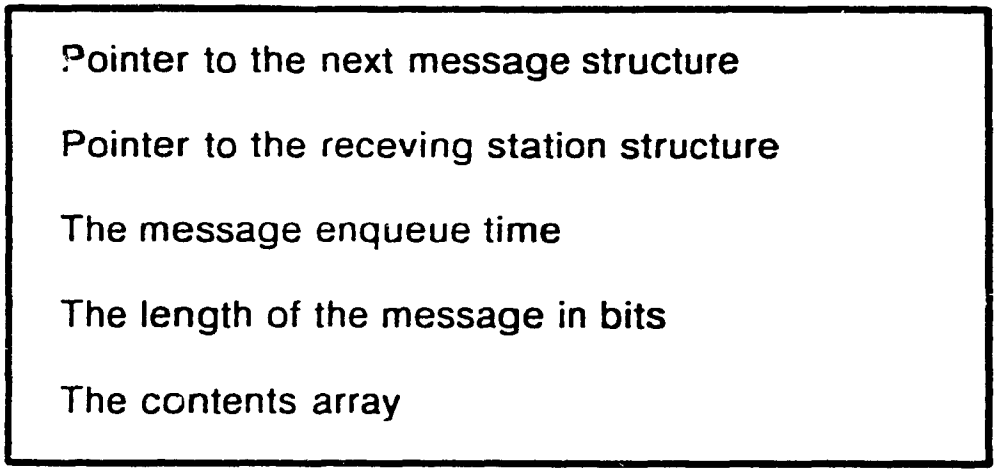


Figure 20 LANSF message structure

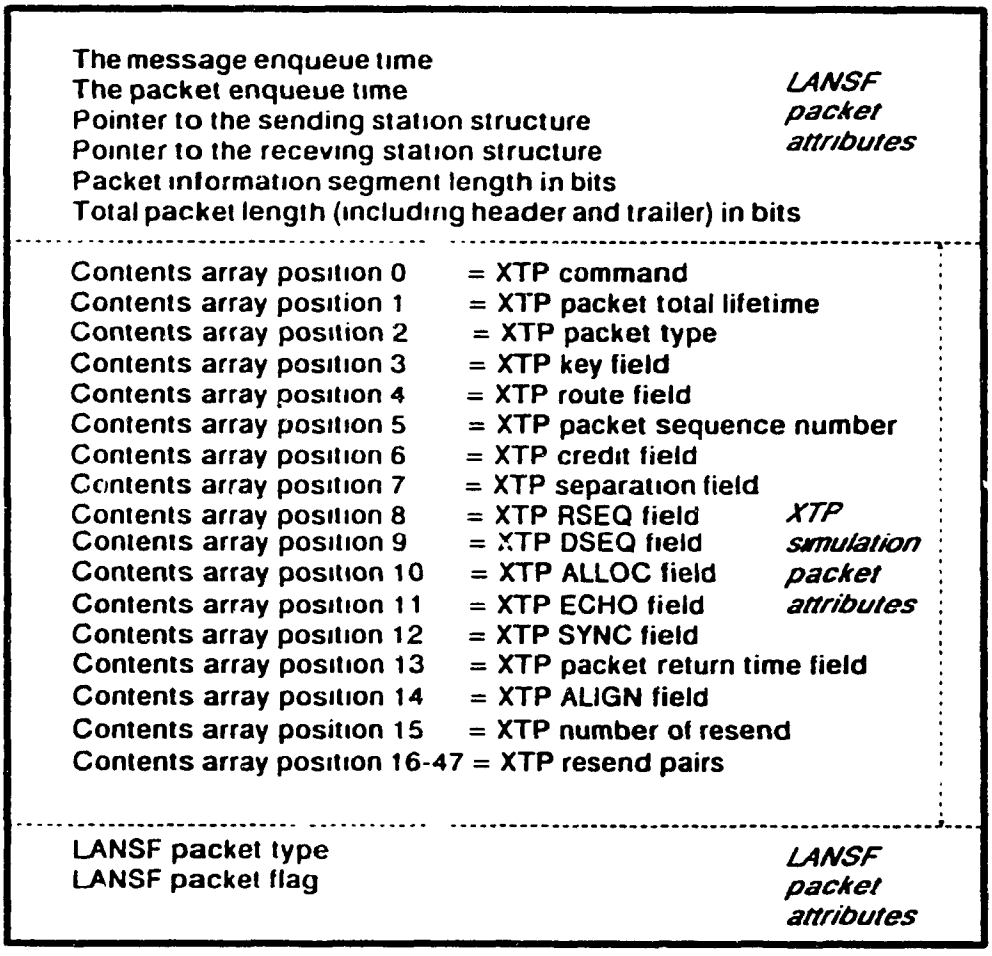


Figure 21 LANSF and XTP simulation packet structure

CHAPTER 5

XTP PERFORMANCE SIMULATION USING LANSF

"More often than we realize, reality conspires to imitate art."

Paul Saffo ACM

Our simulation of XTP uses an ETHERNET environment within which a number of stations set up virtual circuit connections to each other, and transfer data using the XTP protocol. The important design issues of the XTP simulation are the correctness of the protocol implementation and the flexibility of the program (it should be relatively easy to build other protocols from the current implementation). During the design of the simulation program, some formal specification tools were employed: using finite state machines for process construction, and using the LOTOS specification language for the specification of inter-process communication. The following section describes the details of our simulation design.

5.1 The Simulation Assumptions

Figure 22 is the conceptual diagram of the station architecture of our simulation. The architecture is similar to most of the communication board designs. There are four simulated chips: the main processor runs all the XTP processes; the ETHERNET control chip performs all the CSMA/CD operations; the timer chip handles all the XTP timer

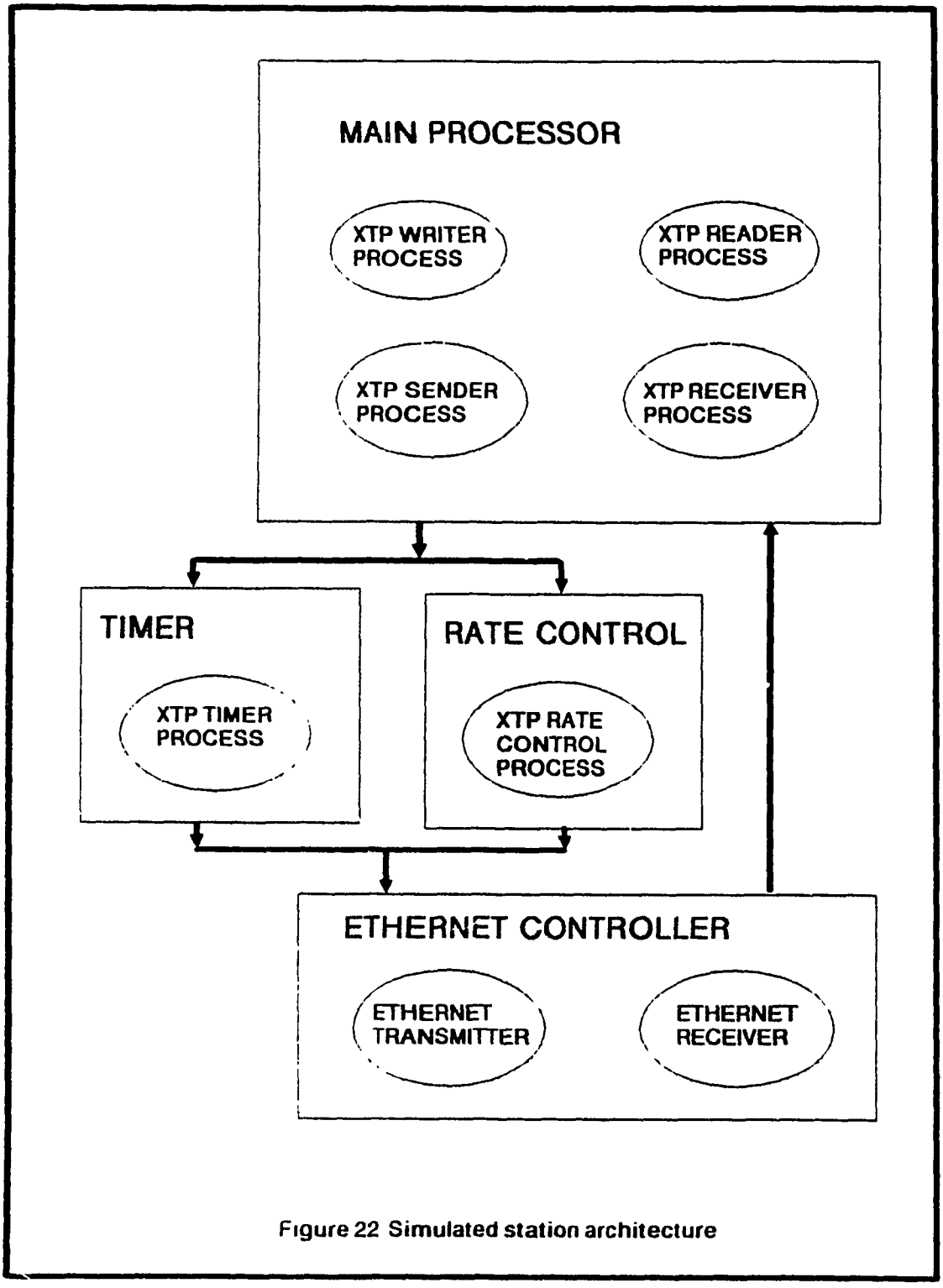


Figure 22 Simulated station architecture

processes; and the rate control chip does all the separation insertion between out-going packets.

There are a number of assumptions in our simulation:

(1) The data link layer service that XTP makes use of is the IEEE 802.2 class I service, which is a connectionless service.

(2) All channels (queues) between simulation processes have unbounded length.

(3) Message fragmentation time is not simulated.

5.2 The Link and Physical Level Simulated Environment

The link layer used in our simulation is the IEEE 802.2 class I which provides a connectionless service to the layer above it. The physical layer is the IEEE 802.3 CSMA/CD.

The link and physical layer simulation routines are adapted from the original ETHERNET simulation example provided by the University of Alberta implementers. We made modifications to the initial wait signal on the ETHERNET transmitter and receiver: the ETHERNET transmitter initially waits for the sending signal from the serializer process or the timer process; the ETHERNET receiver signals the XTP receiver after it has received a packet from the link and has mapped the packet to the input queue. Listings 1 and 2 give the pseudo code for the ETHERNET transmitter and receiver processes respectively.

Listing 1 Pseudo code for the ethernet_sender process

```
PROCESS ethernet_sender

    Created by:
       LANSF simulator;

    Input signals:
       ETHER_SEND from the serializer process or the
                  timer process;

       JAM or COLLISION from the port server process;

    Output signals:
       ETHER_DONE to rate_controller process;

BEGIN

    WAIT for ETHER_SEND signal from the serializer process
         or from the timer process;

    DEQUEUE an event item from the oup_q queue;

    LISTEN to port;

    IF there is no activity THEN
       SEND current packet;
    ELSE
       CONTINUE to listen;
    ENDIF;

    WAIT for collision signal;

    WAIT for transmission completion;

    IF collision heard THEN
       ABORT sending;
       SEND jam packet;
       BACK UP;
       CONTINUE to listen again and do retransmission;
    ENDIF;

    ENQUEUE an event item to rtm_1 queue;

    SIGNAL rate_controller process that the packet
           has been sent;

    RETURN to the beginning of the process;

END PROCESS ethernet_sender
```

Listing 2 Pseudo code for the ethernet_receiver process

```
PROCESS ethernet_receiver

   Created by:
      LANSF simulator;

   Input signals:
      MY_PACKET from the port server process;


   Output signals:
      ETHER_ARR to the xtp_receiver process;

BEGIN

   WAIT for MY_PACKET signal from port server process;
   GET the packet;
   MAP the packet to the inp_q queue;
   SIGNAL xtp_receiver process with ETHER_ARR;
   RETURN to the beginning of the process;

END PROCESS ethernet_receiver
```

5.3 The Design of the XTP Simulation

In our simulation, there are nine processes in an XTP entity: the initializion process, the xtp_writer process, the xtp_reader process, the xtp_sender process, the xtp_receiver process, the timer process, the rate_control process, the credit_control process, and the serializer process. The initialization process is responsible for setting up the virtual circuit simulation data structures. The xtp_writer and xtp_reader processes serve as transport service accessing points (TSAP). The xtp_sender and xtp_receiver processes are like other protocol implementations: they perform the core operations of the protocol. In this case, it is XTP. The serializer process simulates the main processor which runs the xtp_reader, xtp_sender and xtp_receiver processes. The timer process

does timer interrupt operations for ctimer, wtimer, and rtimer; and the credit_timer process is assumed to be run on a different chip. The rate_control process is also assumed to be run on a different chip.

All the initialization processes will be terminated automatically by themselves after they have completed the initialization. The other processes will be terminated by the simulator when one of the exit conditions specified in the test data file is met.

Appendix B shows the function call structure of the XTP simulation program. Figure 23 is the data flow diagram for the simulation program.

5.3.1 The Supporting Data Structures and Routines

Figure 24 illustrates the data structures used in the XTP performance simulation, all adapted from the protocol definition [3]. However, some fields are not used in the simulation, because of unnecessary overheads. For example, the user data buffer in XTP packets is not used because in the simulation of the data transmission, LANSF just delays for a time equal to the transmission time of the packet, rather than actually transferring data. As a result, no user data buffer is required. Although most of the IPC could be done by using the signalling mechanism provided by the LANSF package, parameter passing between processes is not possible. Local variables in processes are not preserved during events. Eight queues are implemented to accommodate

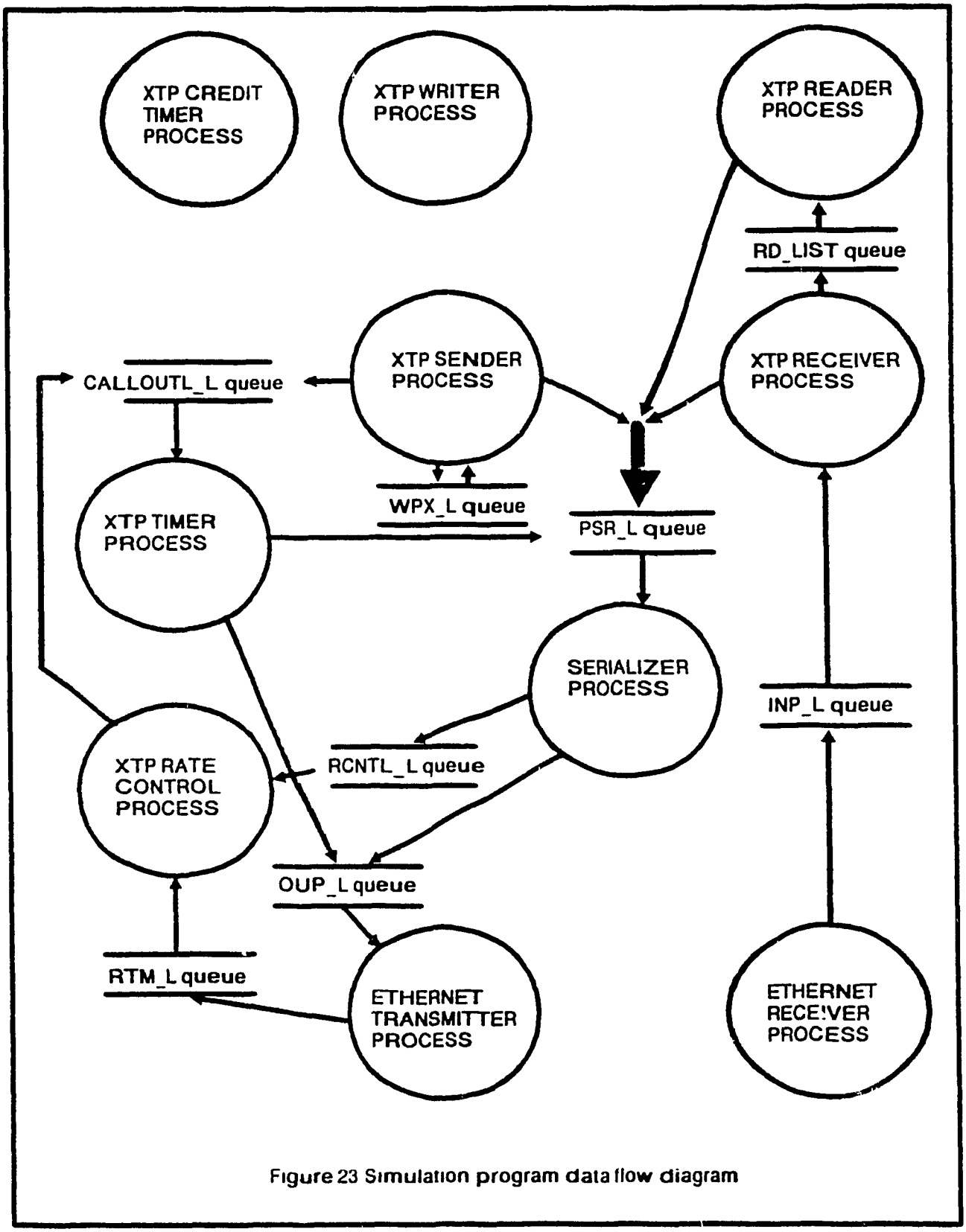


Figure 23 Simulation program data flow diagram

XTP simulation control packet structure

XTP command field
XTP packet total life time
XTP packet type
XTP key field
XTP route field
XTP packet sequence number
XTP credit field
XTP separation field
XTP RSEQ field
XTP DSEQ field
XTP ALLOC field
XTP ECHO field
XTP SYNC field
XTP packet return time field
XTP ALIGN field
XTP number of resend pairs
XTP resend pairs

XTP simulation context record structure

Pointer to next context record
Peer station id
Context state
Sending control packet structure
Receiving control packet structure
Next output sequence number
Current output sequence number
Final output sequence number
Maximum I/O queue size
Current output queue size
Current input queue size
Received packet count
WTIMER interval length
RTIMER interval length
CTIMER interval length
Output round robin limit
Estimated round trip time
Context wait packet flag
Previous LANSF packet flag
Message first packet flag
Message start time
Message end time
Accumulated message delay time
Waiting packet queue structure

Figure 24 LANSF and XTP simulation data structures

the parameter passing among processes and to preserve local variables. All the queues are built in a generic form (Figure 25).

The purposes of the eight queues are as follows:

inp_l : links ethernet_receiver and xtp_receiver processes; used for passing received packets.

oup_l : links the serializer and timer processes to the ethernet_sender process; used for passing output packets.

rd_list : links the xtp_receiver process to the xtp_reader process; used to pass data packets.

wpx_l : used for preserving a pointer to a context which has packets waiting to be sent.

callout1_l : used by the timer process.

psr_l : links the xtp_reader, the xtp_sender, and the xtp_receiver processes to the serializer process; used for passing wake-up signals and delay length on data copy and packet checksum.

rtm_l : links the ethernet_sender process to the rate_controller process; used for passing the send complete time and destination identifier.

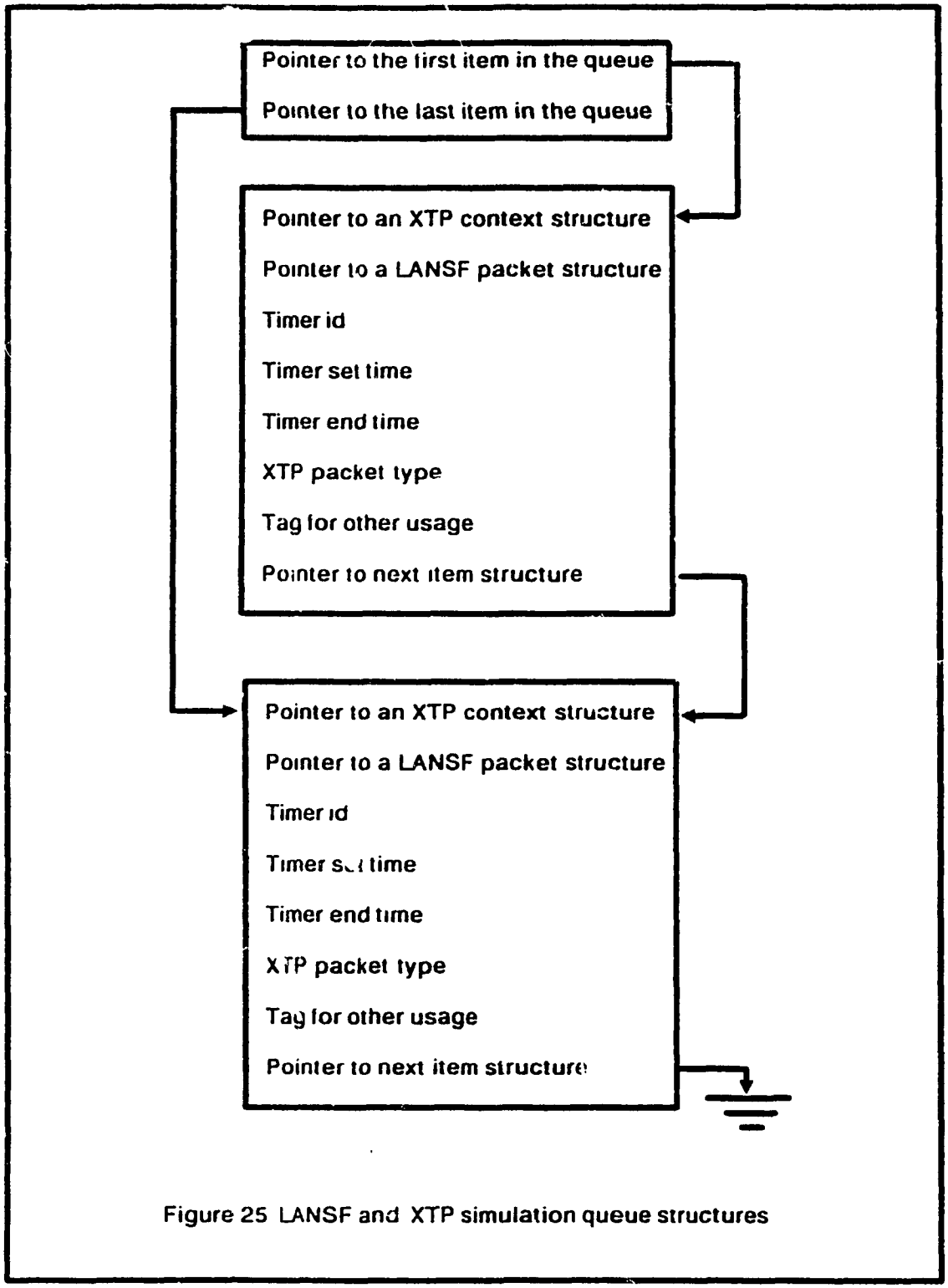


Figure 25 LANSF and XTP simulation queue structures

rcntl_1 : links the serializer process to the rate control process; used for passing output packets.

The routines that operate on these queues are also designed in such a way that they are generic. To append an event item to a queue, the user has to use the function append_event which takes the following parameters: pointer to the queue, pointer to the context, pointer to the packet, tag value, delay length, and the packet type. To remove an item from the head of a queue, the user has to use the function get_e_item which returns a pointer to character. It is the user's responsibility to cast the pointer to the desired form. The function get_e_item takes two parameters: command for the item to be returned and the pointer to the queue. The possible commands are as follows: GET_CONTEXT returns a pointer to a context, GET_PACKET returns a pointer to a packet, and GET_E_ITEM returns a pointer to an event item.

In addition to the above eight queues, each station also has a context list which is used to link all the XTP virtual circuit contexts. The routine operating on the context list is called get_context. It takes the source identifier from a received packet, and returns a pointer to the context associated with the packet.

Other than the eight queues used for carrying information between processes, signals are also used to wake up sleeping processes. Figure 26 illustrates the signals

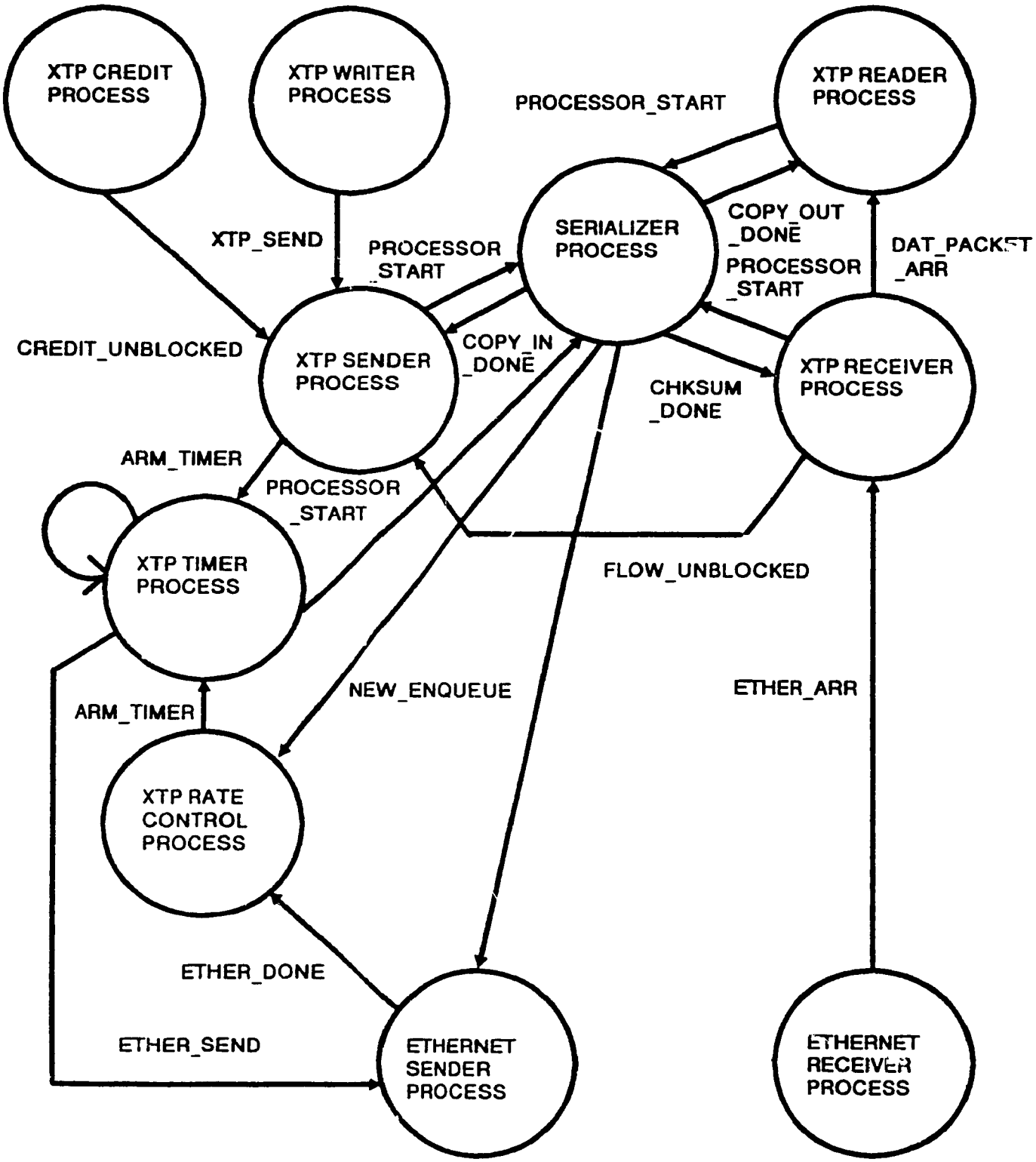


Figure 26 Simulation program signal flow diagram

among the simulated process. The following signals are used in our simulation.

PROCESSOR_START: used by the xtp_reader, the xtp_sender and the xtp_receiver processes to wake up the serializer process. This signal indicates to the serializer process that there is a new request for using the processor.

XTP_SEND: used by the xtp_writer process to indicate to the xtp_sender process that a new message has been received from the LANSF standard client process.

COPY_IN_DONE: used by the serializer process to wake up the xtp_sender process. It signals that the process of copying data from user space to xtp space has been completed.

COPY_OUT_DONE: used by the serializer process to wake up the xtp_reader process. This means that the process of copying data from the xtp space to user space has been completed.

CHKSUM_DONE: used by the serializer process to wake up the xtp_receiver process when the checksum delay has been completed.

ARM_TIMER: used by the xtp_sender, rate_controller and timer processes to signal the timer process to start the countdown of the first item on the callout list.

DAT_PACKET_ARR: used by the xtp_receiver process to signal the xtp_reader process that a new data packet has been enqueued to the rd_list.

NEW_ENQUEUE: used by the serializer process to signal

the rate_controller process that a new and unblocked packet has been enqueued to the rcntl_1.

ETHER_DONE: used by the ethernet_sender process to signal the rate_controller process that a packet has been successfully sent and timing information has been enqueued to the rtm_1.

ETHER_SEND: used by the timer process and the serializer process to wake up the ethernet_sender process. This signal indicates to the ethernet_sender process that there is a new out-going packet being enqueued to the oup_1.

ETHER_ARR: used by the ethernet_receiver process to signal the xtp_receiver process that a packet has been received and enqueued to inp_1.

5.3.2 Initialization Process

The initialization process sets up the virtual circuit simulation environment. This process will terminate after the initialization has completed and the process entry in the simulation will be erased. As a result, the initialization does not occupy any process time. Figure 27 is the finite state machine of the initialization process. The process starts from the initial state and terminates at the done state after intialization is completed. Listing 3 contains the pseudo-code for the intialization process.

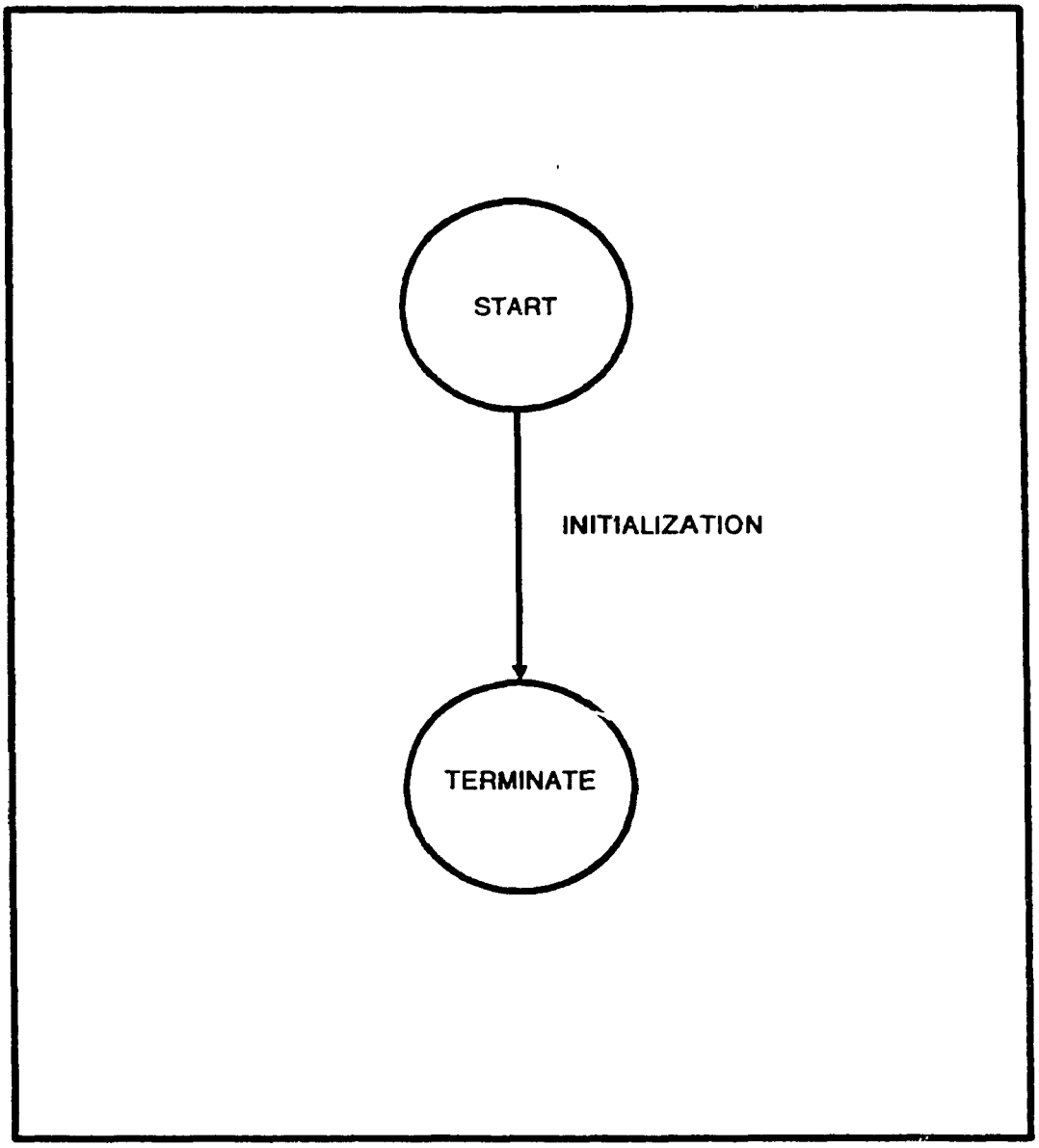


Figure 27 Finite state machine for the XTP initialization process

Listing 3 Pseudo code for the initialization process

```
PROCESS initialization

   Created by: LANSF simulator;

   Input signals: None;

   Output signals: None;

BEGIN

   WHILE  the number of contexts created is less than the
          total number of stations minus one  DO

      ALLOCATE memory for a new context;

      INITIALIZE all default tunable parameter values to the
                 new context;
   END WHILE;

   TERMINATE initialization process;

END PROCESS initialization
```

5.3.3 Writer Process

The xtp_writer process is intended to provide an interface for a user to create active open virtual circuits and then send data [8]. However, the writer process in our simulation merely serves as a triggering mechanism to start the chain reaction of the entire simulation.

The simulated writer has the following functions: (a) to wait for a message arrival signal which is generated by the LANSF scheduler process, (b) to signal the xtp_sender process to send the newly arrived message, (c) to return to (a). Figure 28 is the finite state machine of the xtp_writer process. Listing 4 contains the pseudo-code for the xtp_writer process.

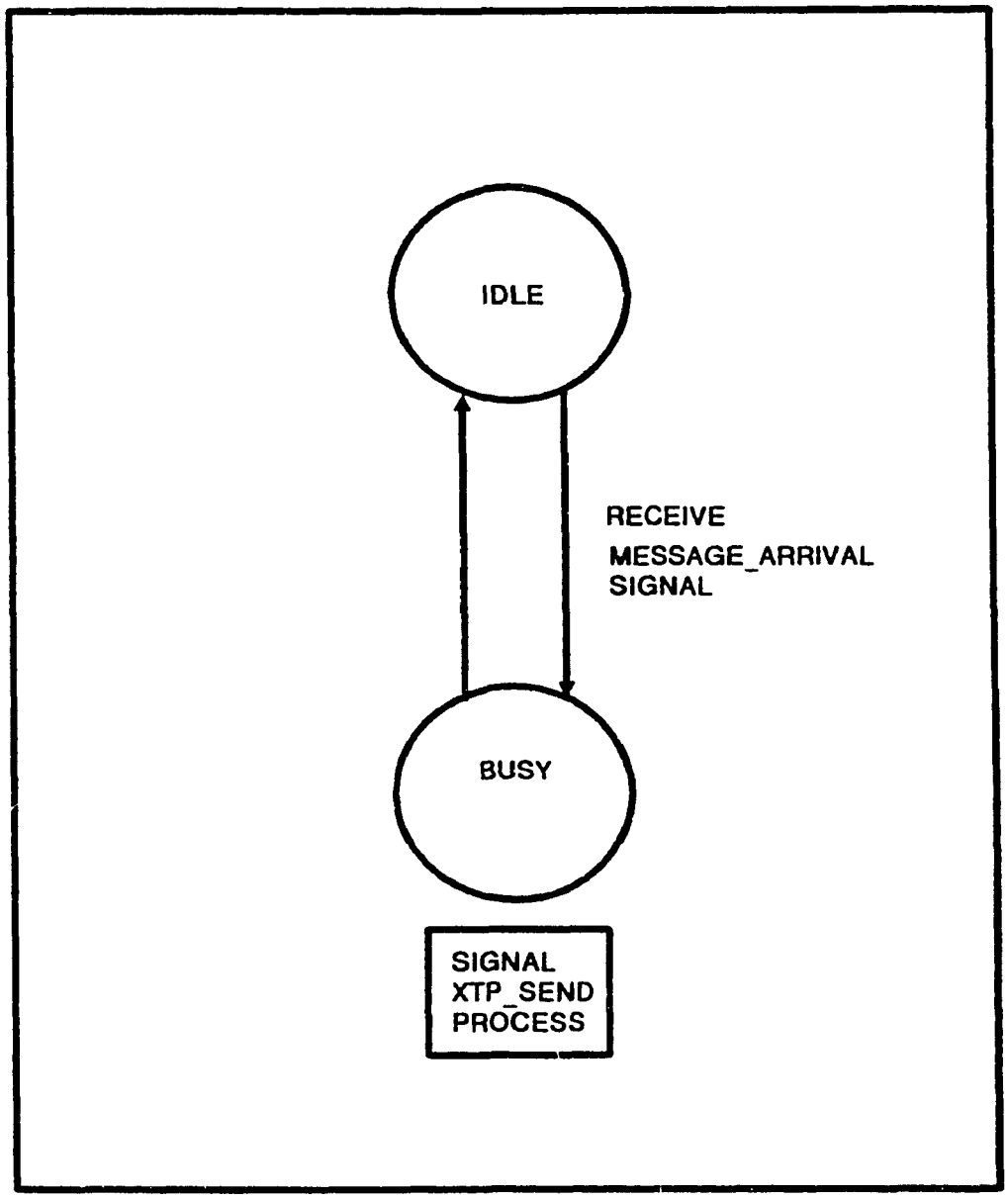


Figure 28 Finite state machine for the xtp_writer process

Listing 4 Pseudo code for the xtp_writer process

```
PROCESS xtp_writer

   Created by:
      LANSF simulator;

   Input signals:
      MESSAGE_ARRIVAL from standard client process;

   Output signals:
      XTP_SEND to xtp_sender process;

BEGIN
   WAIT for MESSAGE_ARRIVAL signal from the standard client;
   SIGNAL the xtp_sender process to send the message;
   RETURN to the beginning;
END PROCESS xtp_writer
```

5.3.4 Reader Process

The actual purpose of the xtp_reader process is to provide an interface for an XTP user to create a passive open on an XTP virtual circuit and to perform the management of the out-of-order packet recording [8]. The xtp_reader process in our simulation does not create passive open simply because there is no user to actually receive the incoming data. The passive open contexts are created by the xtp initializing process described above.

The functions of the simulated xtp_reader process are as follows: (a) to receive incoming data packets that are within the range of the received sequence number and the allocation size, (b) to record out-of-order packets and generate a reject packet to the sending site, (c) to simulate the data copy-out delay. Figure 29 is the finite state machine for the xtp reader process. Listing 5 contains the pseudo-code for the xtp_reader process.

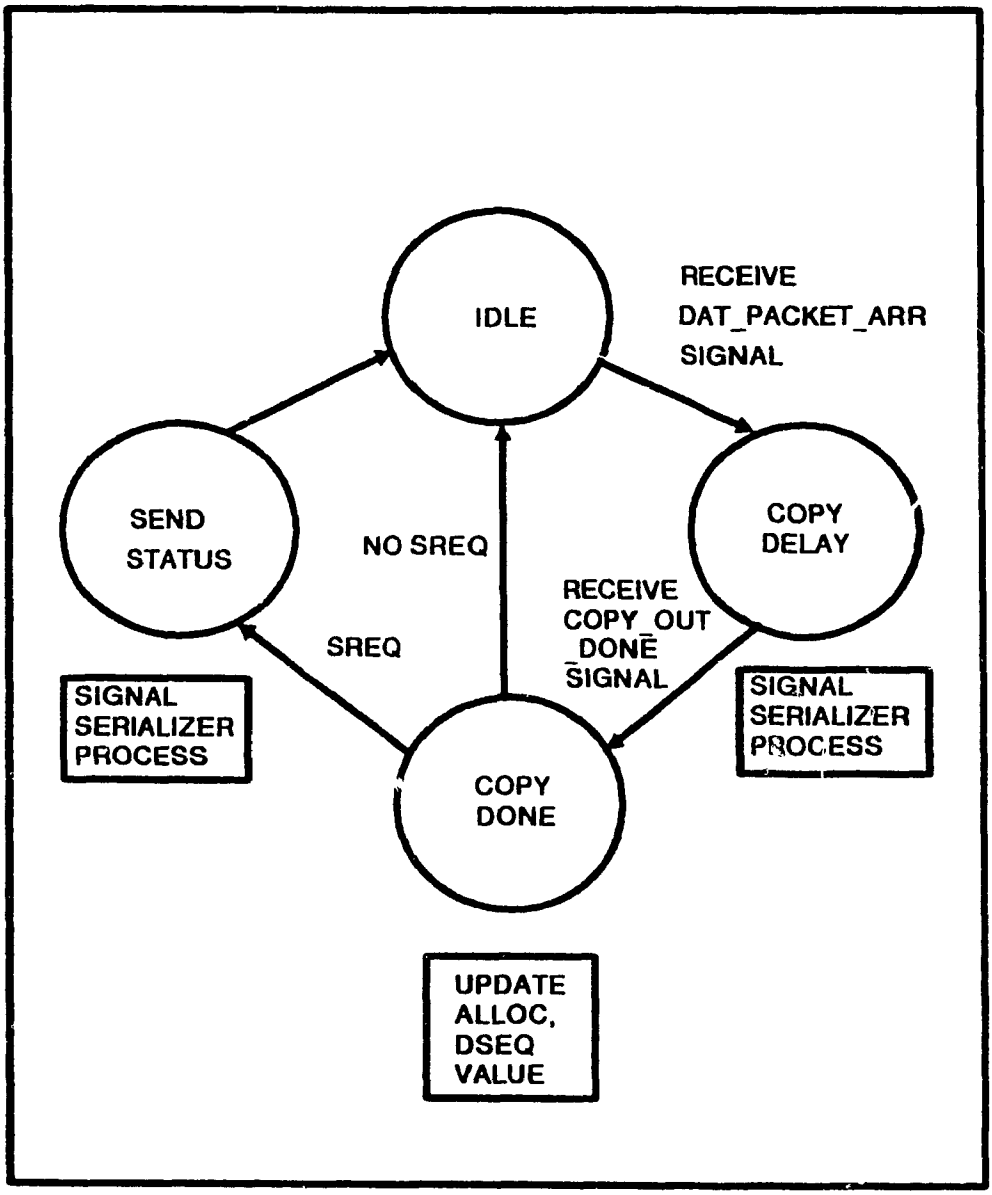


Figure 29 Finite state machine for the xtp_reader process

Listing 5 Pseudo code for the xtp_reader process

```
PROCESS xtp_reader

   Created by:
      LANSF simulator;

   Input signals:
      DAT_PACKET_ARR from the xtp_receiver process;
      COPY_OUT_DONE from the serializer process;

   Output signals:
      PROCESSOR_START to serializer process;

BEGIN

   WAIT for DAT_PACKET_ARR signal from the xtp_receiver
        process;

   IF  rd_list queue is empty THEN
      RETURN to the beginning;
   ENDIF

   IF the packet sequence number is greater than the
      expected sequence number THEN
      RECORD the sequence number in a resend pair;

      IF there is no outstanding reject packet THEN
         SEND a reject packet to the sender;
      ENDIF

      RETURN to the beginning;
   ENDIF;

   EXTEND allocation size;
   REDUCE input queue size;

   IF enqueue an event item to psr_1 queue returns the
      value of head THEN
      SIGNAL the serializer to start;
   ENDIF;

   WAIT for COPY_OUT_DONE signal from the serializer
        process;
   DEQUEUE an event item from the rd_list queue;

   IF the packet command flag has status request bit set
      THEN
      SEND status control packet to sender site;
   ENDIF;

   RETURN to the beginning;

END PROCESS xtp_reader
```

5.3.5 Sender Process

The xtp_sender process performs the core operation of the protocol including flow control and credit control. The reason that the rate control mechanism is not included as a part of the sender process is to preserve parallelism in the sending process. The xtp_sender process is designed in such a way that the entire process is a non-blocked machine because the sender process should have the ability to perform multiplexing among the virtual circuits. Packet fragmentation is performed by LANSF itself, and the delay caused by the fragmentation is not simulated.

The simulated xtp_sender has the following functions: (a) to fragment the message into packets, (b) to enforce end-to-end flow control and local credit control, and (c) to preserve incoming packets that are blocked by either flow control or credit control. Figure 30 is the finite state machine for the XTP sender process. Listing 6 contains the pseudo-code for the xtp_sender process.

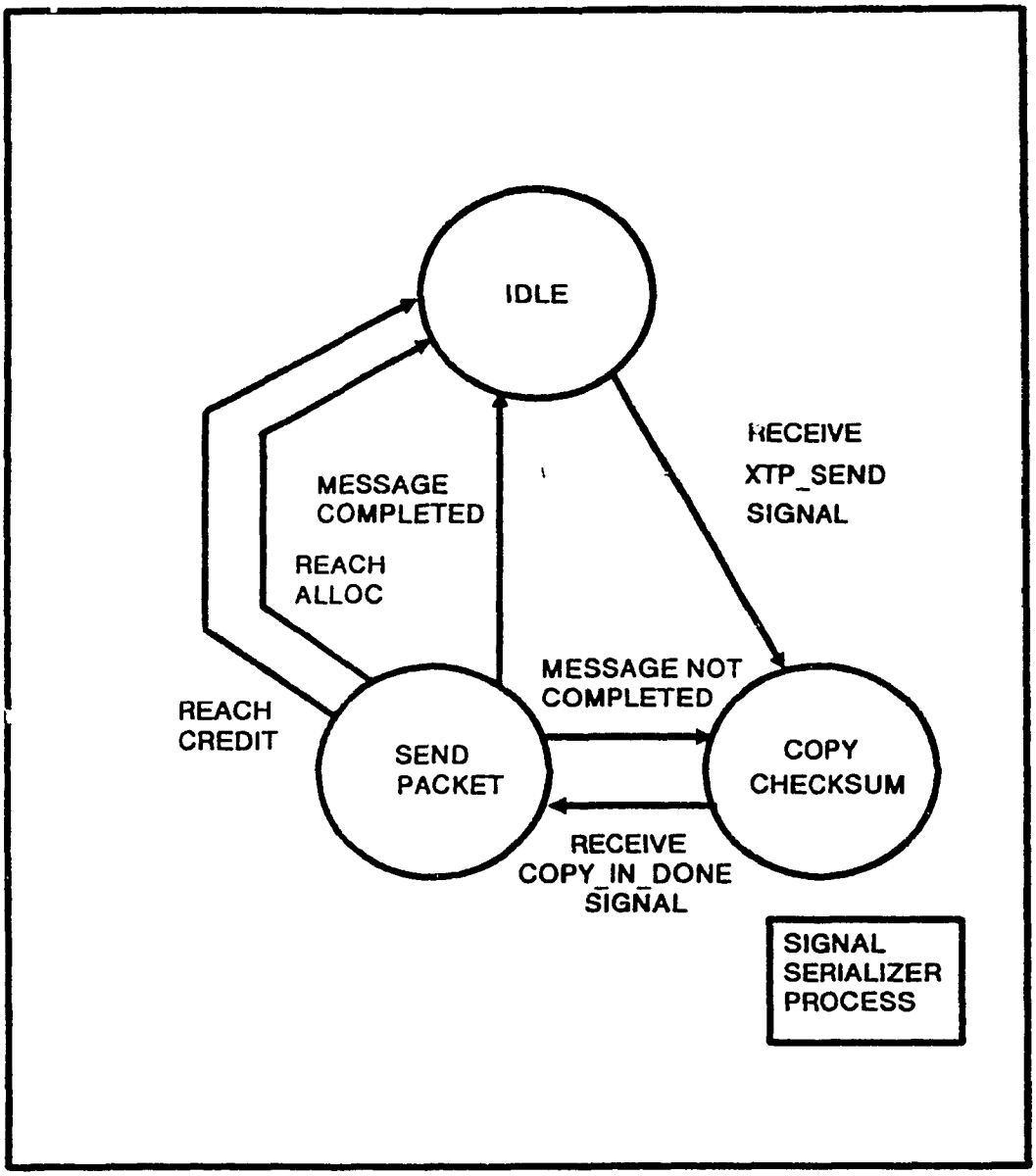


Figure 30 Finite state machine for the xtp_sender process

Listing 6 Pseudo code for the xtp_sender process

```
PROCESS xtp_sender

   CREATED by:
      LANSF simulation;

   Input signals:
      XTP_SEND from the xtp_writer process;
      COPY_IN_DONE from the serializer process;

   Output signal:
      PROCESSOR_START to the serializer process;

BEGIN

   WAIT for XTP_SEND signal from the xtp_writer process;

   IF the current packet in the station is full THEN
      RELEASE the packet;
   ENDIF;

   IF function get_next_packet returns a null pointer THEN
      IF function find_wait_x returns a context pointer THEN
         CONTINUE at label is_wait;
      ELSE
         RETURN to the beginning;
      ENDIF;
   ENDIF;

   ENQUEUE an event item to the per_1 queue;

   SIGNAL the serializer process to start the copy-in
          delay;

   WAIT for COPY_IN_DONE signal from serializer process;

   IF fail to find the context pointer which is associated
      with the current packet THEN
      RELEASE the current packet;
      RETURN to the beginning;
   ENDIF;

   IF the current context is blocked by flow or credit
      control THEN
      SAVE the current packet;
      RETURN to the beginning to get the next packet;
             (from another message)
   ENDIF;

   IF the context already has packet waiting to be sent
      THEN
LABEL is_wait
      SET context state to BUSY;
```

```
      IF there are still packets after sending up to the
         maximum allocation or maximum credit allowance
         THEN
         SAVE the current packet;
         RETURN to the beginning to get the next packet;
         (from another message)
      ENDIF;

   ENDIF;
   IF the current packet is full THEN
      SEND the current packet to the serializer process;
   ENDIF;

END PROCESS xtp_send


FUNCTION send_packet

   Input Parameters:
      pointer to the sending context;
      pointer to the packet;

   Output:
      zero        if the last packet is not up to allocation
                  limit or credit limit;

      nonzero     if the context is blocked by either
                  allocation or credit;

   Side effects: none;

BEGIN

   COPY the current packet to a new packet;

   IF the current packet is the FIRST packet THEN
      SET the packet command field to indicate FIRST packet;
      ARM ctimer;
   ENDIF;

   IF the output sequence is EQUAL to or GREATER than the
      receiver's buffer allocation THEN
      SET the command field inside the packet to
          indicate status request;
      ARM wtimer;
      SET return value to be nonzero;
   ENDIF;

   IF the packet is the last one in the message THEN
      SET packet command to indicate end of message;
   ENDIF;

   ASSIGN current sequence number to the packet;
```

```
    EXTEND the next sequence number;
    ENQUEUE the packet to the psr_1 queue;
END FUNCTION send_packet
```

5.3.6 Receiver Process

The xtp_receiver process performs the core xtp packet reception operations. It is responsible for discarding packets for which the sequence number is greater than the buffer allocation or less than the delivered sequence number. It is also the simulated xtp_receiver's responsibility to perform the necessary retransmission. The error recovery mechanism is implemented, but it is not used in the current simulation. The reason that the retransmission is not done by the xtp_sender process is that there is no actual user data in the simulation and the xtp_receiver can act for the xtp_sender, simplifying the simulator.

The simulated xtp_receiver process has the following functions: (a) to wait for the packet reception signal from the ETHERNET receiver, (b) to delay a period corresponding the calculation of a checksum, (c) to check the packet sequence number against alloc and dseq values, (d) to forward the packet to the xtp_reader process, (e) to return to (a). Figure 31 illustrates the finite state machine of the xtp_receiver process. Listing 7 contains the pseudo-code for the xtp_receiver process.

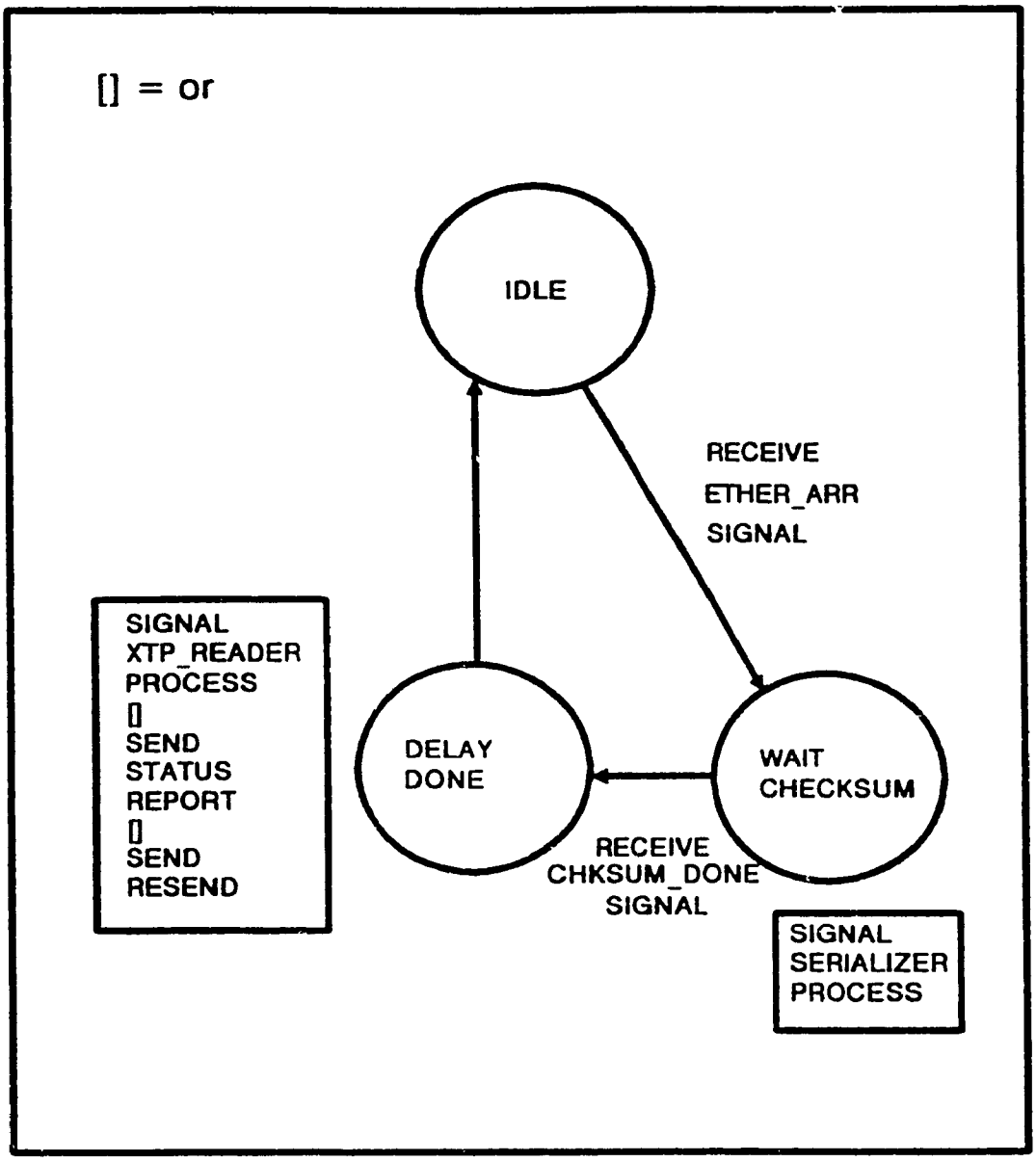


Figure 31 Finite state machine for the xtp_receiver process

Listing 7 Pseudo code for the xtp_receiver process

```
PROCESS xtp_receiver

   Created by
      LANSF simulator;

   Input signals:
      ETHER_ARR from ethernet_receiver process;
      CHKSUM_DONE from the serializer process;

   Output signals:
      DAT_PACKET_ARR to xtp_reader process;

BEGIN

   WAIT for ETHER_ARR signal from the ethernet_receiver
        process;

   IF the inp_1 queue is empty THEN
      RETURN to the beginning;
   ENDIF;

   ENQUEUE an event item which contains the checksum delay
           length to psr_1 queue;

   IF the psr_1 queue was empty THEN
      SEND PROCESSOR_START signal to the serializer process;
   ENDIF;

   WAIT for the CHKSUM_DONE signal from the serializer
        process;

   DEQUEUE an event item from the inp_1 queue;

   GET the context associated with the packet;

   IF the packet is a data packet THEN
      IF the packet sequence number is greater than the
         allocation or less than the delivered sequence
         number  THEN
         DISCARD the packet;
         RETURN to the beginning;
      ENDIF;

      IF the packet sequence is equal to the expected
         sequence number THEN
         EXTEND the expected sequence number by the
                length of the received packet;
      ENDIF;

      IF the virtual circuit was  expecting a retransmission
         THEN
         UPDATE the resend pair in the context;
```

```
      ENDIF;

      ENQUEUE the packet to the rd_list queue;

      SIGNAL the xtp_reader process that a new data packet
           has arrived;
   ENDIF;

   IF the packet is a control packet THEN
      IF the packet command field does not have the status
         request bit turned on THEN
         COPY the status to the context;
         IF the number of resend is greater than zero THEN
             DO retransmission;
         ENDIF;
      ELSE
         SEND a control packet which contains the current
              receiver status to the sending site;
      ENDIF;
   ENDIF;

   IF the packet is a diagnose packet THEN
      IF the context state indicates that the writing site
         is closed THEN
         DO close context;
      ENDIF;
   ENDIF;

   IF the packet is a maintenance or management packet THEN
      DO nothing;
   ENDIF;

   RETURN to the beginning;

END PROCESS xtp_receiver
```

5.3.7 Timer Process

There are four timers in XTP: context life timer (CTIMER), wait reply timer (WTIMER), rate control timer (RTIMER), and credit control timer (CRTIMER). All timers are maintained by the sending site. The receivers are not required to maintain any timers, thus receivers perform less function and therefore increase their throughput. The WTIMER is used by a sender to control the response time of a status request from a receiver, when the sender is blocked by flow

control. WTIMER currently is set to twice the estimated round trip delay. The CTIMER is the context keep-alive timer; currently CTIMER is set to sixty seconds. The RTIMER (rate control timer) is used to time the rate control interval which is implementation dependent [8]. The credit timer fires an action to update the credit field on every active context. The duration of the credit timer is one sixtieth of a second.

In our simulation CTIMER and WTIMER are turned off during the no-error simulation since they do not affect the overall performance. The credit timer is simulated as a separate process on the assumption that there is separate hardware to perform this function.

The CTIMER, the WTIMER and the RTIMER are managed by a timer process in our simulation. The timer process delays the length which is specified by the first event item in the callout list. Then the timer process waits for either the signal for timer expiration or the signal for a new insertion in front of the callout list. A routine called arm_time is provided for arming any one of the three timers. Arm_timer is responsible for performing event insertion on the calloutl_1 queue, which is similar to the timer call-out-list in many operating systems. If an insertion is made in front of the first item in the calloutl_1 queue, the arm_timer routine will signal the timer server process to restart the time count down. Whenever the timer process receives a timer-expired signal from the timer server

process, it removes the first event item from the callout list and passes the event item to a process interrupt handler. The timer interrupt handler performs operations associated with a specific timer: If the timer expired is the CTIMER, the interrupt handler will check the number of times that the WTIMER has gone off since the CTIMER went off. If the number is greater than four times, the interrupt handler will terminate the simulation. Otherwise the interrupt handler will send one control packet with the status request bit being turned on and will start a WTIMER. Once a status request packet is sent the ctimer count will be incremented. The simulated timer process has following functions: (a) to wait for the ARM_TIMER signal, (b) to delay a period specified by the first event item in the call-out-list and wait for the ARM_TIMER signal, (c) to perform timer interrupt handling and to restart from (a), or to restart from (b). Figure 32 is the finite state machine for the timer process. Listing 8 contains the pseudo-code for the timer process.

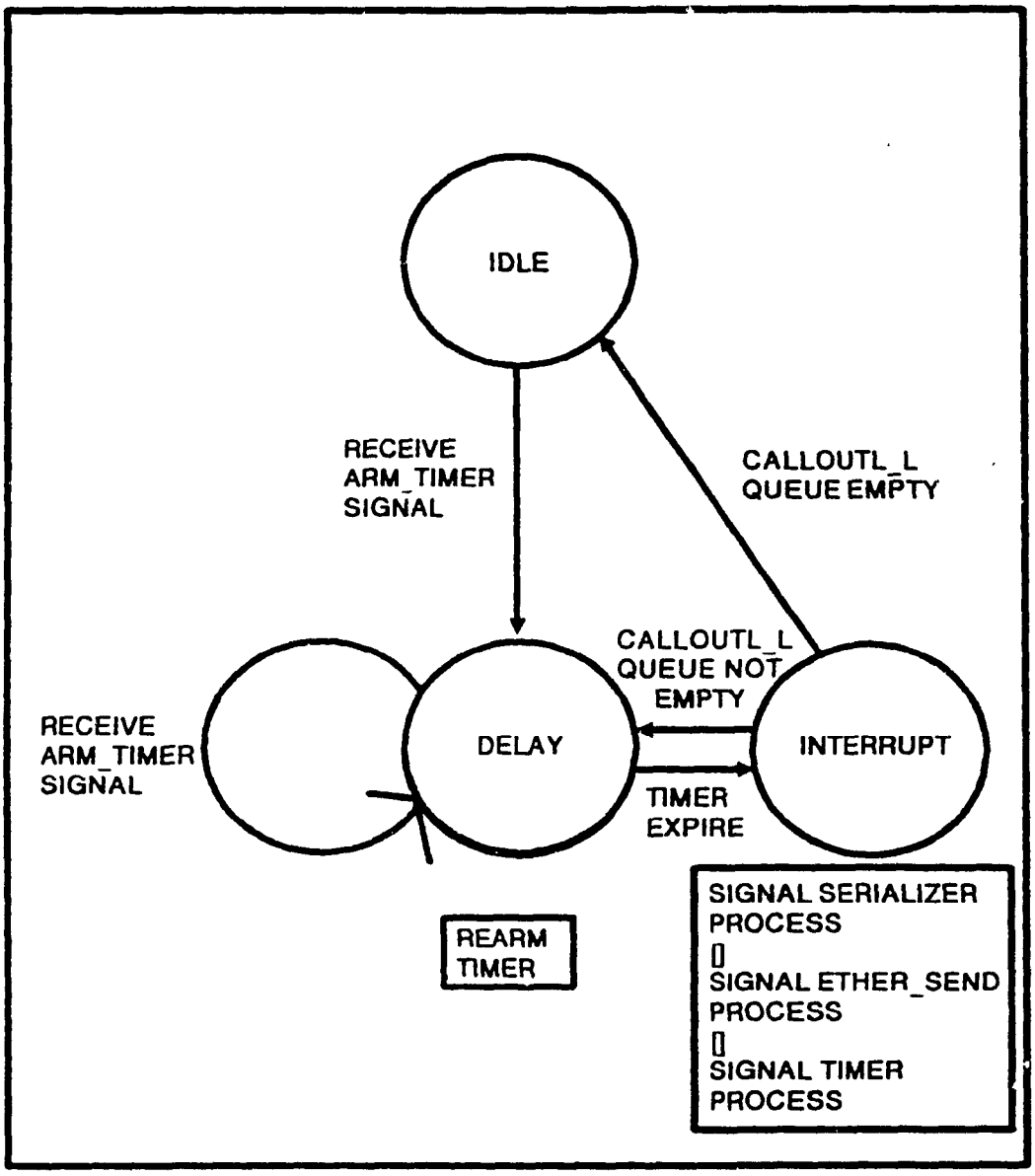


Figure 32 Finite state machine for the timer process

Listing 8 Pseudo code for the timer process

```
PROCESS timer

   Created by: LANSF simulator;

   Input signals:
      ARM_TIMER from the xtp_sender process or
                the rate_controller process or
                the timer process;
      TIMER_OFF from the LANSF timer server;

   Output signal:
      PROCESSOR_START to the serializer process;
      ETHER_SEND to the ethernet_sender process;

BEGIN

   WAIT for the ARM_TIMER signal;

   IF the calloutl_1 queue is empty THEN
      RETURN to the beginning;
   ENDIF;

   WAIT  for  either the TIMER_OFF or ARM_TIMER signal;

   IF the received signal is TIMER_OFF THEN
      DEQUEUE an event item from the calloutl_1 queue;
      PASS the event item to the timer interrupt handler;
      RETURN to the beginning;
   ENDIF;

   IF the received signal is ARM_TIMER THEN
      REARM timer;
   ENDIF;

END PROCESS timer
```

5.3.8 Credit Control Process

The credit_timer process updates the credit field in every active context to the maximum default value. If the context was blocked by credit control, the credit timer signals that the virtual circuit is unblocked by turning off the credit blocked bit in the context state. The credit timer also signals the xtp_sender process that a virtual circuit has been unblocked. The duration for the credit

update process is one sixtieth of a second in our simulation. Figure 33 is the finite state machine of the credit_timer process. Listing 9 contains the pseudo-code for the credit_timer process.

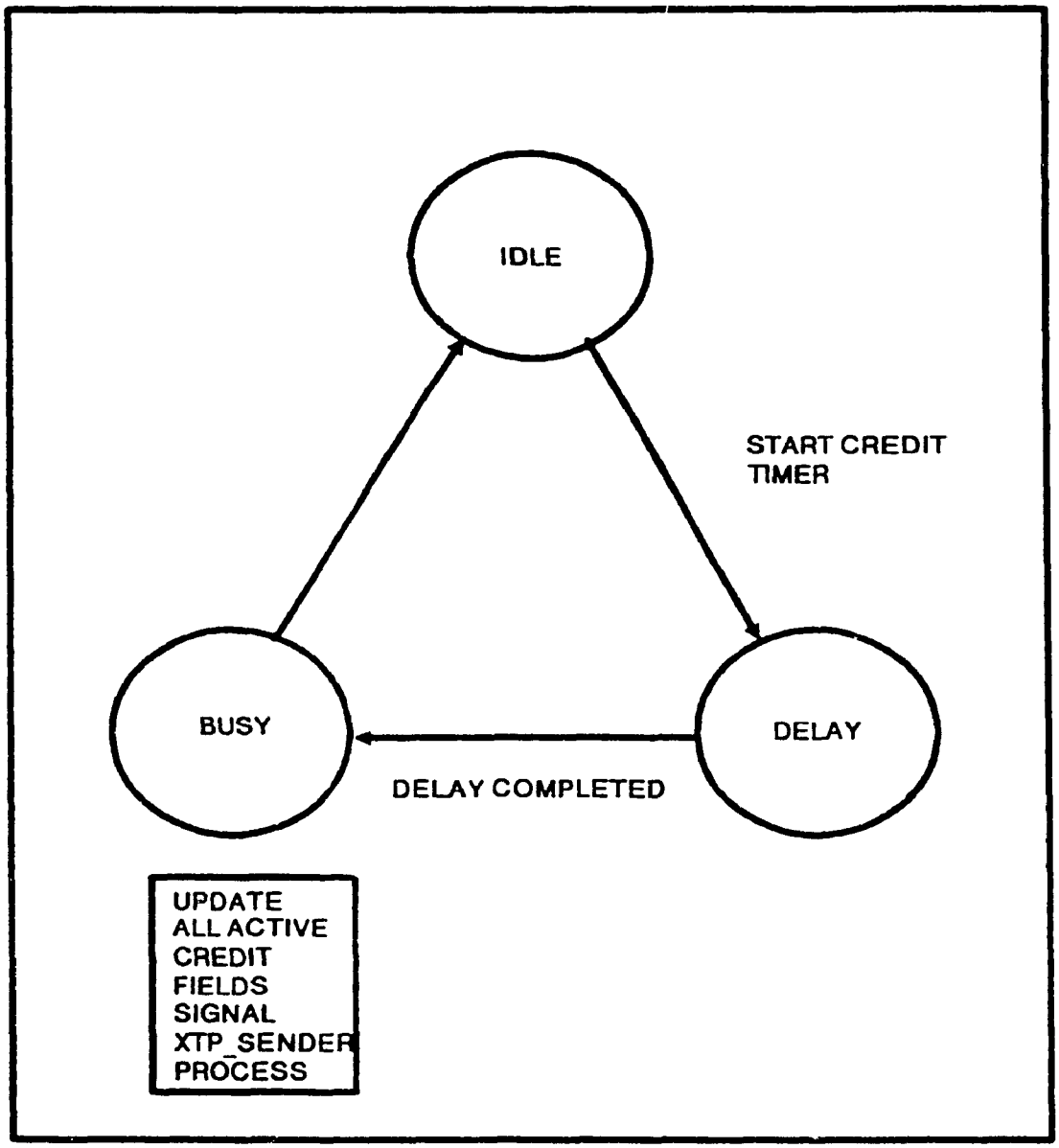


Figure 33 Finite state machine for the XTP credit timer process

Listing 9 Pseudo code for the credit_timer process

```
PROCESS credit_timer

   Created by: LANSF simulator;

   Input signals: None;

   Output signals:
      CREDIT_UNBLOCKED to the xtp_sender process;

BEGIN

   DELAY 1/60 of a second;

   WHILE there is context to be updated DO
      IF the context is blocked by credit THEN
         RESET the context state to unblocked;
         SEND  the  CREDIT_UNBLOCKED  signal to  the  sender
               process;
      ENDIF;
      UPDATE the credit field to maximum default value;
   END WHILE

   RETURN to the beginning;

END PROCESS credit_timer
```

5.3.9 Rate Control Process

The rate_controller process inserts delay between packets that go on the same route. In our simulation, each station sets up virtual circuits that are connected to other stations. However, within a station, there is no virtual circuit which goes to the same target station. As a result, the rate control is based on per station in our simulation. The simulated rate control process has the following functions: (a) to accept a packet from the serializer process, (b) to calculate the difference between the last time a packet has been sent to the receiving station and the current time, (c) to arm the timer with the remaining separation delay time, (d) to mark the context to indicate

rate blocked, (e) to wait for send-completed time from the ETHERNET sender, (f) to record the last-sent-completed time to the context and to return to (a). Figure 34 is the finite state machine of the rate_controller process. Listing 10 contains the pseudo-code for the rate_controller process.

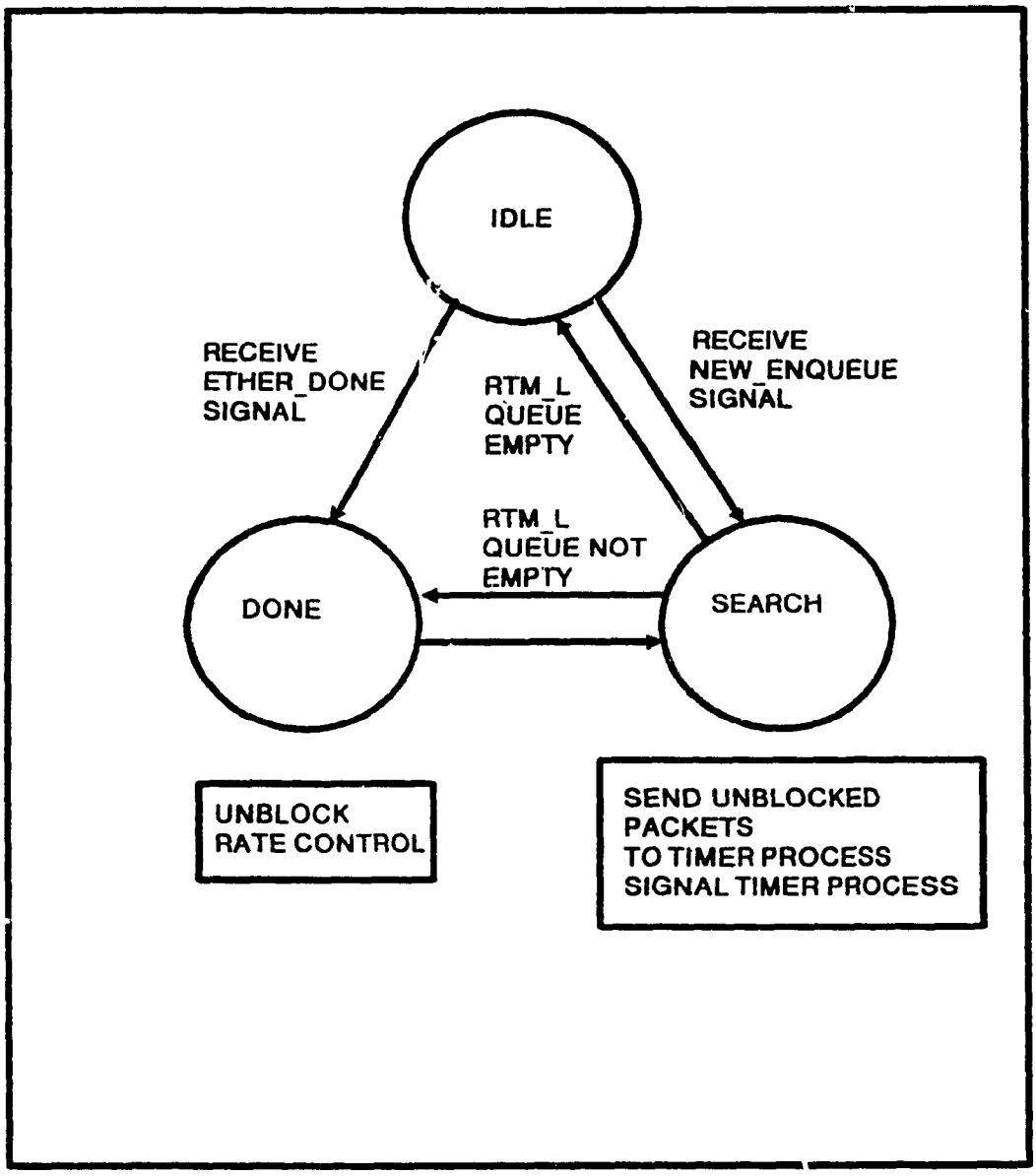


Figure 34 Finite state machine for the rate_controller process

Listing 10 Pseudo code for the rate_controller process

```
PROCESS rate_controller

   Created by: LANSF simulator;

   Input signals:
      NEW_ENQUEUE from the serializer process;
      ETHER_DONE from the ethernet_sender process;

   Output signals:
      ARM_TIMER to timer process;

BEGIN

   WAIT for either NEW_ENQUEUE or ETHER_DONE signal;

   IF the received signal is NEW_ENQUEUE THEN
      WHILE not at the end of the rcntl_1
            queue DO

         IF the packet is not blocked by rate control THEN
             REMOVE the packet from the rcntl_1 queue;
             CALCULATE the time offset of the separation;
             ARM timer process with new packet send time;
             BLOCK context for rate control;
          ENDIF;

      END WHILE

      IF the rtm_1 queue is not empty THEN
         CONTINUE  at the point where the ETHER_DONE  signal
               is tested;
      ELSE
         RETURN to the beginning;
      ENDIF;
   ENDIF;

   IF the received signal is ETHER_DONE THEN
      IF the rtm_1 queue is empty THEN
         RETURN to the beginning;
      ENDIF;

      FIND context associated with the sent packet;
      UNBLOCK rate control in the context;
      RECORD last send time in context;

      WHILE  not at the end of the rcntl_1 queue  DO
         IF the packet is not blocked by rate control THEN
            REMOVE packet from the rcntl_1 queue;
            CALCULATE the time offset of the separation;
            ARM timer process with new packet;
            BLOCK context for rate control;
```

```
      ENDIF;
    END WHILE
    RETURN to the beginning;
  ENDIF;

END PROCESS rate_controller
```

5.3.10 Serializer Process

The purpose of the serializer process is to simulate the fact that the sender, receiver, writer and reader processes are running on a single CPU. The serializer process schedules the three processes in FIFO order. Whenever a process wishes to use the processor, it passes to the serializer process an event item specifying the expected processing time delay and the signal which the process wishes to receive after the delay has expired. The serializer sleeps for the length specified by the first event item on its input queue (psr_1). When the serializer wakes up, it will check the contents of the first event item, and perform the operations according to the event. The operations are to signal the waiting process or to enqueue packets to the rcntl_1 or oup_1 queues.

The simulated serializer process has the following functions: (a) to wait for the processor-start signal, (b) to delay the period specified by the first event item, (c) to perform actions according the event type, (d) to return to (a). Figure 35 is the finite state machine for the serializer process. Listing 11 contains the pseudo-code for the serializer process.

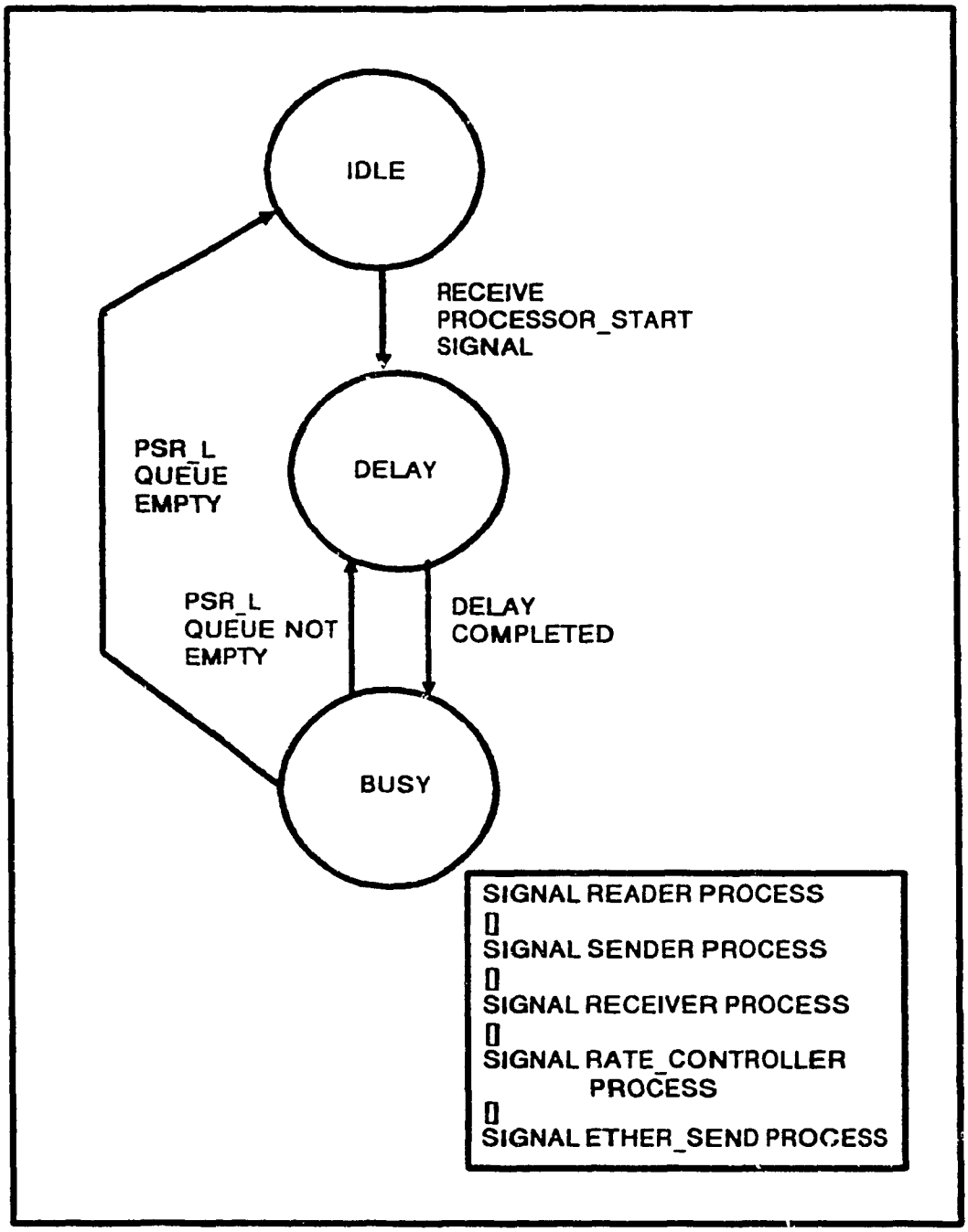


Figure 35 Finite state machine for the XTP serializer process

Listing 11 Pseudo code for the sirializer process

```
PROCESS serializer

   Created by: LANSF simulator;

   Input signals:
      PROCESSOR_START from the xtp_reader, xtp_sender, or
                      xtp_receiver process;

   Output signals:
      COPY_IN_DONE to the xtp_sender process;
      COPY_OUT_DONE to the xtp_reader process;
      CHECKSUM_DONE to the xtp_receiver process;
      ETHER_SEND to the ethernet_sender process;
      NEW_ENQUEUE to the rate_controller process;

BEGIN

   WAIT for the PROCESSOR_START signal;

   DELAY the length of the first event item in psr_1 queue;

   DEQUEUE the first event item from the psr_1 queue;

   IF the packet type specified by the event item is NONE
      THEN
      SEND signal specified by the event item;
   ELSE IF the packet type is FIRST THEN
           ENQUEUE the packet to the oup_1 queue;
           SIGNAL the ethernet transmitter to send;
           MARK the context to indicate rate control block;
   ELSE IF the packet is not blocked by rate control THEN
           ENQUEUE the packet to the rcntl_1 queue;
           SIGNAL the rate controller process that an
           unblocked packet has arrived;
   ELSE ENQUEUE the packet to the rcntl_1 queue;
   ENDIF;

   RETURN to the beginning;

END PROCESS serializer
```

5.4 The Simulation

5.4.1 The Simulation Run Environment

The performance simulation of XTP has been run on a SUN4 processor. The operating system on the SUN4 is SUN OS 4.0. The standard UNIX debugging tool dbx was used during the development of the simulation program.

5.4.2 The LANSF Tunable Parameters

There are many tunable parameters in the LANSF software. We are only interested in four of them:

(1) The message length, which specifies the length of the message that the LANSF client will generate to the XTP writer process. This parameter is specified in bits.

(2) The message interarrival time which specifies the message arrival rate according to a user-defined time unit.

(3) The number of stations in a network.

(4) The number of messages that the LANSF client should generate for a simulation run.

The rate of client level requests can be varied by changing the above four tunable parameters.

5.4.3 The XTP Tunable Parameters

There are seven tunable parameters in our XTP simulation:

(1) alloc, which defines the receiving buffer size in bytes.

(2) credit, which specifies the maximum number of bytes that a virtual circuit could send in a burst.

(3) separation, which specifies the minimum packet spacing on a per route basis.

(4) wtimer, which is the wait timer value.

(5) ctimer, which is the context life timer value.

(6) copy-delay, which is the delay caused by copying data from and to user space.

(7) checksum-delay, which is the checksum delay on the incoming and outgoing packets.

5.4.4 Measurements

The measurements that we obtained are the throughput and the individual message delay time for different message sizes under various traffic loads.

The last message completion time is recorded in the xtp_receiver process. When the xtp_receiver process receives a CNTL packet which does not have the SREQ bit on (status report packet), it checks the pre_packet field in the context to ensure that the last packet that went to the destination was the last packet of the message. If the pre_packet field indicates that the last packet was the end of the message, the xtp_receiver process will record the last message completion time in the context record.

The throughput of the simulation is calculated by dividing the number of bytes delivered to the destination user by the last message completion time. The individual

message delay is calculated by dividing the offered load by the actual throughput and multiplying by the minimum message delay time.

After the simulation has finished, a routine called calcu_result calculates the performance results using the above formulae.

To ensure the correctness of the simulation, print statements were inserted between every action to output the information associated with the process state. The first round of the simulations were run without any copy-checksum delay to ensure that the maximum effective throughput corresponded to the expected maximum throughput.

5.4.5 The Simulation Plan

The number of stations used in the XTP frame relay simulation was two: one sender and one receiver. Since normal operation of a network does not saturate the MAC layer, and since simulation of multiple conversation-pairs would only mask the properties of XTP under the contention resolution properties of the MAC layer, we have simulated only the case of a single virtual circuit at this time.

The XTP wtimer and ctimer were turned off, since they do not have any effect on the performance in a no error environment, and the expected error rate in an ETHERNET LAN are too low to produce significant effects on the throughput. The XTP rtimer was also turned off, because there is no gateway in the simulation.

The message sizes that we used in our simulations are

6 bytes, 128 bytes, 1024 bytes and 8192 bytes. The following are the reasons for choosing the above four sizes:

- The 6 byte message is the minimum number of bytes that a user can send, which fits the XTP minimum packet length criterion. This message size could represent terminal accessing activity.
- The 128 byte message represents remote procedure call activity.
- The 1024 byte message represents page fetch operations.
- The 8192 byte message represents file transfer operations.

There are ten different offered load figures used in the simulation of each message size. The offered load figures were calculated by partitioning the expected maximum throughtput into ten equal intervals. Three more offered load numbers were added to the simulation. They were calculated by adding 10, 20, and 30% of extra load to the maximum expected throughput. The last three added numbers were used to determine the performance of XTP when it is saturated. The expected maximum throughput was derived by using the ratio of the user data length to the total packet length to multiply the total bandwidth (the propagation delay is omitted).

The simulation of each message size at each offered load interval was run with and without copy-checksum delay.

The LANSF client generated 10,000 messages to XTP in each simulation run. To ensure the consistency of the simulation results, each simulation was run three times with different random number generation seeds, and the average of the three results was used.

The simulated processor has ten million instructions per second execution speed. The maximum throughput of copying data in and out of user space is 3.8 megabytes per second which includes the execution time of the read system call, locking down page, and the copying time [33]. The time to copy one byte to user space is 2.6315e-7 second. The checksum delay is also assumed to be included in the copying delay in our simulation, because it is possible to implement the checksum algorithm in such a way that the checksum result is produced at the completion of the copying process.

5.4.6 Simulation Results

For 6 byte messages, Tables 4 and 5 give the throughput and delay results, repectively. Figures 36 and 37 show graphs of the results in Tables 4 and 5, plotted versus the offered load. The column labelled "No Delay" shows the results when there is no delay due to data copying or checksum calculation. The column labelled "Delay" shows the results when data copying and checksum calculations are modelled.

Tables 6 and 7, and Figures 38 and 39, give the results

for 128 byte messages. Similarly, Tables 8 and 9, and Figures 40 and 41, give the results for 1024 byte messages. Finally, Tables 10 and 11, and Figures 42 and 43, give the results for 8192 byte messages. Note that the scales used in the various graphs are not identical. For the 8192 byte case, the messages are fragmented (by LANSF) into five 1442 byte packets and one final packet of 982 bytes. The copy/checksum delay given is the total delay; it is actually simulated in pieces corresponding to the size of the fragments.

The flat (saturation) parts of the throughput curves closely approximate the calculated maximum expected throughput. These values depend on the the ratio of "useful" bytes to the total bytes exchanged. For XTP version 3.3, these numbers are shown in the second column of the Table 12. In version 3.4, the control packet format is redefined, to permit omitting unnecessary resend pairs, which shortens the length of a control packet in the case where no errors occur. The throughput values that result for these cases are shown in the third column of Table 12, along with the percentage improvement in the fourth column.

Msg length 48(bit) 6(byte) Msg cp delay 16 (time unit) Msg chk delay 1 (time unit)		Effective Throughput			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		B/S	KB/S	B/S	KB/S
22,472	2,730	2,687	3.0	2,687	3.0
11,261	5,328	5,360	5.0	2,472	5.0
7,513	7,986	8,033	8.0	8,033	8.0
5,636	10,644	10,709	10.7	10,713	10.7
4,510	13,302	13,379	13.4	13,377	13.4
3,759	15,960	16,059	16.1	16,058	16.1
3,222	18,618	18,735	18.7	18,666	18.7
2,820	21,276	21,335	21.3	21,335	21.3
2,506	23,934	21,464	21.5	21,464	21.5
2,256	26,582	21,477	21.5	21,477	21.5
2,051	29,250	21,480	21.5	21,480	21.5
1,880	31,908	21,481	21.5	21,480	21.5
1,735	34,566	21,481	21.5	21,477	21.5

Table 4 Throughput vs offered load for 6 byte messages

Msg length 48(bit) 6(byte) Msg cp delay 16 (time unit) Msg chk delay 1 (time unit)		Message Delay			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		Time Unit	ms	Time Unit	ms
22,472	2,730	2,495	0.25	2,528	0.25
11,261	5,328	2,441	0.24	2,472	0.25
7,513	7,986	2,441	0.24	2,473	0.25
5,636	10,644	2,441	0.24	2,471	0.25
4,510	13,302	2,442	0.24	2,474	0.25
3,759	15,960	2,442	0.24	2,475	0.25
3,222	18,618	2,440	0.24	2,482	0.25
2,820	21,276	2,449	0.25	2,481	0.25
2,506	23,934	2,737	0.27	2,774	0.28
2,256	26,582	3,039	0.30	3,079	0.31
2,051	29,250	3,344	0.33	3,387	0.34
1,880	31,908	3,648	0.36	3,695	0.37
1,735	34,566	3,952	0.40	4,004	0.40

Table 5 Delay vs offered load for 6 byte messages

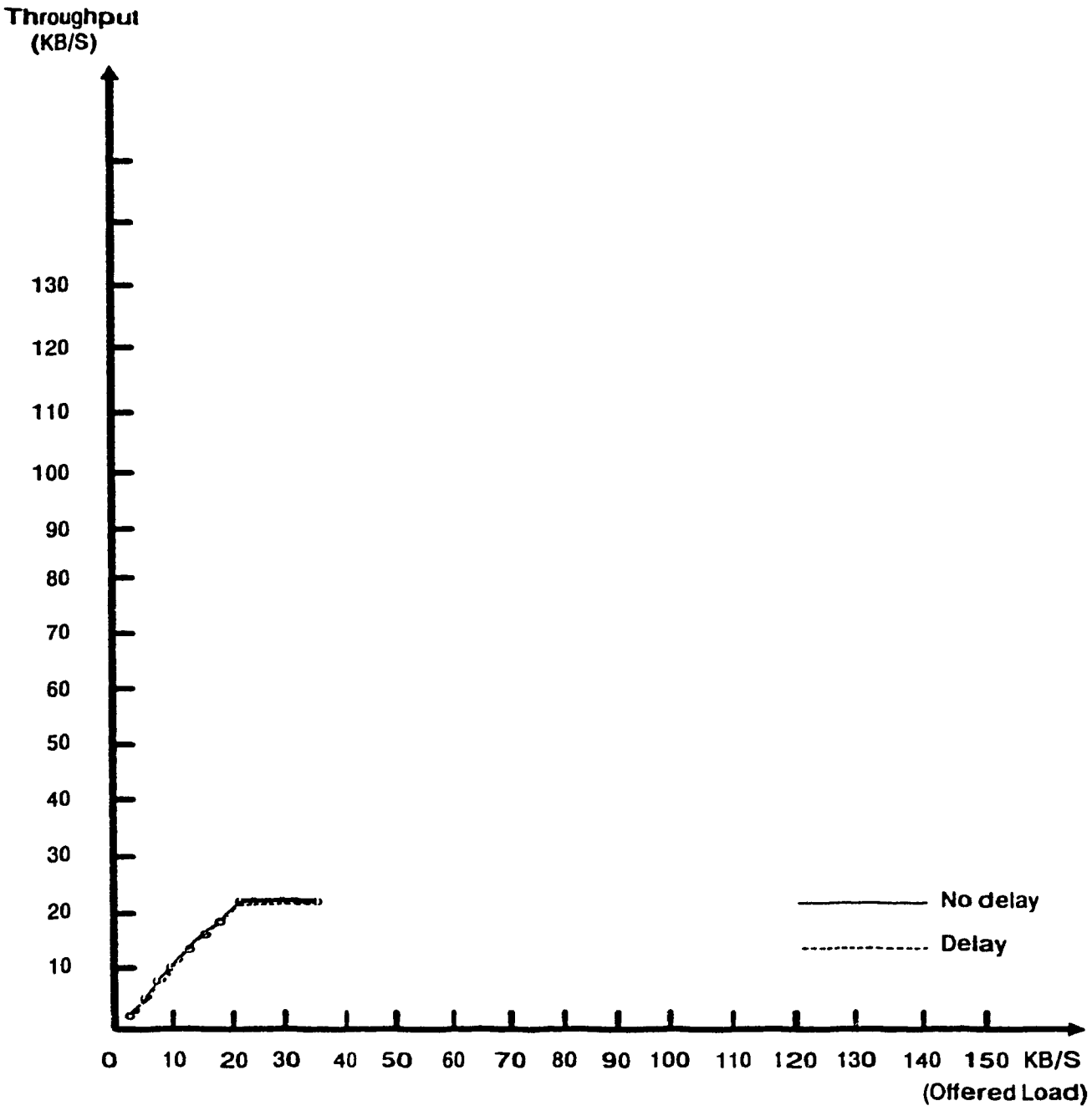


Figure 36 Throughput vs offered load for 6 byte messages

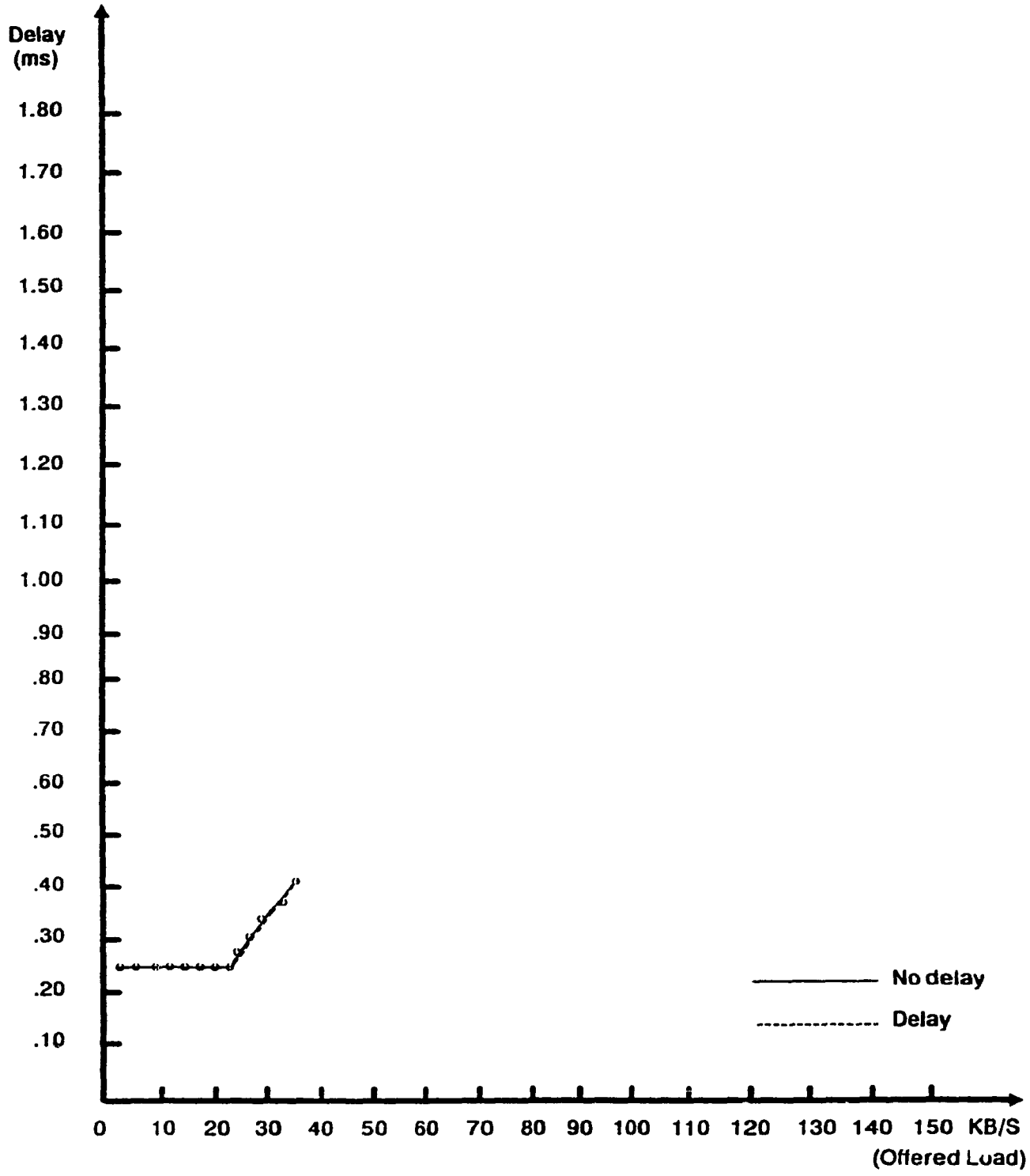


Figure 37 Delay vs offered load for 6 byte messages

Msg length 1024(bit) 128(byte) Msg cp delay 336 (time unit) Msg chk delay 1 (time unit)		Effective Throughput			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		B/S	KB/S	B/S	KB/S
32,258	39,680	39,660	39.7	39,740	39.7
16,077	79,616	79,536	79.5	79,780	79.8
10,741	119,168	119,056	119.1	119,417	119.4
8,064	158,720	158,861	158.9	158,476	158.5
6,455	198,272	198,145	198.1	198,085	198.1
5,382	237,824	237,661	237.7	237,451	237.6
4,615	277,776	277,139	277.1	276,875	276.9
4,038	316,928	316,537	316.5	316,443	316.4
3,591	356,352	356,010	356.0	344,744	344.7
3,232	396,032	370,801	370.8	346,174	346.2
2,937	435,712	371,087	371.1	345,459	345.5
2,692	475,392	371,089	371.1	345,350	345.4
2,485	515,072	371,110	371.1	345,474	345.5

Table 6 Throughput vs offered load for 128 byte messages

Msg length 1024(bit) 128(byte) Msg cp delay 336 (time unit) Msg chk delay 1 (time unit)		Message delay			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		Time Unit	ms	Time Unit	ms
32,258	39,680	3,343	0.34	4,089	0.41
16,077	79,616	3,435	0.34	4,096	0.41
10,741	119,168	3,435	0.34	4,095	0.41
8,064	158,720	3,428	0.34	4,110	0.41
6,455	198,272	3,434	0.34	4,108	0.41
5,382	237,824	3,434	0.34	4,110	0.41
4,615	277,776	3,439	0.34	4,117	0.41
4,038	316,928	3,436	0.34	4,110	0.41
3,591	356,352	3,350	0.34	4,242	0.42
3,232	396,032	3,666	0.37	4,695	0.47
2,937	435,712	4,030	0.40	5,187	0.52
2,692	475,392	4,397	0.44	5,649	0.56
2,485	515,072	4,763	0.48	6,118	0.61

Table 7 Delay vs offered load for 128 byte messages

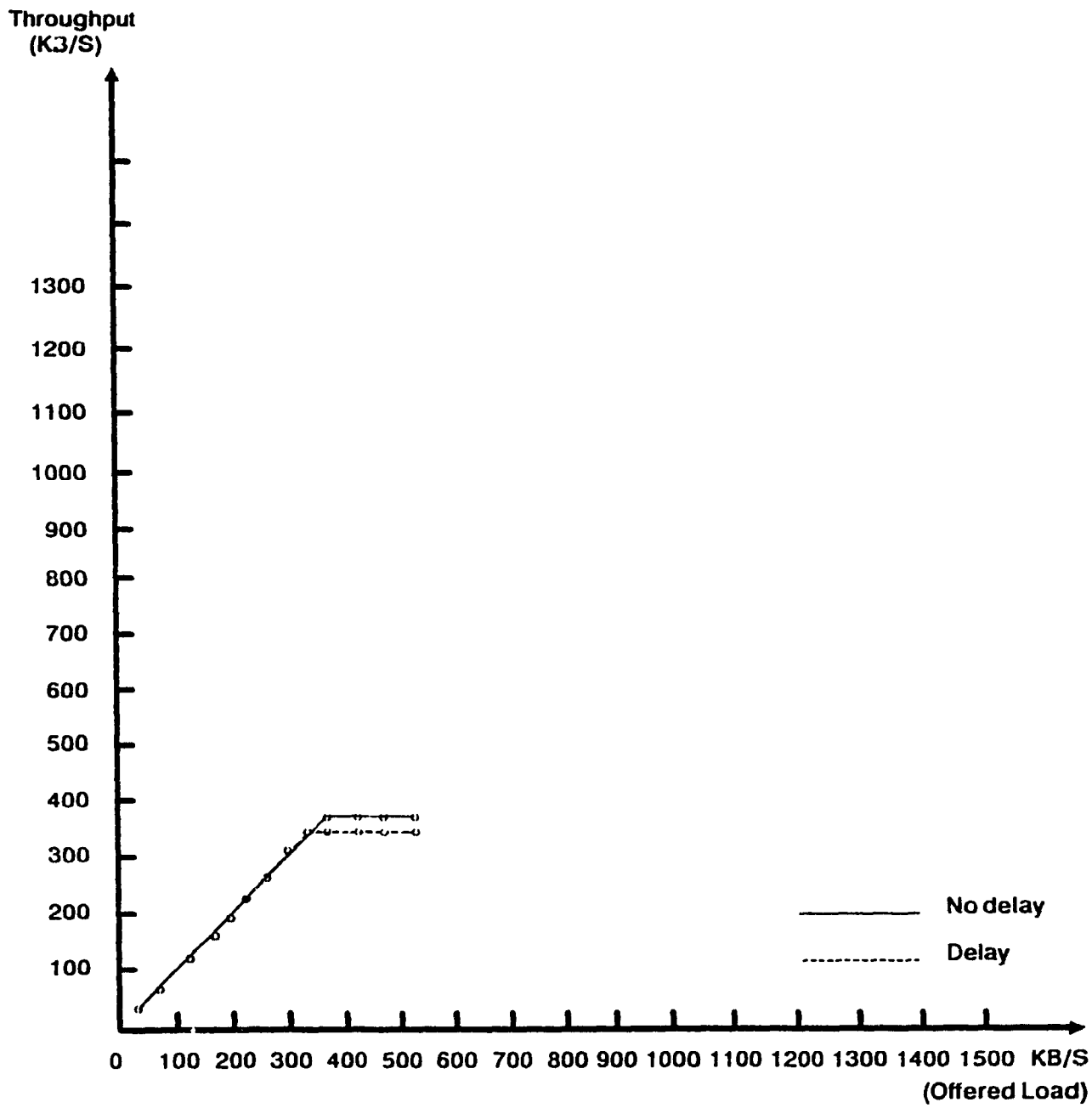


Figure 38 Throughput vs offered load for 128 byte messages

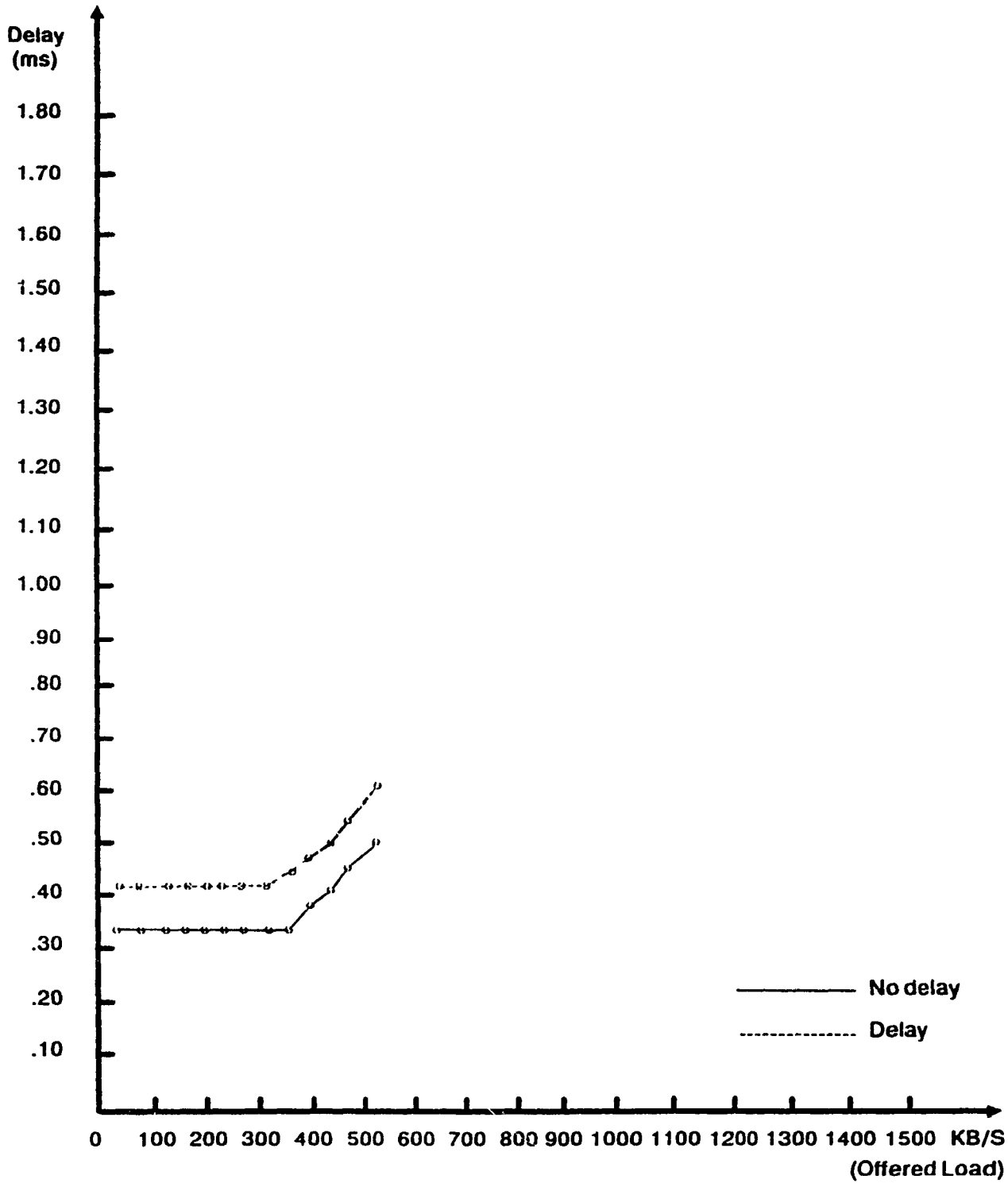


Figure 39 Delay vs offered load for 128 byte messages

Msg length 8192(bit) 1024(byte) Msg cp delay 2,694 (time unit) Msg chk delay 1 (time unit)		Effective Throughput			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		B/S	KB/S	B/S	KB/S
104,166	98,324	98,127	98.1	98,715	98.7
51,813	197,632	197,240	197.2	198,476	198.5
34,602	295,936	295,442	295,4	297,123	297.1
25,974	394,240	393,973	394.0	395,864	395.9
20,790	492,544	491,582	491.6	494,388	494.4
17,331	590,848	588,974	590.0	593,127	593.1
14,858	689,152	685,739	685.7	681,572	681.6
13,003	787,456	785,844	785.8	761,952	762.0
11,645	875,520	883,920	883.9	761,306	761.3
10,405	984,064	961,325	961.3	760,000	760.0
9,460	1,082,368	964,367	964.4	738,966	739.0
8,673	1,184,748	964,318	964.3	738,269	738.3
8,006	1,278,976	964,362	964.4	739,793	740.0

Table 8 Throughput vs offered load for 1 Kbyte messages

Msg length 8192(bit) 1024(byte) Msg cp delay 2,694 (time unit) Msg chk delay 1 (time unit)		Message Delay			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		Time Unit	ms	Time Unit	ms
104,166	98,324	10,621	1.06	15,925	1.59
51,813	197,632	10,621	1.06	15,920	1.59
34,602	295,936	10,617	1.06	15,924	1.59
25,974	394,240	10,607	1.06	15,922	1.59
20,790	492,544	10,621	1.06	15,928	1.59
17,331	590,848	10,633	1.06	15,926	1.59
14,858	689,152	10,652	1.07	16,165	1.62
13,003	787,456	10,621	1.06	16,523	1.65
11,645	875,520	10,499	1.05	18,387	1.84
10,405	984,064	10,850	1.09	20,701	2.07
9,460	1,082,368	11,897	1.19	23,417	2.34
8,673	1,184,748	13,023	1.30	25,656	2,57
8,006	1,278,976	14,058	1.41	27,640	2.76

Table 9 Delay vs offered load for 1 Kbyte messages

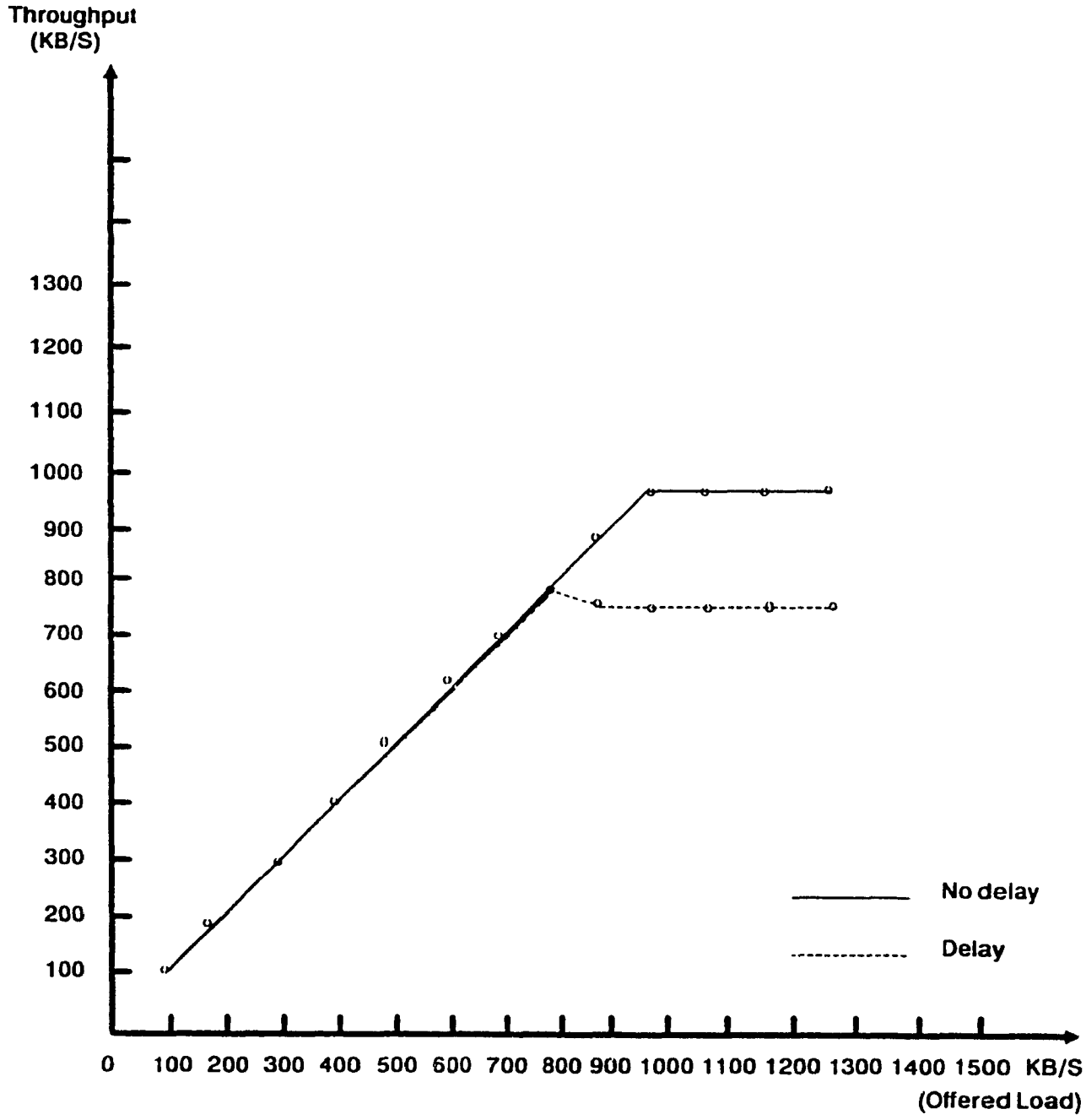


Figure 40 Throughput vs offered load for 1 Kbyte messages

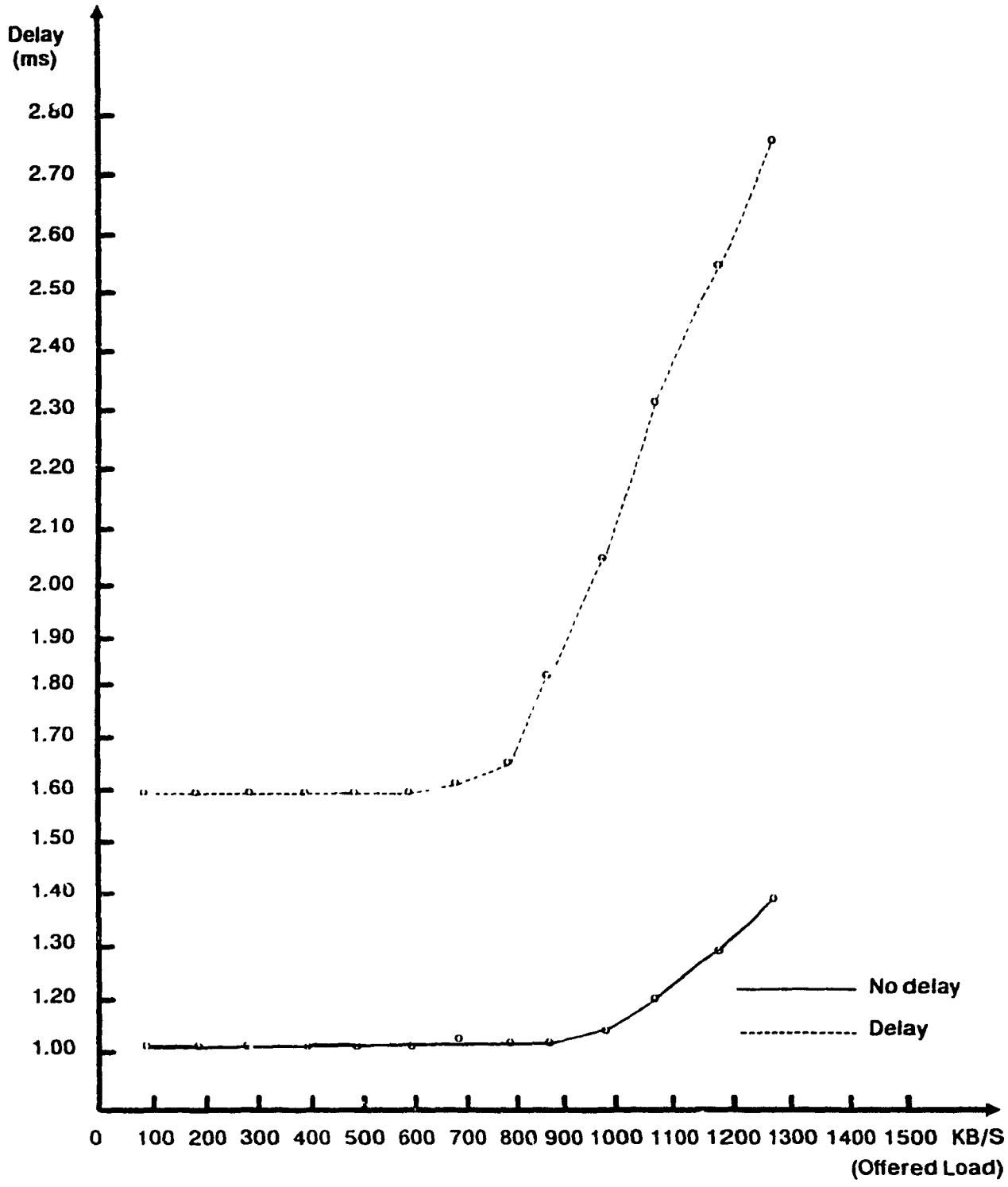


Figure 41 Delay vs offered load for 1 Kbyte messages

Msg length 65,536(bit) 8192(byte) Msg cp delay 21,559 (time unit) Msg chk delay 1 (time unit)		Effective Throughput			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		B/S	KB/S	B/S	KB/S
714,285	114,688	112,692	112.7	112,651	112.7
357,142	229,376	132,313	132.3	230,035	230.0
238,095	344,064	346,475	346.5	345,023	345,0
178,571	458,752	461,707	461.7	461.719	461.7
142,857	573,440	577,233	577.2	577,480	577.5
119,047	688,128	692,667	692.7	693,075	693.1
104,166	786,432	791,773	791.8	791,093	791.1
89,285	917,502	923,919	923,9	917,416	917.4
79,363	1,032,192	1,038,653	1,038.9	1,005,057	1,005.1
71,428	1,146,880	1,125,287	1,125.3	1 001,730	1,001.7
64,935	1,261,568	1,125,169	1,125.2	998,929	998.9
59,524	1,376,256	1,127,718	1,127.7	995,570	995.6
54,945	1,490,944	1,126,562	1,126.6	991,874	991.9

Table 10 Throughput vs offered load for 8 Kbyte messages

Msg length 65,536(bit) 8192(byte) Msg cp delay 21,559 (time unit) Msg chk delay 1 (time unit)		Message Delay			
Msg Inter (Time Unit)	Msg Inter (Byte/Second)	No Delay		Delay	
		Time Unit	ms	Time Unit	ms
714,285	114,688	72,017	7.20	115,941	11.59
357,142	229,376	70,171	7.02	113,555	11.36
238,095	344,064	70,272	7.03	113,565	11.36
178,571	458,752	70,311	7.03	113,150	11.32
142,857	573,440	70,299	7.03	113,085	11.31
119,047	688,128	70,300	7.03	113,069	11.31
104,166	786,432	70,286	7.03	113,211	11.32
89,285	917,502	70,273	7.03	113,892	11.39
79,363	1,032,192	70,323	7.03	116,956	11.70
71,428	1,146,880	72,122	7.21	130,383	13.04
64,935	1,261,568	79,342	7.93	143,823	14.38
59,524	1,376,256	86,360	8.64	157,428	15.74
54,945	1,490,944	93,652	9.37	171,182	17.12

Table 11 Delay vs offered load for 8 Kbyte messages

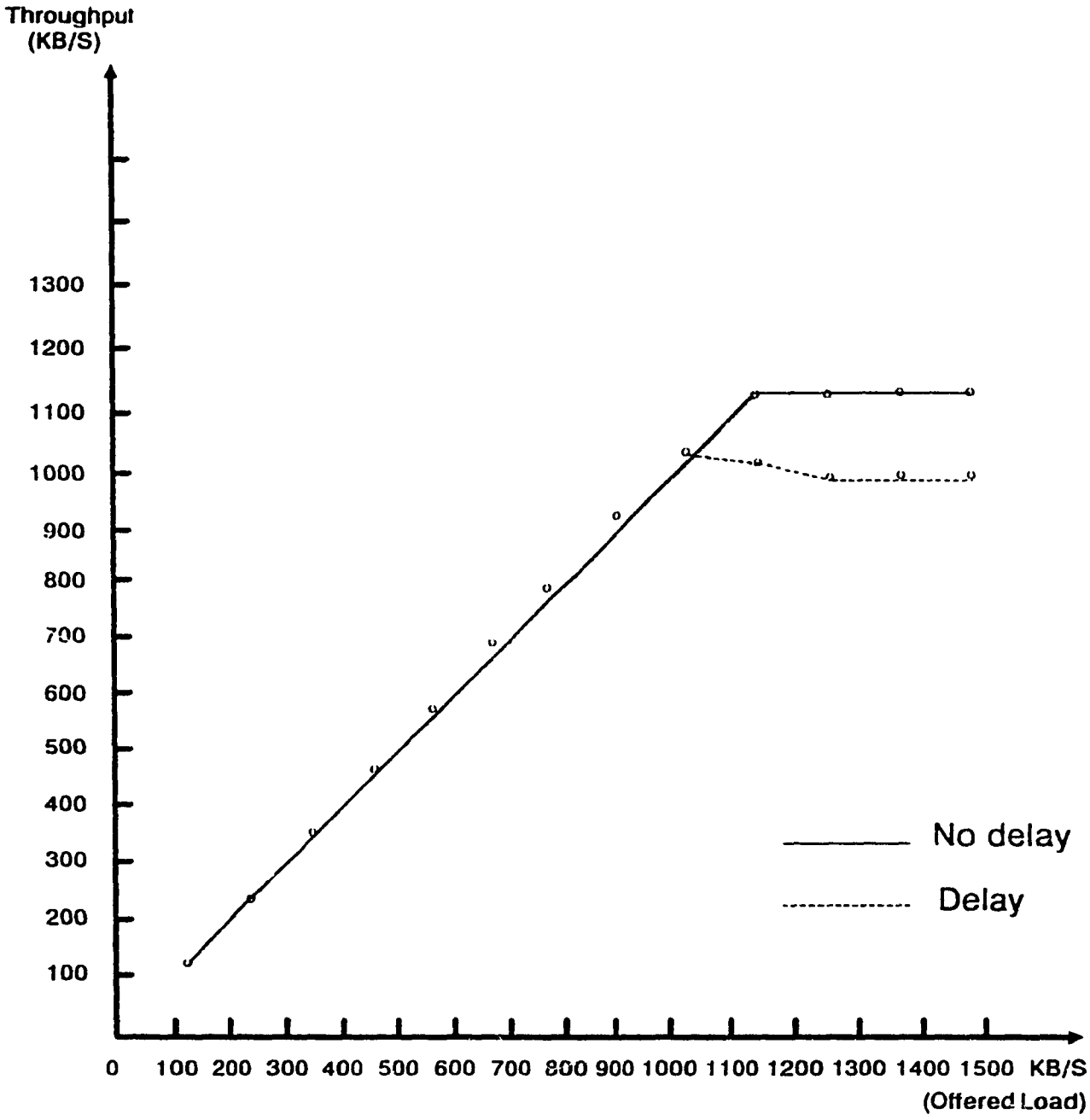


Figure 42 Throughput vs offered load for 8 Kbyte messages

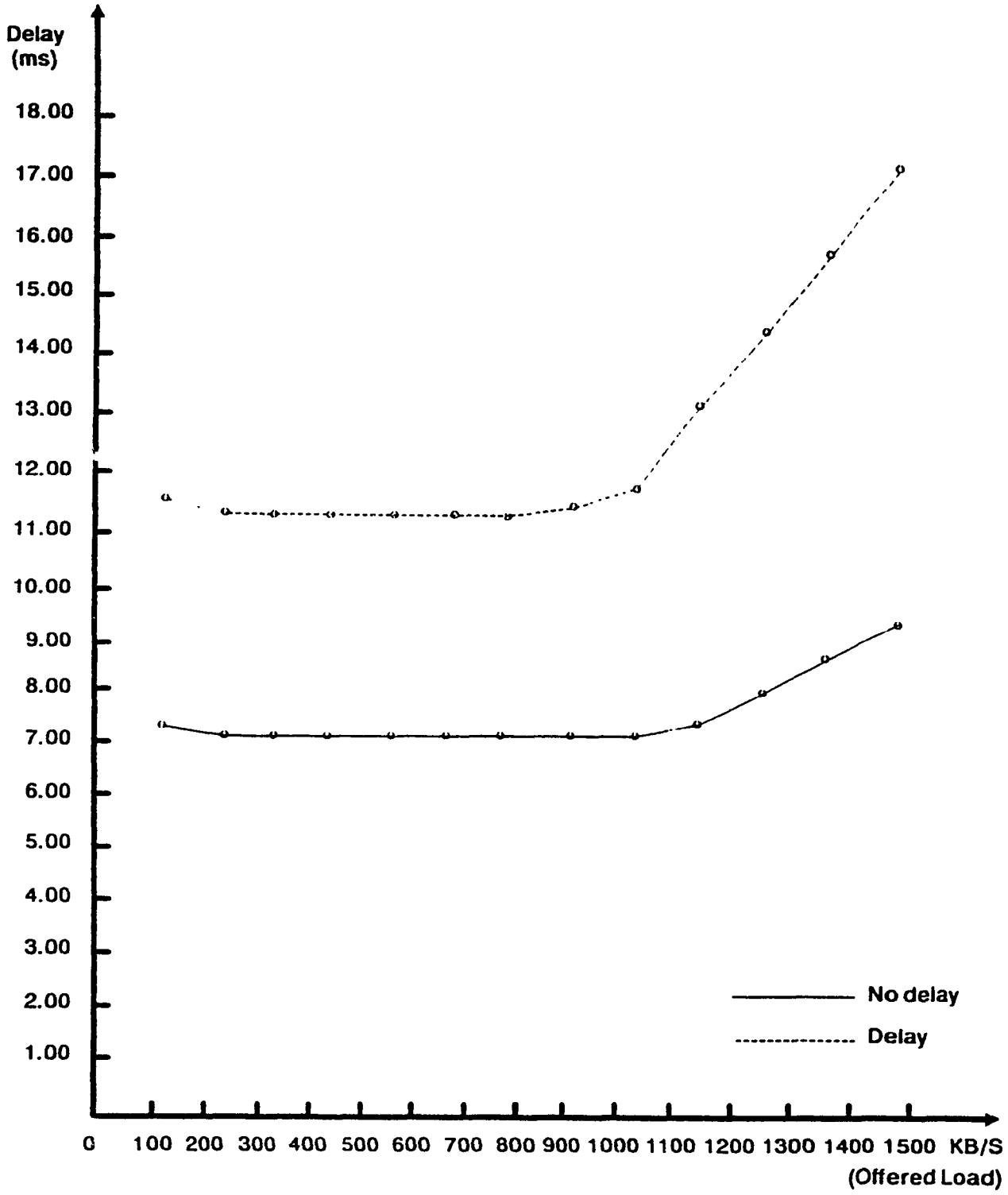


Figure 43 Delay vs offered load for 8 Kbyte messages

Message Length (bytes)	Throughput		
	Fixed Number of Resend Pairs (bytes)	Variable Number of Resend Pairs (bytes)	Percent Improvement (%)
6	26,596	48,701	83.0
128	396,040	579,710	46.0
1,024	984,615	1,092,150	10.9
8,912	1,169,217	1,186,559	1.5

Table 12 Throughput comparison of fixed and variable number of resend pairs

CHAPTER 6

CONCLUSION

"You can observe a lot just by watching."

Yogi Berra

6.1 Project Goals

As noted in the introduction, the goal of this project was to design and validate a simulator for XTP, based on the IEEE 802.2/3 link and physical layers, and using LANSF as a simulation tool. To do this it was necessary to extend and adapt LANSF, so that it could deal with virtual circuits; to construct a conceptual model of the XTP protocol, which permitted organization of the processes within the simulation; and to determine the ranges of the parameters that would stress the operation of the simulation sufficiently to inspire confidence in its design.

The extensions were simple to implement: they consist of adding queues for parameters passing among processes, multiplexing ability, multiple timer handling, virtual circuit capability, and error recovery to LANSF. The conceptual model was more difficult - it has to be pieced together from an understanding of the XTP protocol description. Certainly the understanding of the structure of XTP would have come much more quickly if the version 3.4 document had been available at the start of the project, and a more formal specification of XTP would have helped considerably.

Determining the stress points proved to be particularly easy. The performance of the protocol at maximum throughput is determined entirely (in the error-free case) from the propagation times and transmission times of the packets involved, plus components to account for the copying delays and checksumming delays. The performance of a transport-level connection will be lower when there is contention for the (shared) physical medium, but this is effectively a lowering of the raw data-link bandwidth, and is not a property of the transport level protocol itself.

6.2 XTP Performance

From the simulation, we find that XTP is capable of providing a highly efficient transport service to its users - up to 80% of the raw bandwidth usage in the file transfer applications in an ETHERNET environment. XTP performance also depends on how quickly data copying and checksumming can be done. The achievable throughput drops markedly when this factor is taken into account. In a no error enivironment, XTP performance for short packets is improved by 83% (6 byte packets) or 46% (128 byte packets) when the variable resend pairs of version 3.4 are used [9], because the resend pairs take up to 128 byte of space which is a considerable amount of the extra overhead in the no error case.

6.3 Future Work

6.3.1 General

In order to gain wide use, LANSF could develop a direct interface to some formal protocol specification language, such as ESTELLE or LOTOS, because one of the major shortcomings of LANSF is its inability to express timed message channels in high-level abstract terms. Another possible exploration would be to adapt a finite state model for simulation.

More research in high level protocol simulations is needed to understand how the tunable parameters affect the simulation results.

Although the formal specification techniques used during the simulation design helped to prove partial correctness of the implementation, they could not contribute to proving the total correctness because of the state explosion problem in finite state specifications and the problem of specifiying non-atomic events in the trace theory. From the experience of designing and building the XTP simulation, we found that there are strong needs to have formal specification tools that could perform automatic proofs of the total correctness of the protocols.

6.3.2 XTP and Others

To simulate the XTP multicasting operations, it will be necessary to add the following functions to the XTP LANSF program:

- Damping control for the XTP control packets;

- change the message type declaration in the testdata file to inform the standard client to generate broadcast type messages to the XTP processes.

It would also be interesting to add buffer allocation measurement to the LANSF simulation. The count field in the queue structure and the maximum value field in the context could be used for this purpose.

The current implementation could be used to investigate the performance of XTP in specific network configurations. It could also be moved on top of another MAC layer, e.g., token ring, token bus, or FDDI.

Another interesting research area would be to implement other protocols, e.g., VMTP, AMEOBA TP, TP4 or TCP using the current implementation. To change protocols, the designers only have to change the structures of the xtp_sender and xtp_receiver processes.

References

[1] Almes, G.T., Bluck, A.P., Lazowska, E.D., and Noe, J.D.: "The Eden System: A Technical Review", IEEE. Trans. Software Engineering SE-11, pp43-59, Jan. 1985.

[2] Chanson, S.T., Ravindran, K., and Atkins, S.: "LNTP - an Efficient Transport Protocol for Local Area Network", Department of Computer Science, University of British Columbia, 1986.

[3] Cheriton, D.R.: "The V Distributed System", Comm. ACM, Vol.31, No.3, pp314-332, March 1988.

[4] Cheriton, D.R.: "VMTP: A Transport Protocol for the Next Generation of Communication Systems", Proceedings of Sigcomm 86, Stowe, Vt.(Aug 5-7), ACM press, pp406-415, New York, 1986.

[5] Cheriton, D.R., and Mann, T.: "A Decentralized Naming Facility", Technical report STAN-CS-86-1098, Computer Science Department, Stanford University, Apr. 1986.

[6] Cheriton, D.R. and Zwaenpoel, W.: "Distributed Process Groups in V kernel", ACM Trans. on Computer Systems, Vol.3, No.2, May 1985.

[7] Chesson, G.: "Protocol Engine Project", Silicon Graphics Inc., 1988.

[8] Chesson, G., Eich, B., Schryver, V., Cherenson, A., and Whaley, A.: "XTP Protocol Definition Revision 3.3", Protocol Engines Inc., 1988.

[9] Chesson, G., Eich, B., Schryver, V., Cherenson, A., and Whaley, A.: "XTP Protocol Definition Revision 3.4", Protocol Engines Inc., 1988.

[10] Clark, D.D.: "Window and Acknowledgment Strategy in TCP", Internet Protocol Implementation Guide, Network Information Center, SRI International, Menlo Park, Calif., Aug. 1982.

[11] Clark, D.D., Lambert, M.L., and Zhang, L., "NETBLT: A bulk data transfer protocol", DARPA Network Working Group Request for Comments 969, Network Information Center, SRI International, Melno Park, Calif, Dec. 1985.

[12] Cooper, E.: "Replicated Procedure Call", 10th Symp. on Operating Systems Principles, pp63-78, Dec. 1985.

[13] DARPA : "Transmission Control Protocol", Request for Comments Number 793, 1981.

[14] Distributed System Group: "V - System 6.0 Reference Manual", Leland Stanford Junior University, 1986.

[15] Fletcher, J.G., and Watson, R.W., "Mechanisms for a Reliable Timer-Based Protocol", Computer Networks 2, North-Holland, Amsterdam, The Netherlands, pp 271-290, 1978.

[16] Gburzynski, P., and Rudnicki, P., "LANSF: A Configurable System for Modeling Communication Networks version 1.4", University of Alberta, 1988.

[17] Gifford, K., Needham, R.M., and Schroeder, M.D.: "The Cedar File System", Comm. ACM., Vol.31, No.3, pp288-298, Mar 1988.

[18] ISO : "International Standard ISO/DIS 8073", 1983.

[19] Liskov, B.: "Distributed Programming in Argus", Comm. ACM., Vol.31, No.3, pp300-312, Aug. 1988.

[20] Litman, A.: "The DUNIX Distributed Operating System", Operating System Review, ACM press, Vol. 22, No. 1, pp42-51, Jan. 1988.

[21] Marlow, D.: "NSWC Tactical LAN Transport Protocol Effort PEI TAB Meeting 5/12/88", 1988.

[22] Needham, R.M. and Herbert, A.J.: "The Cambridge Distributed Computing System", Addison-Wesley, Reading, Mass, 1982.

[23] Popek, G.J., and Walker, B.J.: "The LOCUS Distributed System Architecture", MIT Press, London, England, 1988.

[24] Popesen-Zeletin, R.: "Some Critical Considerations on the ISO/OSI RM from a Network Implementation Point of View", Proceedings of Sigcomm 84, Vol.14, No.2, ACM press, pp188-193, New York, 1984.

[25] Rashid, F.R., and Rolartson, G.G.: "Accent: A Communication Oriented Network Operating System Kernel", ACM 8th Symp. Operating Systems Principles, ACM press, New York, pp64-75, 1981.

[26] Stallings, W.: "Data and Computer Communication", Macmillan Publishing Co. Inc., U.S., 1985.

[27] Tanenbaum, A.S., and van Renesse, R.: "Distributed Operating Systems", vol.17, pp417-470, Dec. 1985.

[28] Tanenbaum, A.S.: "Computer Networks", Prentice-Hall Inc., New York, 1981.

[29] Thacker, C.P., Stewart, L.C., and Scatterthwaite, E.H., Jr.: "Firefly : A Multiprocessor Workstation", IEEE Trans. on Computers, Vol. 37, No.8, Aug. 1988.

[30] van Renesse, R., and Tanenbaum, A.S.: "Performance of the World's Fastest Distributed Operating System", Operating System Review, ACM press, Vol.22, No.4, pp25-34, Oct. 1988.

[31] Watson, R.W., and Mamrak, S.A.: "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choice", Transactions on Computer Systems, ACM press, Vol.5, No.2, pp99-120, May 1987.

[32] Welch, B.: "The Sprite Remote Procedure Call System", Technical Report UCB/CSP 86/261, Computer Science Division (EECS), University of California, Berkeley, 1985.

[33] Whaley, A.D.: "Some Speed Measurment on a 10 MIPS CPU", Protocol Engines Inc., April 1989.

[34] Yu, A., Atwood, J.W., and Radhakrishnan, T.: "Enhancing a Local Area Network for Internetworking", 12th Conference on Local Computer Networks Proceedings, IEEE Computer Society, pp66-71, October 1987.

APPENDIX A

Figure 44 is the directory layout of the LANSF package [16].

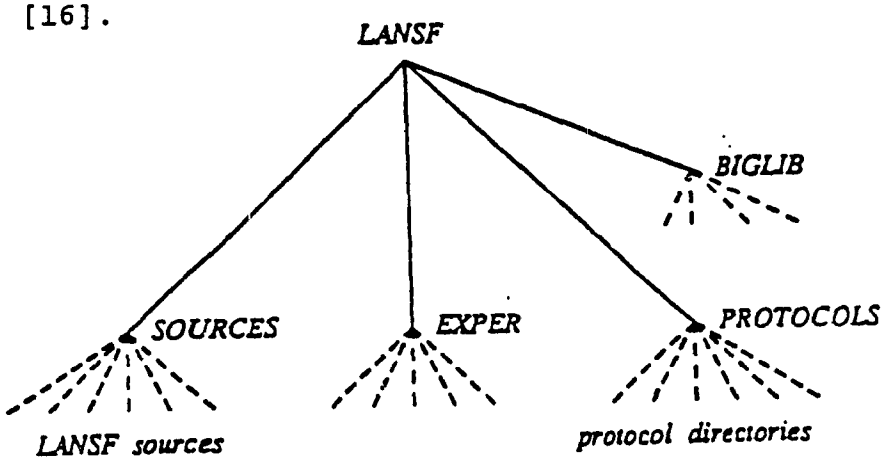


Figure 44 Directory layout of the LANSF package.

The following is a summary of how to use LANSF [16].

A user specified protocol must be named "protocol.c", and the necessary definitions for protocol.c must be contained in a file named "protocol.h". To compile the executable version of LANSF for a particular protocol, the user executes ~user/LANSF/EXPER/mk -p <directory name>. <directory name> is the name of the directory which contains the desired procotol specification. There are other optional flags for other compilation purposes. After a successful compilation, an executable file called lansf is produced in the same directory where the original specification resided.

To run the simulation, the user can type lansf testdata. The testdata is a file containing the simulation test data (which will be explained later). By simply omitting the testdata in the command line, the user can type in the test data from his or her console.

If the testdata file is not in the directory with the lansf executable file, the path name of the testdata must be fully specified in the command line.

There are other option flags available for user to run the simulation. They are listed in Appendix B of [16].

The test data file must be specified in four sections :

(1) Time section

In this section, the user must specify the number of ITU's in a virtual second [16].

(2) Configuration section

In this section, the user has to specify the network configuration. The parameters include the number of station(s), ports, links, port assignment to each station and distance between stations, number message types, message inter-arrival time, message minimum and maximum length, and number of senders and receivers.

(3) Protocol-specific section

In this section, the user can specify propagation delay, minimum and maximum packet length, and header plus trailer size (all in bits). The other parameters are minimum and maximum space between packets, and minimum, maximum jam length.

(4) Exit conditions

The simulation exit conditions consist three choices: maximum number of message, virtual time limit, and CPU time limit. If any one of above conditions is met, the simulation will be terminated.

The structure of the protocol.c file consists of three parts:

(1) Initialization

This section starts with the declaration of in_protocol(). Within the section, the user can read in all protocol-specific parameters, by using the following predefined functions provided by LANSF.

read_integer() returns a long integer.

read_real() returns a real number.

read_big() returns a non-negative integer which is greater than the capacity of the long format. This function is mainly used for inputting integers that are used for ITU's.

(2) Protocol-specific output

This section starts with a predefined declaration - out_protocol(). The function of this section is to output protocol specific results after the simulation run.

(3) Protocol-specific processes

This section is right after the out_protocol() section. In this part, the user can define all the necessary processes for his or her protocol. The following are all the predefined functions that the user can use in the process of designing his or her protocol. It is a summary from [16].

a. For output purposes

```
out_integer( ivalue, string );
int ivalue;
```

```
char *string;
```

This outputs an integer with a string for explanation purposes.

```
out_real( rvalue, string );
float rvalue;
char *string;
```

This outputs a real number with a string.

```
out_big( bvalue, string );
TIME bvalue;
char *string;
```

This outputs a TIME type value, and a string.

```
out_text( svalue, string );
out_string( string );
char *string;
```

This outputs a string.

b. Creating a process

```
int (*new_process ( code, version )) ()
int (*code)(), version;
```

This creates a separate process in a station. The code is a pointer to the process code which actually is a function in C. The version allows user to create several processes which executes the same code (similar to forking in UNIX). However, there is no forking in LANSF, version number is appended to the function name for the purpose of distinguishing the processes.

```
terminate;      /* stop a process */
```

c. Wait requests

```
wait_event( server, event, act );
int server, act;
int event;
```

Server are predefined as TIMER, DELAY, CLIENT, and SIGNAL.

Event can be defined as integer, character, or other. Act is the action supposed be taken when the waiting event occurs.

0 - 1023 the port server, the parameter value is equal to port number;

1024 the client (this value is assigned to symbolic constant CLIENT);

1025 the basic timer server (constant TIMER);

1026 the alternate timer server (constant DELAY);

1027 the signal server (constant SIGNAL);

```
continue_at( act );
```

It is same as :

```
wait_event( DELAY, 0, act );
return;
```

d. Client

User can either use the standard client provided by LANSF, or can define his or her own Client. Since LANSF is originally developed for MAC level protocol simulation, the standard client is not suitable for high level protocol simulations.

```
int  get_packet( mtp, min, max, frm )
int  mtp, min, max, frm;

int  get_next_packet( min, max, frm )
int  min, max, frm;
```

get_next_packet looks at all the station ports.

For both functions, the received packet will be automatically put into current_packet, and current_packet_status is set to 1.

```
release_current_packet();
```

There are three types of events a process essentially wants to wait for

```
(1) MESSAGE_ARRIVE        /* -1 */
(2) MESSAGE_INTERCEPT     /* -2 */
(3) message type identifier   /* 0 to n-1     */
TIME generate_inter_arrival_time( mtp )
long generate_message_length( mtp )
TIME generate_inter_burst_length( mtp )
int  generate_burst_size( mtp )
int mtp;
generate_sender( mtp, sender, group )
int                       mtp;
STATION                   **sender;
COMMUNICATION_GROUP       **group;
```

This generates a sender of a spcific communication group to a message.

```
STATION   *generate_receiver( mtp, sender, group )
int                       mtp;
STATION                   *sender;
COMMUNCATION_GROUP        *group;
```

This generates a receiver of a specific communication group to a message.

```
MESSAGE   *generate_message( mtp, sender, receiver lgth )
int                mtp;
STATION            *sender, *receiver;
long               lgth;
```

This function generates a user specified type message, and appends the message to the station message queue.

```
PACKET    make_packet( snd, rcv, lgth )
STATION   *snd, *rec;
long      lgth;
```

This creates a LANSF packet structure. This is a very good feature for customizing specific protocols. The shortcoming of this function is that it does not create the PCONTENTS array structure, which can be used by the user for passing protocol specific data. In our simulation make_packet() is modified to create non-standard packets that have the PCONTENTS array. The new make_packet accepts five parameters: the first three are identical to the original make_packet(); and the last two are the XTP packet type and the XTP packet command.

e. Port events

The functions that are associated with ports are mainly used by MAC level simulations. Thus, they are not discussed in this thesis. The interested reader can refer to [16].

f. Starting and Terminating Transmissions

Again, the functions that are associated with packet transmission are mostly used by MAC level protocol simulations, and therefore, they are not discussed in detail.

```
start_transfer( port_id, packet )
int       port_id;
PACKET    packet;
transmit_packet( port_id, packet, eot_action )
int       port_id, eot_action;
PACKET    packet;
stop_transfer( port_id )
int port_id;
abort_transfer( port_id )
int port_id;
```

g. Generating Random Numbers

For same reason mentioned above, they are not discussed here.

h. Operations on Flags

```
set_flag( flags, n )
clear_flag( flags, n )
FLAGS     flags;    /* #typedef FLAGS long   */
int       n;        /* the bit position */
```

set bits on flags associated with packets.

int flags_set(flags, n)

int flags_clear(flags, n)

These two function will return the original contents of the bit.

i. Signal

This is most important feature in LANSF.

```
generate_signal( n, s, i1, i2 )
int       n;        /* signal number */
int       s;        /* station number */
char      i1, i2;   /* values to be returned */
send_signal( n, s );
internal_signal( n );
clear_all_signals();
priority_signal( n );
int       n;
```

j. Error Handling

```
excptn( string )    /* terminate the simulation due to
                       error condition, similar to
                       perror(); exit(); */
char      *string;  /* used for stdout */
assert( cond, string )
int  cond;
char *string;
```

k. Memory Management

```
char      *memreq( size )
```

```
int         size;
release( prt, size )
char        *ptr;
int         size;
```

1. Receiving Packets

```
accept_packet( packet, port_id )
PACKET      *packet;
int         port_id;
my_packet(  packet  )    /* packets associated with  the
                            current station */
PACKET      *packet;
```

m. Operations on Big numbers

LANSF provides a number of possible operations on large integer numbers such as TIME type variables. The interested reader can refer to [16].

Appendix B Funtion Call Chart for LANSF

```
 1  clear_timer [protocol.c]
 2  credit_timer [protocol.c]
 3          wait_event
 4          generate_signal
 5          excptn
 6  dump_x [protocol.c]
 7          printf
 8  ethernet_receiver [protocol.c]
 9          wait_event
10          accept_packet
11          map_packet [protocol.c]
12                  memreq
13                  bzero
14                  bcopy
15                  append_event [protocol.c]
16                          memreq
17                          excptn
18                          bzero
19          generate_signal
20          excptn
21  ethernet_sender [protocol.c]
22          wait_event
23          last_eoa_sensed
```

```
24          l_tolerance
25          start_transfer
26          end_transfer
27          append_event [see line 15]
28          generate_signal
29          release_out_packet [protocol.c]
30                  get_e_item [protocol.c]
31                  release
32          start_jam
33          end_jam
34          backoff [protocol.c]
35                  exp
36                  l_uniform
37          excptn

38  in_protocol [protocol.c]
39          read_big
40          read_integer
41          new_process
42          d_action_list
43          d_action_item

44  lookup_delay [protocol.c]
45          printf

46  out_protocol [protocol.c]
47          out_transport [protocol.c]
```

```
48                      out_string
49                      printf
50              out_string
51              out_big
52              out_integer
53              flush_line

54      rate_controller [protocol.c]
55              wait_event
56              get_e_item
57              send_next [protocol.c]
58                      arm_timer [protocol.c]
59                              memreq
60                              excptn
61                              bzero
62                              insert_q [protocol.c]
63                              generate_signal
64                      release
65              release
66      show_q [protocol.c]
67              printf

68      timer [protocol.c]
69              wait_event
70              get_e_item
71              timer_intr_handler [protocol.c]
72                      release
```

```
73                  send_cntl [protocol.c]
74                          make_pkt [protocol.c]
75                              excptn
76                                  memreq
77                                  bzero
78                                  bcopy
79                          lock_processor [protocol.c]
80                                  append_event
                                    [see line 15]
81                                  generate_signal
82                  arm_timer [see line 58]
83                  excptn
84                  generate_signal
85                  append_event [see line 15]
86          excptn

87  initialization [protocol.c]
88          memreq
89          excptn
90          bzero
91          init_context [protocol.c]
92          terminat_

93  serializer [protocol.c]
94          wait_event
95          get_e_item
96          processor_intr_handler [protocol.c]
```

```
 97                  generate_signal
 98                  append_event [see line 15]
 99          release
100  xtp_reader [protocol.c]
101          wait_event
102          lock_processor [see line 79]
103          get_e_item
104          record_resend [protocol.c]
105                  send_cntl [see line 73]
106          release
107          send_cntl [see line 73]
108          do_close [protocol.c]
109                  send_cntl [see line 73]
110                  bzero
111                  init_context
112          excptn
113  xtp_receiver [protocol.c]
114          wait_event
115          lock_processor [see line 79]
116          get_e_item
117          get_context [protocol.c]
118          update_x [protocol.c]
119                  update_resend [protocol.c]
120                          bcopy
121                  append_event [see line 15]
```

```
122                     generate_signal
123                     send_cntl [see line 73]
124                     bcopy
125                     do_resend [protocol.c]
126                             make_pkt [see line 74]
127                             arm_timer [see line 58]
128                             bcopy
129                     do_close [see line 108]
130                     printf
131             release
132             excptn

133     xtp_sender [protocol.c]
134             wait_event
135             release_current_packet
136             get_next_packet
137             find_wait_x [protocol.c]
138             get_context
139             save_packet [protocol.c]
140                     memreq
141                     excptn
142                     bcopy
143                     append_event [see line 15]
144             send_wait [protocol.c]
145                     get_e_item
146                     send_packet [protocol.c]
147                             memreq
```

```
148                             excptn
149                             bzero
150                             bcopy
151                             lock_processor [see line 79]
152                     release
153             send_packet [see line 146]
154             excptn

155     xtp_writer [protocol.c]
156             wait_event
157             generate_signal
```