INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road. Ann Arbor. MI 48106-1346 USA
313/761-4700 800/521-0600

Simulation of the Error Control Procedures in the Xpress Transport Protocol

Duc Quang Duong

A Thesis

in

The Department

of

Computer Science

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

March, 1997

© Duc Quang Duong, 1997

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26013-5

Canada

ABSTRACT

Simulation of the Error Control Procedures in the Xpress Transport Protocol

Duc Quang Duong

Computer networking technology has seen explosive growth over the past thirty years. Along with the rapid development of high speed communication media, the demand for a more efficient transport layer protocol is increasing. Many research activities have been devoted to this area. The Xpress Transport Protocol (XTP) is one such example. XTP was designed to fix some deficiencies in the Transmission Control Protocol (TCP), and to provide a highly efficiency transport service.

This thesis presents the simulation of the error control procedures of XTP by using a simulation tool called SMURPH (System for Modeling Unslotted Real-time Phenomena), which is an object-oriented configurable simulator. Ethernet is the chosen LAN environment for the simulation of XTP in SMURPH.

The simulation results show that XTP has excellent performance, even for large file transfers in an environment with high bit error rates. The excellent performance of XTP is due to the excellent design decisions made for the error control mechanisms.

Fiber optic transmission medium is rapidly becoming available for communication networks. The low bit error rates are a beneficial result of the move toward fiber optic networks. The thesis thus investigates the performance of XTP and the behavior of the XTP error control for bulk transferring in a low bit error rate environment.

The thesis also studies the Ethernet Capture Effect problem, which results in performance degradation of XTP simulation. The thesis shows that XTP can overcome the Ethernet Capture Effect by using an enhanced backoff algorithm for its collision resolution.

ACKNOWLEDGEMENTS

I am very grateful to my thesis supervisor Dr. J.W. Atwood, whose generous support and guidance have made the completion of this thesis possible. I would also like to express my gratitude to the members of my thesis defense committee.

I would like to especially thank Mr. Guo Kun and Mr. Georges Chung Kam Chung for their support during my thesis study.

I would also like to thank my parents, my brothers, and my sister for all of their support.

Finally, I wish to thank Mrs. Sandra Wright, of Toronto, Ontario, and all Sisters at St. Joseph Mother House, of London, Ontario.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Page

LIST OF LISTINGS

Page

LIST OF ABBREVIATIONS

BEB	Binary Exponential Backoff Algorithm.
CABEB	Capture Avoidance Binary Exponential Backoff Algorithm.
ITU	Indivisible Time Unit.
FDDI	Fiber Distributed Data Interface.
FSM	Finite State Machine.
LAN	Local Area Network.
LANSF	Local Area Network Simulation Facility.
MAC	Medium Access Control.
RPC	Remote Procedure Call.
SMURPH	System for Modeling Unslotted Real-time Phenomena.
TCP	Transmission Control Protocol.
TP4	Transport Protocol class 4.
VLSI	Very Large Scale Integration.
VMTP	Versatile Message Transaction Protocol.
XTP	Xpress Transport Protocol.
WAN	Wide Area Network.

1. INTRODUCTION

Computer networking technology has seen explosive growth over the past thirty years. The transmission rate of the Local Area Network (LAN) has been increased from 10 million bits per second (Mbits/sec) in the 1970s, to the order of 100 million bits per second in the 1980s. We expect to see the deployment of Wide Area Network (WAN) gigabit per second networks in the 1990s.

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

XTP's aim of achieving high efficiency in transport service is reflected by the design decisions made in the error control protocol. This thesis presents the simulation of the error control procedures of XTP. The simulation results can be used for performance analysis of XTP.

The tool used to simulate XTP is the *System for Modeling Unslotted Real-time Phenomena* (SMURPH). The word "unslotted" means that the user has absolute freedom in specifying the flow of time as well as its granularity.

As mentioned in [Gburzynski91a], simulating a communication protocol has the advantage of being less time consuming than building a corresponding mathematical model. In addition, assumptions that are usually made in the mathematical models can be easily implemented.

The thesis consists of 8 chapters:

The second chapter introduces the background of XTP, and describes the operations of XTP, which includes connection management, data transfer, error control, termination management, and multicasting.

The third chapter explains the basic structure of the simulation tool SMURPH, which has been used in this thesis as the performance simulation tool for XTP.

The fourth chapter presents our design of the XTP simulation program. It includes all the data structures, algorithms, and pseudo code of the XTP simulation programs.

The fifth chapter documents the test plan for the XTP simulation program. It states all the assumptions on which the simulation program and the simulation test plan were based. This chapter then shows all the simulation results, as well as the analysis of the XTP throughput performance obtained from these simulation results.

The sixth chapter studies the performance of XTP and the behavior of the XTP Error Control in a very low error rate environment.

The seventh chapter explains the Ethernet Capture Effect problem in the XTP simulation. It then presents an enhanced backoff algorithm than can be implemented to solve this problem.

The last chapter is the conclusion of the thesis. It also proposes future enhancements to our work.

2. AN INTRODUCTION TO XTP

2.1 Background

XTP is designed to provide high throughput and a reliable transport service to its users, in an optical fiber network. The XTP design is motivated by the needs of contemporary and future distributed, real time, transactional, and multi-media systems, with high transmission speed, and low error rate.

The two most successful transport protocols, the Transmission Control Protocol (TCP)(1977), and the ISO Transport Protocol class 4 (TP4)(1982) were designed for an era when network transmission bandwidth was low and error rates were high. Many new generation of protocols have been designed to provide the services that TCP had failed to provide. Delta-t (for connection management)(1978), NETBLT (for bulk data transfer)(1986), VMTP (for transactions)(1986), and many other new generation of protocol designs are the result of identifying and fixing some deficiency in TCP. XTP joins this list with contributions in: **orthogonal** protocol functions for separating paradigm from policy, separation of rate and flow control, explicit first-class support for reliable multicast, and data delivery service independence.

Separation of paradigm and policy. At the core of XTP is the set of mechanisms whose functionality is orthogonal to one another. The most notable effect of this is that XTP clearly separates communication paradigm (datagram, virtual circuit, transaction, etc.) from the error control policy employed. Further, flow and rate control as well as error control can be tailored to the communication at hand. If desired, any of these control procedures can be turned off.

Separation of rate and flow control. Flow control operates on end-to-end buffer space. Rate control is a producer/consumer concept that considers processor speed and congestion. TCP does not provide rate control, and combats congestion with low-start and

other heuristics. XTP provides mechanisms for shaping rate control and flow control independently.

Explicit reliable multicast support. The transport layer multicast is a unique feature in XTP. It does not exist in other well-known transport protocols, such as TCP and TP4. The potential applications of multicast (e.g., distributed databases, distributed simulation, multimedia workstations, teleconferencing, sensor data distribution) are so numerous that multicast is XTP's most distinguishing and important feature. XTP's multicast is not an attachment to unicast; rather, each mechanism used for unicast communications is available for multicast as well. The number of communicants is orthogonal to paradigm and policy.

Data delivery service independence. XTP is a transport protocol, yet with the advent of switched networks rather than routed internetworks, a traditional network layer service may not be appropriate in every instance. XTP requires only that the underlying data delivery service provide framing and delivering of packets from one XTP-equipped host to another. XTP also employs parametric addressing, allowing packets to be addressed with any one of several standard addressing formats.

XTP takes benefit from many new generations of protocols. Its design choices aim at meeting the performance goals, the requirements of new high speed networking environment, and the demands of VLSI execution environments.

The work reported in this thesis is based on the XTP protocol specification, revision 4.0 (issued in 1995). Only a minimal description of the protocol is given in this Chapter; interested readers are referred to the most recent version of the XTP specification [XTPForum95]. Furthermore, the influences of the conventional protocols (such as TCP, TP4, Delta-t, VMTP, etc.) to the design of XTP are described in [Strayer92].

2.2 Protocol Concepts

XTP provides a means for reliably delivering packets between two or more end systems. Each host retains and exchanges state information with its peer. The state information is encoded as control information in packets. The following concepts are basic to XTP: **context, association, data stream, and packet.**

2.3 Context

The term **context** denotes state information kept within one end system. This information represents one instance of an active communication between two (or more) XTP end points. A context must be created before sending or receiving XTP packets. There may be many active contexts at an end system, one for each active conversation. Figure 2.1 illustrates active contexts in two end systems.

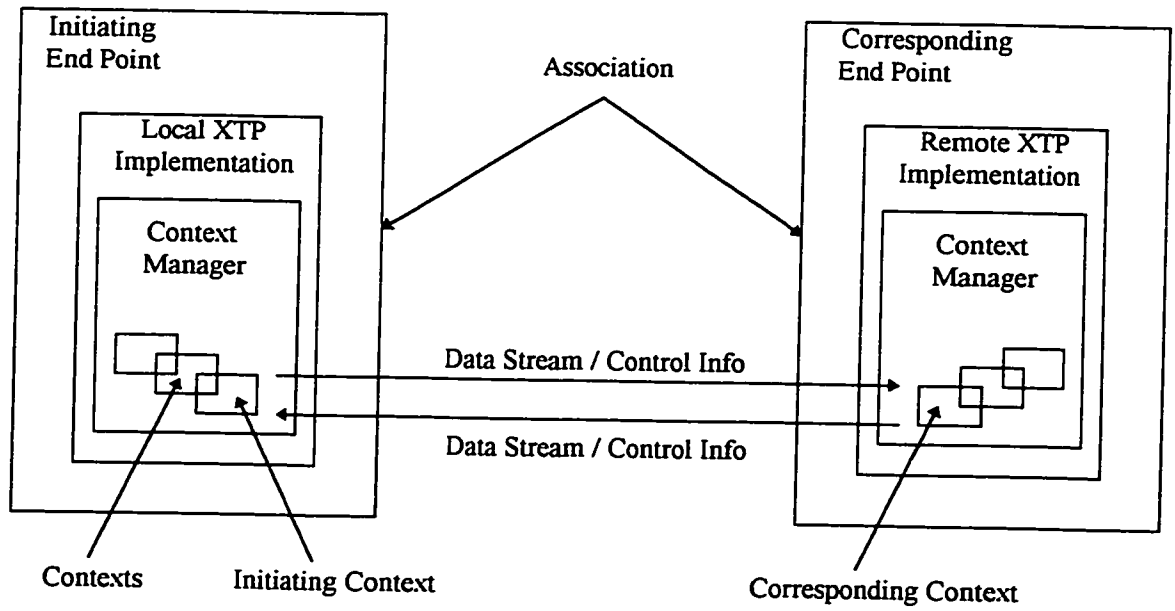


Figure 2.1: XTP Communication Model.

Each context manages both an outgoing data stream and an incoming data stream as well as the potential for both sending and receiving control information. The relationships between context, and data stream, and the exchange of control information are illustrated in figure 2.1 (for simplicity, this figure shows a unicast association; a multicast association is similar with more end points).

2.4 Association

An association is the name given to the aggregate of a pair of active contexts and the data streams between them as illustrated in figure 2.1.

The process of creating an association is not symmetric. That is, one host is an initiator and the other host is a passive listener. However, once an association exists, it is symmetric: data and control packets may flow in full duplex fashion in either direction between end systems.

2.5 Data Stream and Sequence Space

A data stream is an arbitrary length string of bytes, where each byte is represented by a sequence number. XTP sequence numbers are 64-bit unsigned integer values.

Sequence numbers provide the basis for flow control and error control between XTP end systems. Flow control regulates the volume of data that may flow between end points by controlling the portion of sequence space that may be transmitted. Data reception is acknowledged in terms of sequence numbers. Also, transmission errors and retransmission are defined in terms of sequence number pairs, called *spans*, which delineate a portion of sequence space.

2.6 XTP Packet Types and Structures

A packet is the basic unit for information exchange between the endpoints of an association. The fields within a packet hold the information pertaining either to the state of the association or to the data being transferred. In XTP, there are 7 types (figure 2.2) of packets with 2 formats: a control format and an information format (figure 2.3). Both formats have the same header but have different bodies. The control packets carry a Control Segment and are used to exchange protocol state information between contexts in an association. User information, including user data and protocol diagnostic messages, is carried in the Information Segment of the information packets.

Type	Format	Function
CNTL	Control	State Exchange control packet
ECNTL	Control	Error Control Packet
TCNTL	Control	Traffic Control Packet
DATA	Information	User Data Packet
FIRST	Information	Initial Packet of An Association
JOIN	Information	Multicast Join Packet
DIAG	Information	Diagnostic Packet

Figure 2.2: Seven Types of XTP Packets.

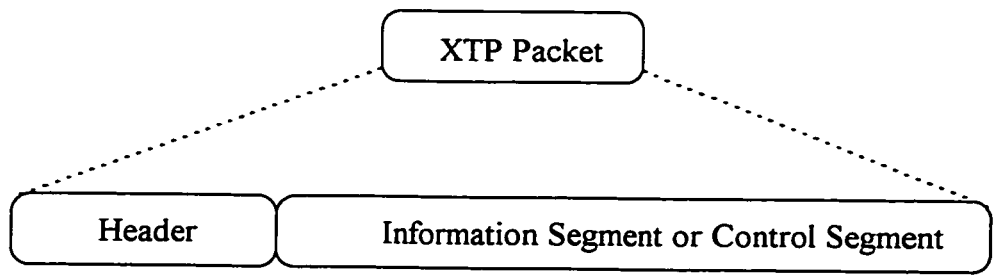


Figure 2.3: XTP packet: 2 major segments.

2.6.1 Header Format

All XTP packets use a fixed header syntax consisting of the following fields: *key*, *cmd* (command), *dlen* (data length), *check* (check sum), *sort* (priority), *sync* (synchronizing number), and *seq* (sequence number). The *key* field steers the packet to the proper destination context. The *cmd* field dictates how the packet to be processed. The *dlen* and *seq* fields identify the packet's contents with respect to the data stream. The *check* and *sync* fields are used to determine the validity of the packet, and *sort* field orders the act of parsing the packet among all contending activities. The XTP header and its fields are shown in figure 2.4 below.

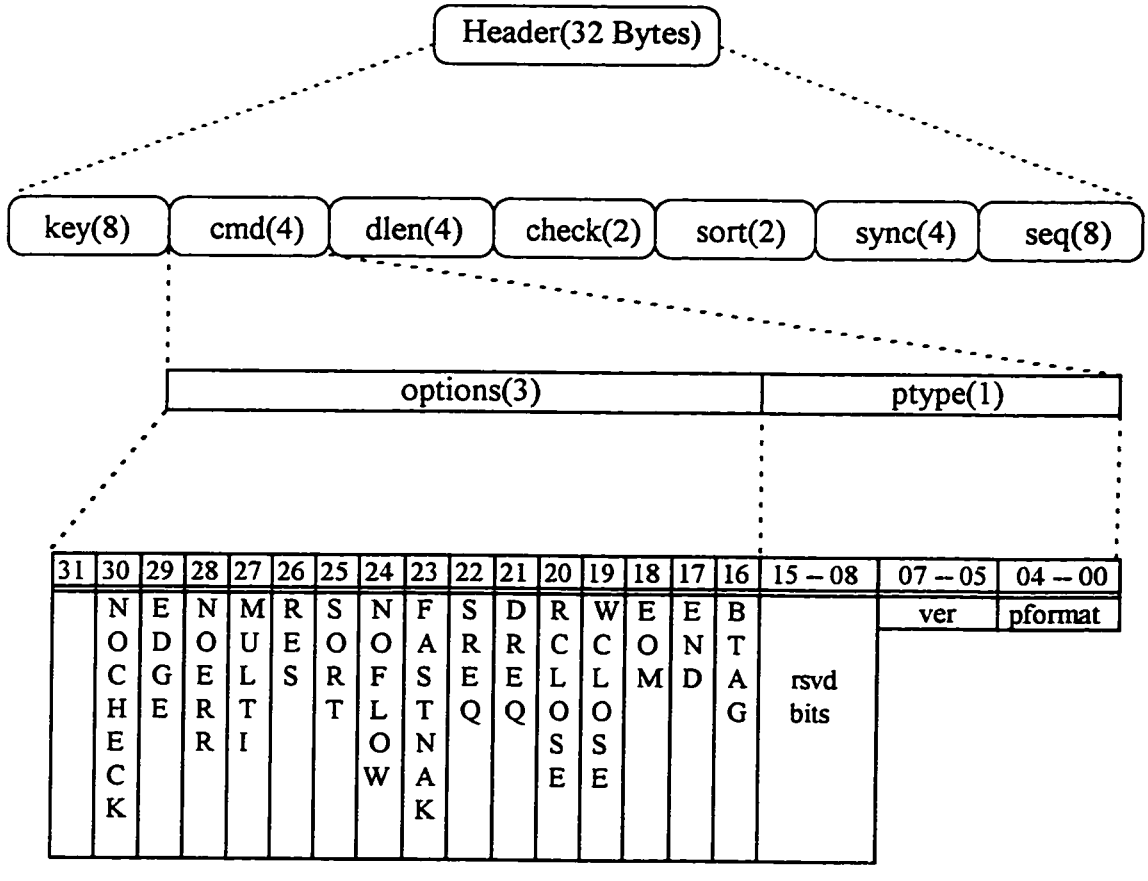


Figure 2.4: XTP Header Fields.

The 32-bit *cmd* field carries the options set for the packet in the *options* field, plus the type of the packet and the version of the protocol that generated the packet in the *ptype* field.

The bitflags in the *options* field select XTP modes and mechanisms. The logic convention is positive: a function is enabled if the corresponding bit is set (value is 1), it is disabled if the bit is cleared (value is zero). Zero or more bits may be set in the header. In the following, we briefly discuss the meaning of some of the bits in the *options* field.

NOCHECK
When set, this bit indicates that the checksum is calculated over the header field only, and the rest of the packet is not summed. When cleared, the checksum is calculated over the whole packet.

NOERR
When set, this mode bit informs the receiver that the sender will not retransmit data, and directs the receiver to disable error correction processing. This is called no error mode.

MULTI
When set, this bit indicates use of multicast mode. The value of this bit must be the same in all packets over the life time of the association.

RES
When set, this bit enables reservation mode. By setting this bit, the sender indicates to the receiver that the *alloc* values provided by the receiver in its control packets (section 2.6.3.1) must represent actual client buffer space available, not XTP internal buffer space. The purpose of reservation mode is to avoid overflowing XTP buffers during bulk transfer.

SORT
When set, this bit indicates that the value in the *sort* field of the header should be interpreted and used for sorting/prioritizing the packet.

NOFLOW

When set, this mode bit indicates that the sender does not observe flow control restrictions (section 2.13). Specifically, the allocation limit imposed by the receiver (in the *alloc* filed of control packets) does not constrain the sender.

FASTNAK

This bit indicates that the receiver should provide aggressive error notification. If a receiver detects an out-of-order packet and the FASTNAK bit is set, then the receiver immediately returns an ECNTL packet (section 2.6.3.2) to the sender to indicate the error. The FASTNAK bit has no effect when the NOERR bit is set.

SREQ

When set, the receiver must respond immediately with a control packet (section 2.6.3.1). The SREQ bit is set by an XTP sender according to its output acknowledgment policy or when responses are needed to recover from errors.

DREQ

When set in a data-bearing packet, the receiver must send a control packet after all enqueued data, up to and including any in this packet, have been delivered to higher-layer applications.

WCLOSE and RCLOSE

These bits are the basis for disconnect handshakes carried out by the close state machines (section 2.10).

EOM

This bit is used to delimit message boundaries in a data stream: when set, this bit denotes the current packet is the last packet of a message.

END

When set, this bit indicates that the sending context is being released. It is used in the last packet of a closing handshake and also when an association is aborted (section 2.10).

2.6.2 Information Segment

The Information Segment encapsulates user and other protocol and diagnostic information. XTP packets containing an Information Segment as their payload segment are called *Information packets*. FIRST, DATA, JOIN, and DIAG packets are the information packets corresponding to the possibilities in figure 2.5. The first two types (FIRST, and DATA) can contain higher layer (user) data. These are referred to as *data-bearing* packets. These other two (JOIN and DIAG) contain transport layer message only.

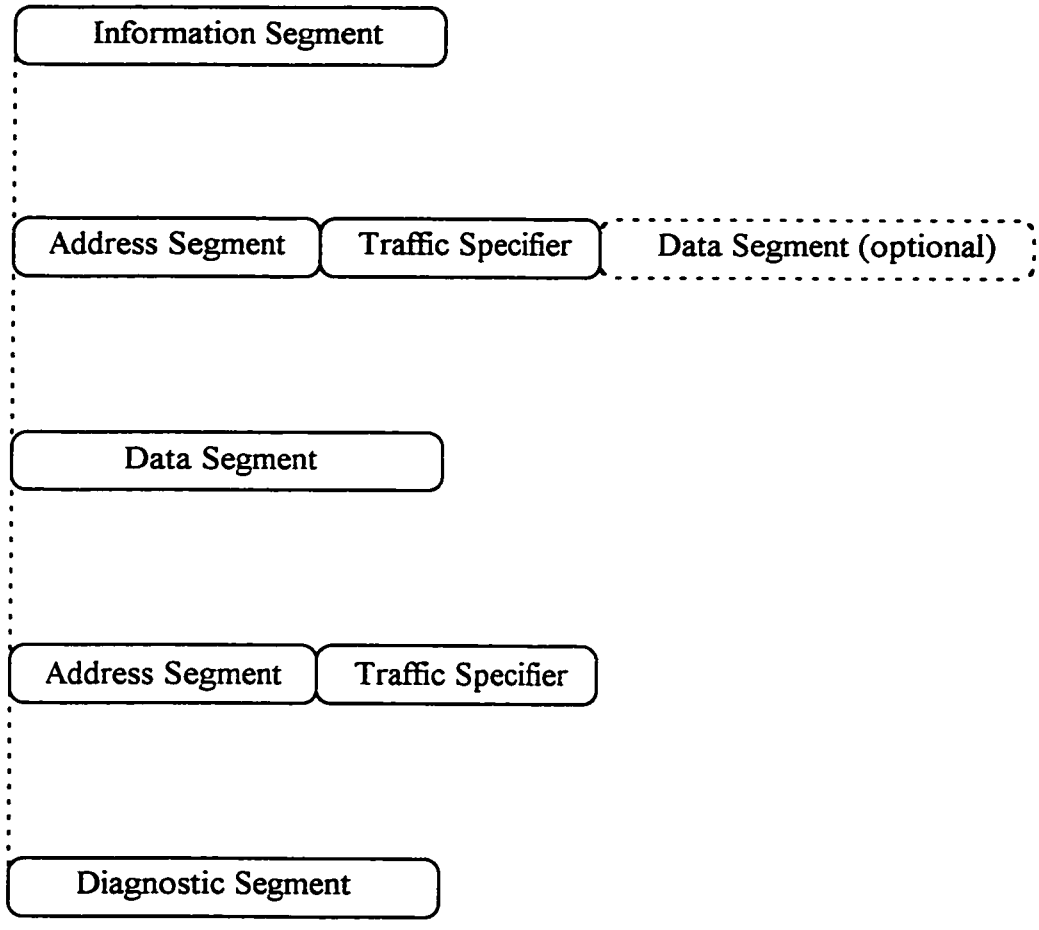


Figure 2.5: XTP packet: Information Segment Structure Overview.

2.6.2.1 FIRST Packet

The syntax for a FIRST packet is shown in figure 2.6. The FIRST packet carries all of the information necessary to find a listening context at a destination host, and establish an association between that context and the sending context (section 2.10). To effect this, the FIRST packet includes an address specification. The Address Segment of the FIRST packet contains destination and source addressing information. The *address* field holds the addresses for the source and destination endpoints. The *aformat* field specifies the format of the address, and the *alen* field specifies the segment's total length.

The *traffic* specifier segment contains the traffic information fields. These fields are used to negotiate traffic shaping information. The *service* field indicates the type of the traffic expected for the association. The traffic shaping information is held in the *traffic* field, whose format is determined by the *tformat* field, and whose length is given by the *tlen* field.

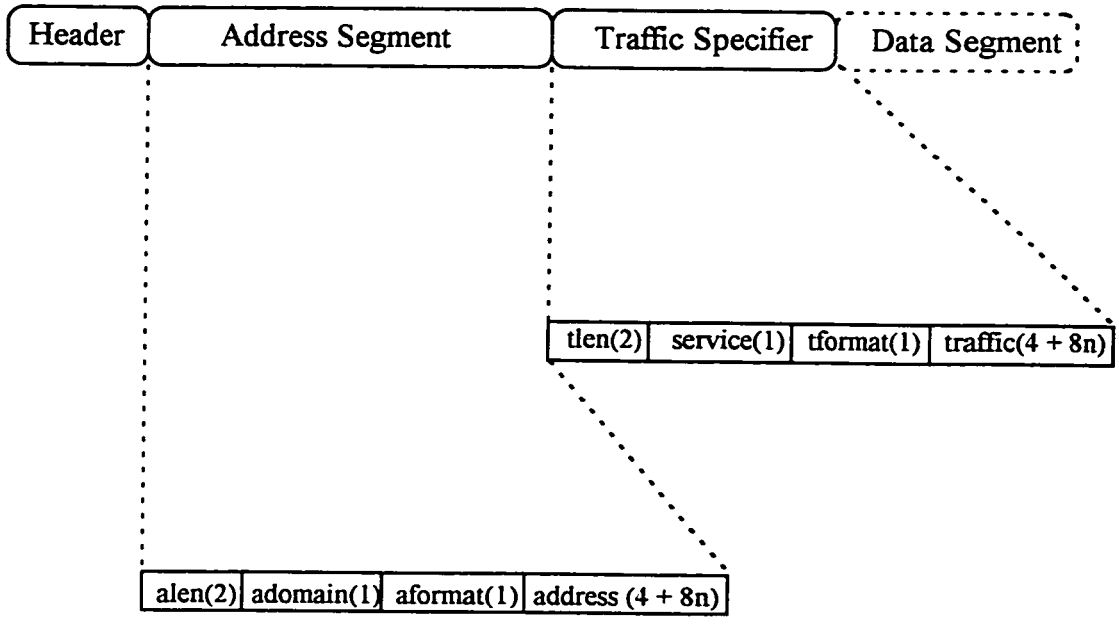


Figure 2.6: FIRST Packet Syntax.

2.6.2.2 DATA Packet

The syntax for the DATA packet is shown in figure 2.7. After establishment of the association, subsequent data transfers in both directions use the DATA packet. The *seq* field (figure 2.4) in the Header indicates the beginning sequence number for the data contained in the Data Segment. The first 8 bytes of a Data Segment may be marked, or "tagged", for special use by higher application. The *btag* field is opaque to XTP, meaning that XTP transmits the contents of the *btag* field but does not look inside or interpret it.

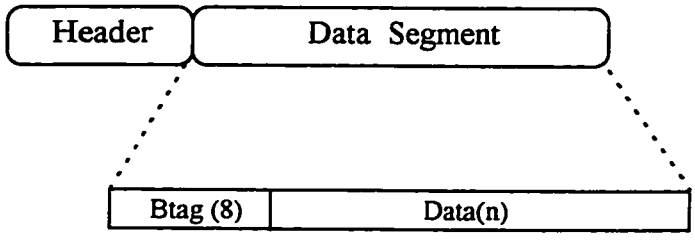


Figure 2.7: DATA Packet Syntax.

2.6.2.3 JOIN Packet

The JOIN packet are used to join an in-progess **multicast** conversation. A JOIN packet has the syntax shown in figure 2.8.

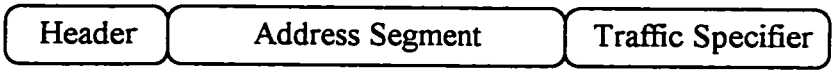


Figure 2.8: JOIN Packet Syntax.

The Address Segment, and the Traffic Specifier Segment of the JOIN packet have the same formats as those in FIRST packet (section 2.6.2.1).

2.6.2.4 DIAG Packet

DIAG packets are used to report pathological conditions that are either fatal or which require corrective action. The format for a DIAG packet is shown in figure 2.9. The *code* field indicates the major error category, and the *val* field modifies that category with more specific information. The *message* field is not parsed by XTP, but may be written to a log file or given to the user.

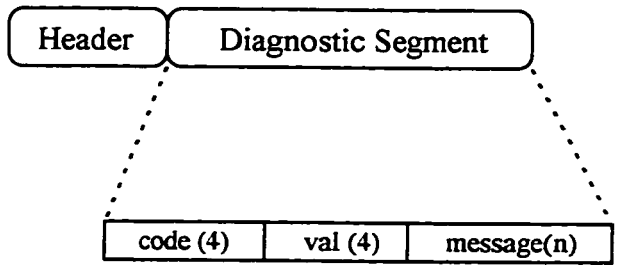


Figure 2.9: DIAG Packet Syntax.

2.6.3 Control Segment

A Control Segment reports the state of the context that sent it. XTP packet contains a Control Segment as their payloads are refered to as *control* packets. Control packets are used to exchange state information between XTP endpoints.

The Control Segment is included in CNTL, ECNTL, and TCNTL packets. The three packet types correspond to the three possibilities for the Control Segment.

2.6.3.1 CNTL Packet

The format of the CNTL packet is shown in figure 2.10. The three fields in the Common Control Segment represent the state information for which a control packet is most

commonly needed. The fields in this segment are common to all three control packet types. The first two fields, *rseq*, and *alloc*, are flow control parameters and together define the flow control window (section 2.13). The *rseq* field holds the highest sequence number for data received without gaps.

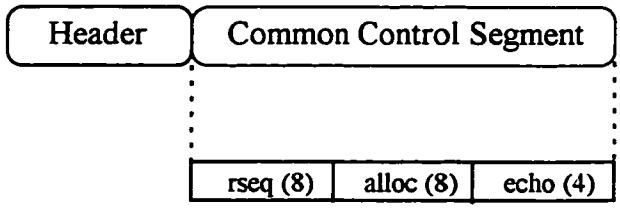


Figure 2.10: CNTL Packet Syntax.

2.6.3.2 ECNTL Packet

The format of the ECNTL packet is shown in figure 2.11. The Error Control segment includes all the fields of the Common Control Segment with two additional fields, *nspan*, and *spans*.

The *nspan* field indicates the number of pairs of sequence numbers held in the *spans* fileld. A pair of sequence numbers, or span, indicate the data that have been correctly received. The gap information, which can be determined from the *spans* field, allows this packet's transmitter to selectively retransmit only the missing bytes of data (section 2.7).

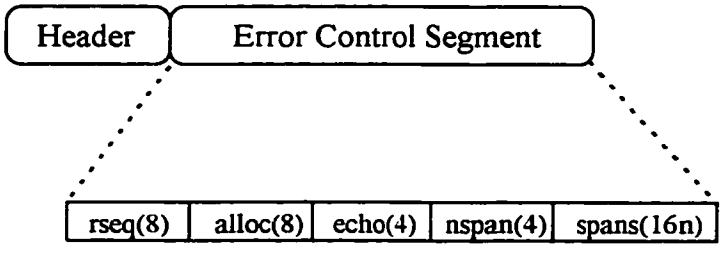


Figure 2.11: ECNTL Packet Syntax.

2.6.3.3 TCNTL Packet

The format of the TCNTL packet is shown in figure 2.12. The TCNTL packet is used to negotiate a traffic specification, which usually happens at or near the beginning of the association. The Traffic Control Segment includes all the fields of the Common Control Segment with two additional fields and a Traffic Specifier. The two additional fields are *rsvd* (reserved) and *xkey* (the exchange key field). The Traffic Specifier contains traffic shaping information, and its format was briefly described in 2.6.2.1 (the FIRST Packet).

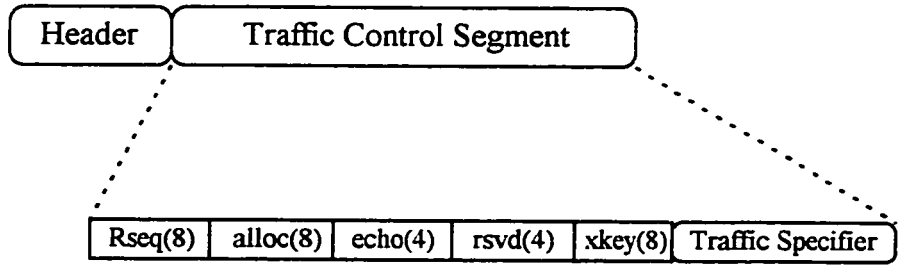


Figure 2.12: TCNTL Packet Syntax.

2.7 XTP Error Control

Error Control in XTP is based on the exchange of information regarding lost or damaged data and the retransmission of these data.

The receiver detects missing packets by checking its incoming data stream for gaps in the sequence number. The receiver records the missing data by keeping the sequence number of the first missing byte, and optionally by keeping a list of spans of correctly received data. When a control packet is sent, the receiver's context checks to see if any data are missing. If there are no data missing, then a CNTL packet is used. This packet acknowledges the receipt of all data whose sequence numbers are less than the value in the *rseq* field (section 2.6.3.1). If missing data have been detected, an ECNTL packet is sent. In addition to using the *rseq* field in this packet to acknowledge data in the same manner as in

the CNTL packet, the *nspan*, and the *spans* fields (section 2.6.3.2) are used to selectively acknowledge spans of data received.

The example in figure 2.13 illustrates the construction of a *spans* field. For an incoming data stream of eleven 100-byte packets with *seq* fields of 0, 100, 200, 300, etc. The receiver only sees packets with *seq* field values 0, 100, 200, 300, 600, 700, 900, 1000. Since bytes 0 to 399 are a contiguous sequence, the receiver fills the *rseq* field of the ECNTL packet with 400. The receiver then describes all other intact spans using the *spans* field. Here, two spans exist, so the value in the *nspan* field would be two and the pairs within *spans* field are [600,800] followed by [900,1100]. This combination of *rseq*, *nspan*, and *spans* makes it possible for the data stream's sender to calculate that sequence numbers 400 through 599, and 800 through 899 should be retransmitted.

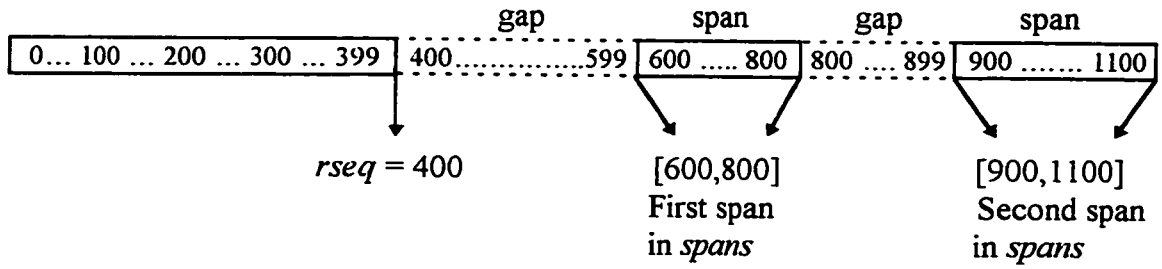


Figure 2.13: Spans in a Data Stream.

Receipt of an ECNTL packet implies that some data have been lost. The *rseq*, *nspan*, *spans* fields specify what data are lost. The transmitter may retransmit data whose sequence numbers start at *rseq* and continue to the highest sequence number sent by the transmitter. This is go-back-N retransmission. Alternatively, the transmitter may selectively retransmit only the data specified as missing. In this case, the transmitter retransmits data starting at *rseq* (400) and continuing up to, but not including, the first value of the first *spans* pair [600,800]. The next piece of retransmitted data is from the second value (800) of the first *spans* pair, and continuing up to, but not including, the first value of the second *spans* pair

[900, 1100]. Figure 2.14 can be used to show the possible packets exchanged between the sender and receiver of the example in figure 2.13.

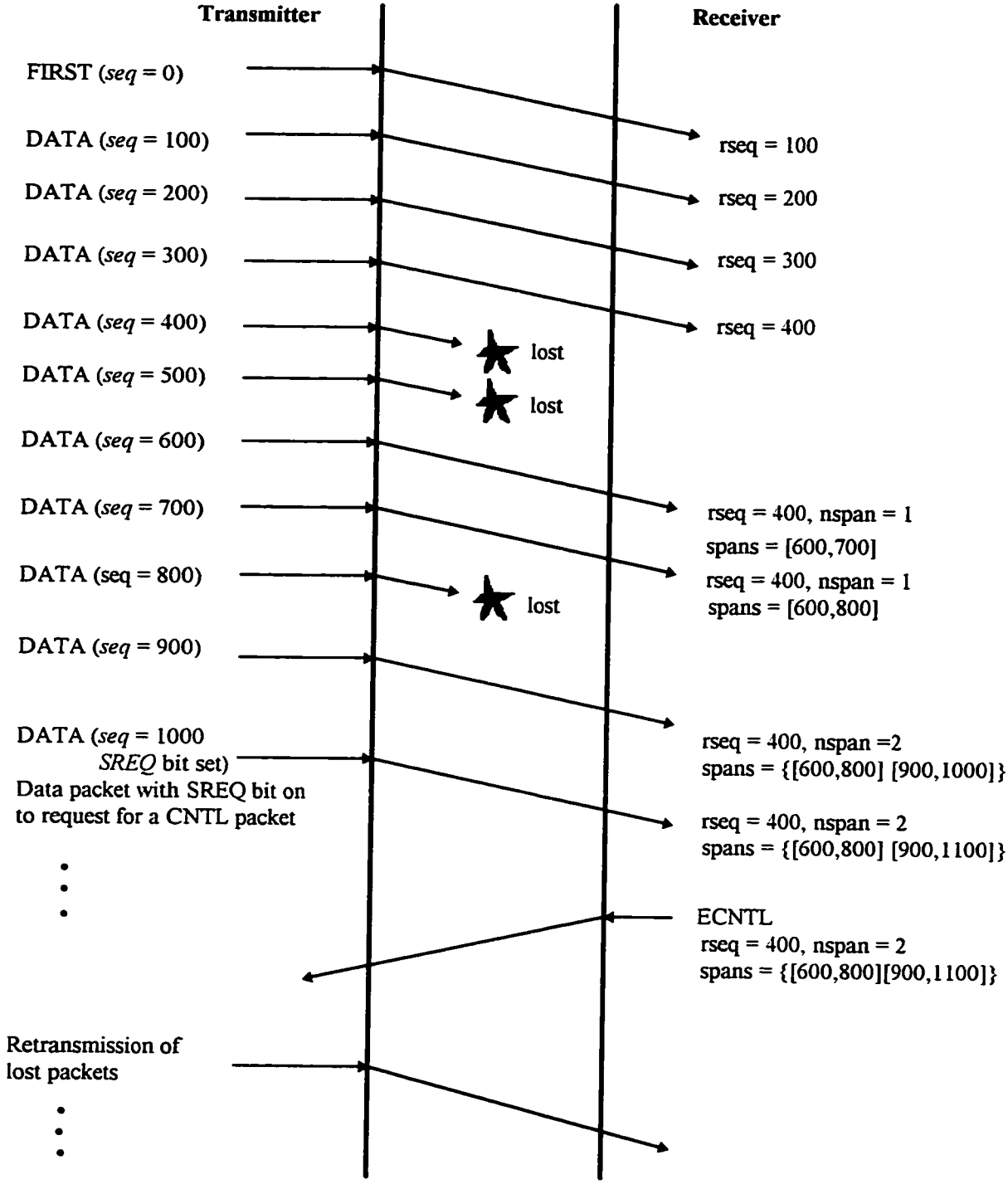


Figure 2.14: Possible packets exchanged for example in figure 2.13.

A Transmitter requests the status of the receiver's incoming data stream by setting the *SREQ* (as shown in figure 2.14) bit in an outgoing packet's header. The *SREQ* bit indicates to the receiver that the status must be reported immediately to the transmitter.

2.8 Status Reports

Status reports in XTP are requested by setting SREQ (or DREQ) bit in an outgoing packet. If SREQ bit is set in an outgoing FIRST, or DATA packet, the CNTL packet response is returned immediately, recording receiver status at the time of the arrival of the requesting packet. If DREQ is attached to a FIRST, or DATA packet, the CNTL packet response is sent once the data have been delivered to the destination host, thus providing more useful information to the sender, including the updated values *alloc* and *rseq*.

SREQ/DSEQ-initiated status reports are controlled by the sender. No policy for setting SREQ or DREQ is defined in XTP, but a timer (Wtimer) (section 2.11) must be started when a packet is sent with SREQ bit set. If WTIMER expires, a new request for the status report is issued, and XTP enters a *synchronizing handshake* (section 2.12), when all further data transmissions are halted until the correct status is received.

2.9 Retransmission Strategy

Data packet retransmissions in XTP are triggered by the arrival of status reports (ECNTL packets with non zero *npans*) showing missing data. As a result, XTP will never retransmit a packet without a positive indication that it has not been received.

2.10 Connection set-up, ending and management

XTP, unlike conventional protocols, uses a single packet to establish a virtual circuit connection. Figure 2.15 shows the establishment of an association. The context B at one end-system is initially in a quiescent state. A user awaiting the start of an association requests that context be placed in the listening state (1). At the other end-system, a user

requests the establishment of an association (2). The context handling this user moves from a quiescent state to an active state, where it constructs a FIRST packet with explicit addressing and service information obtained from the user. The FIRST packet is sent via the underlying delivery service.

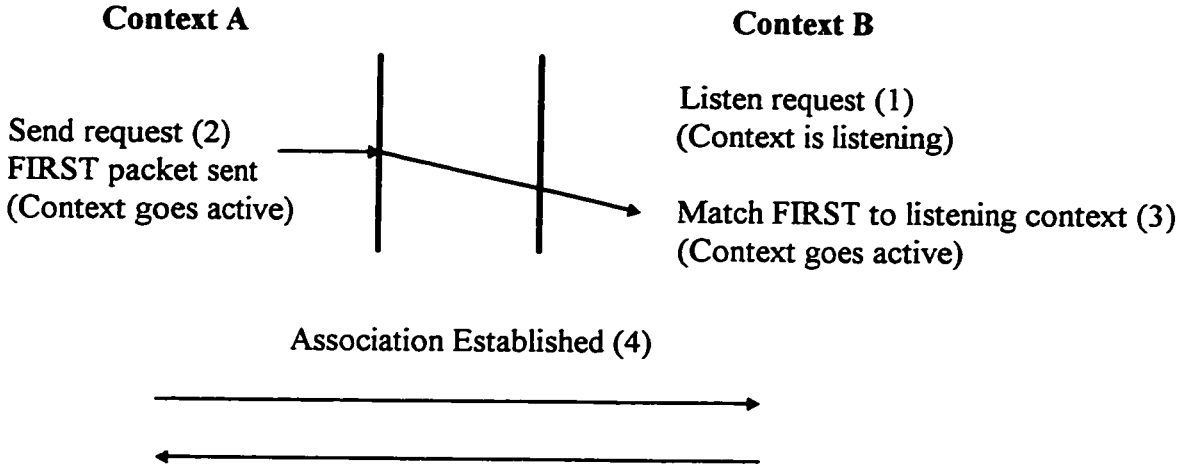


Figure 2.15: Association Establishment.

When the FIRST packet is received by the destination end-system, the address and service information in the FIRST packet is compared against all listening contexts. If a match is found, the listening context is moving to the active state (3). From this point forward, an association is established, and communication can be completely symmetric since there are two data streams, one in each direction, in an association (4). Also, no other packet during the lifetime of the association will carry explicit addressing information. Rather, a unique "key" is carried in each packet, which allows the packet to be mapped to the appropriate context.

In XTP, the closure of an association results in the release of each data stream (as in TCP). Moreover, each data stream can be ended through a graceful close or a forced termination. Three header bits (figure 2.4): WCLOSE, RCLOSE, END are involved in the closure features. A fully graceful independent close is shown in figure 2.16. Context A

initiates the close of its outgoing data stream with a packet with WCLOSE bit set. When all data are accounted for according to the error control parameters for the A-to-B data stream, Context B responds with a packet with RCLOSE set. At this point, the A-to-B data stream is graceful closed, but the B-to-A data stream remains open. Later, context B initiates the close of its outgoing data stream by setting the WCLOSE in an outgoing packet. Context A responds accordingly with an RCLOSE. At this point, Context B sends a packet with the END bit set, and the association is terminated.

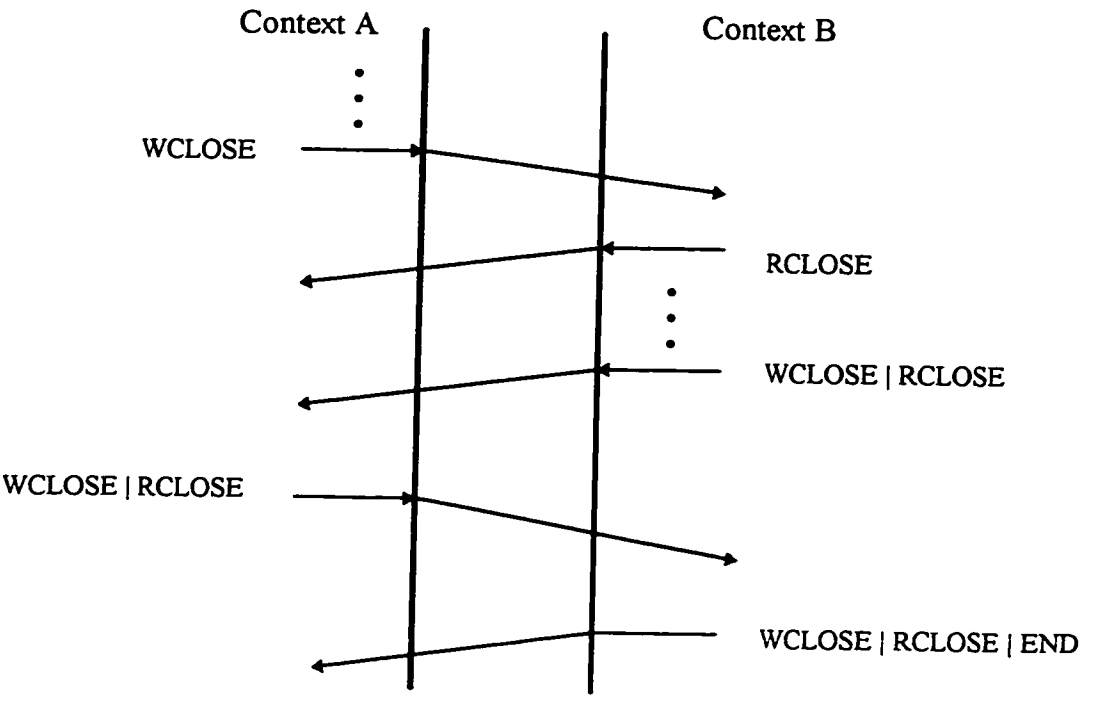


Figure 2.16: Fully Graceful Independent Close.

2.11 Timers

Certain aspects of the XTP control procedures reply on timers to signal when an expected event has failed to occur. Without the use of timers, the protocol risks deadlock. There are four timers in XTP: WTIMER, CTIMER, CTIMEOUT, and RTIMER.

The WTIMER is the timer that guards against the lost of a packet with the SREQ bit set. Whenever a packet is sent with the SREQ bit set, the sender's context also starts the WTIMER, loading it with a smoothed **round-trip time** (section 4.4) estimate. The WTIMER is the amount of time the transmitter will wait for the arrival of the control packet requested. If WTIMER expires before the requested control packet arrives, the context starts the synchronizing handshake, as described in the next section.

The CTIMER is the timer that ensures that the other endpoint of the association is still alive. When an context becomes active, the CTIMER is armed. This is a long duration timer. A count is kept of all the packets that arrive at this context. When the CTIMER expires, the context examines the packet count. If the count is greater than zero, the CTIMER is reloaded, and the packet count is cleared. If the count is zero, the CTIMER is reloaded, and the context enters into a synchronizing handshake as described in the next section.

The CTIMEOUT timer limits the amount of time a synchronizing handshake can continue before the context aborts the association, as described in the next section.

The RTIMER is the rate control timer used to govern the frequency of sending bursts of data. Its use is described in section 2.14.

2.12 Synchronizing Handshake

Whenever a packet is sent with *SREQ* bit set, the transmitter increments a context's variable named *save_sync* by one and pass it into the *sync* field of the packet's header. The context sending the packet with the *SREQ* bit set also starts the WTIMER.

If a control packet arrives at the transmitter before the WTIMER expires, the value in the *echo* field of the control packet is compared with the *save_sync* value. If they are equal, the WTIMER is stopped. If the WTIMER expires, the context enters the synchronizing handshake procedure. The objective of synchronizing handshake procedure is to probe the receiver by sending control packets at exponentially increasing time intervals until a successful handshake occurs or the CTIMEOUT expires or the number of retries exceeds a limit. If either CTIMEOUT expires or the number of retries exceeds a limit, XTP aborts the association. No data-bearing packets are allowed to be sent during a synchronizing handshake, including retransmitted data; retransmission may proceed once the handshake has completed.

2.13 Flow Control

The volume of XTP output is regulated by an end-to-end windowing flow control mechanism. (The rate at which XTP sends packets into the network is regulated by an independent, timer-based mechanism described in next section). XTP's flow control is based on a sliding window of sequence numbers. A sequence number is assigned to each output byte of the data stream, starting with the initialized sequence value.

Two fields in control packets are used in flow control procedures. The value in the *alloc* (allocation) field in a control packet sent to the transmitter indicates the sequence number not to be exceeded by the transmitter. This value represents the upper edge of the flow control window. The value in the *rseq* field in a control packet sent to the transmitter is one greater than the last byte contiguously received. This value serves as the lower edge of the flow control window.

Two options bitflags modify the way flow control is handed. The RES bit in the packet's header sent by the transmitter indicates to the receiver that it should advertise conservative flow control values, specifically, the *alloc* value should reflect only as much buffer space as the user has allocated for the association. This is called *reservation* mode. The NOFLOW bit in a packet's header indicates to the receiver that the transmitter does not wish to adhere to flow control, so flow control in the forward direction will be disabled.

2.14 Rate Control

Rate control governs the producer-consumer relationship between XTP endpoints. Rate control is concerned with how fast packets and their contents can be processed, or consumed, at the receiver. Parameters throttling the rate of production can be fed back to the sender through the rate control parameters of the Traffic Specifier of the TCNTL packet (section 2.6.3.3). If explicit rate control parameters are not available, default *rate* and *burst* parameters must be used.

The output packet rate is regulated by two context variables, **credit** and **burst**, and by the refresh timer called RTIMER. The values for **credit** and **burst** can be calculated from the default parameters *rate*, and *burst*. The *rate* value specifies the maximum data rate in bytes per second. The *burst* value specifies the maximum number of bytes to be sent in a burst of packets. The *rate* value divided by the *burst* value gives the number of **burst**-size transmission per second, or the rate at which the **credit** variable is refreshed. The *burst* value divided by the *rate* value gives the time period of RTIMER (section 2.11).

The **credit** and **burst** variables are initialized with the *burst* value. With each outgoing data-bearing packet, **credit** is decremented by the sizes in bytes of the user data transmitted. Data transmission must cease when **credit** becomes zero or negative. Upon each expiration of RTIMER, the internal variable **credit** is updated with the value **burst** (**credit** is updated approximately *rate/burst* times per second). Output is permitted as long as **credit** is greater

than zero, and is suspended when **credit** becomes zero or negative. The suspended state lasts until **credit** is refreshed at the next RTIMER interval.

2.15 XTP Multicast

Multicast is the capability to provide *Group Communication* (one-to-many), as opposed to Unicast (one-to-one) exchanges. Multicast requires from the underlying service a proper addressing capability, i.e., broadcast or multicast. In XTP, a multicast association is initialized by a FIRST packet with the MULTI bit (in the packet's header) turned on and a multicast address assigned to the address segment. All the multicast packets have the MULTI bit turned on. All the mechanism defined in XTP unicast mode are provided in multicast mode. In particular, flow control and rate control are available. The three error control methods which are used in point-to-point XTP are possible: no error control at all, go-back-n and selective retransmission.

In multicast mode, control packets sent by multicast receivers in response to a received packet with SREQ bit set, are multicast to the group and not only unicast to the sender.

The sender must face a set of control streams. Consequently, at the multicast sender, the management of control packets does not obey the rules of the unicast mode. Multicast mode raises the problem of synchronizing the different control streams. After multicasting a packet with SREQ bit set, the sender must wait for the return of all control packets, then compare the received values so that actions (in terms of error control, flow control, rate control) are taken. The strategy used to deal with that appear to be decisive in the performance of a multicast association.

This thesis simulates XTP in unicast mode, so the XTP multicast operation is not described in more detail in this section. Interested readers are referred to the most recent version of the XTP specification [XTPForum95].

2.16 Conclusion

The basic concepts of XTP were briefly described in the sections above. The influences of the conventional protocols (such as TCP (1977), Delta-t (1978), ISO TP4 (1982), NETBLT (1986), and VMTP (1986), etc.) can be found in [STRAYER92]. It is evident that XTP built much of its functionality on mechanisms and concepts introduced in the above conventional protocols. What makes XTP unique is that it combines these good ideas into a protocol whose mechanisms are as much as possible orthogonal to each other.

3. AN INTRODUCTION TO SMURPH AS A SIMULATOR TOOL

Recent advances in computer and communication systems have resulted in demand for new tools for their analysis. The mathematical modeling techniques have so far proved inadequate in dealing with these systems, and simulation seems to be the only alternative.

This chapter introduces the System for Modeling Unslotted Real-time Phenomena (SMURPH) as the tool which we used to study the performance of XTP with error control. In the following, we introduce the development history, and the basic concepts of SMURPH, to help the reader to understand the simulation design of XTP in the next chapter.

3.1 Development History of SMURPH

SMURPH descends from an earlier package called the *Local Area Network Simulation Facility* (LANSF). LANSF is a software simulation modeling package, which was developed by Dr. Pawel Gburzynski and Dr. Piotr Rudnicky at the University of Alberta in 1988. LANSF was originally developed for investigating Medium Access Control (MAC) level protocols. The idea of LANSF was to provide a friendly environment for describing networks and protocols in such a way that the description would look as an abstract implementation. The environment would take care of accurate modeling of the physical elements constituting the networks (e.g., communication channels).

Although LANSF had been built with intention of using it in research on communication protocols, it can also be used to model other communication bound systems. By now, LANSF has been distributed to over a hundred academic and industrial institutions.

However, one disadvantage of LANSF was its low flexibility in describing compound data structures. Furthermore, the code written in C for simulation programs using LANSF was difficult to maintain and modify. The idea of re-implementing LANSF in C++ was born

from the comments of the Networking Group of the Lockheed Space and Missile Company.

SMURPH is the new version of LANSF programmed in C++. It is not a single interpreter for a variety of networks and protocols, but it configures itself into a stand-alone modeling program for each particular application.

3.2 Programming a protocol in SMURPH

SMURPH package provides a set of predefined types, objects, functions, and macros operations. Some of the predefined data structures of SMURPH are:

- configuration elements of a distributed communication system (e.g., stations, channels) and
- information passed among stations (e.g., messages and packets).

In addition, SMURPH provides:

- functions for accessing the communication system (e.g., start or terminate packet transmission) and

- tools for process handling (process creation, termination, or synchronization through the use of signals).

In a protocol program using SMURPH, the user-supplied types and data definitions together with the protocol processes are contained in C++ files. They look like a regular program in C++ which has access to the SMURPH libraries of types, data structures, functions and macros operations. A special support program is provided whose purpose is to merge the user protocol files with the SMURPH libraries and create a stand alone version of the simulator. With this approach, the user gets the full power of C++ combined with the

power of a realistic, emulated environment for programming, executing, and monitoring communication protocols.

3.3 SMURPH Types

By a SMURPH type we mean a compound, predefined, user-visible type declared as a class with some standard properties. Figure 3.1 presents the hierarchy of built-in basic SMURPH types. We assume that all of them are derived from a common ancestor called *class* which reflects the fact that they are all compound types, and that the words "class" and "type" can be used interchangeably to denote the same concept.

From figure 3.1, all objects exhibiting dynamic behavior belong to type *Object*, which is an internal type, not visible directly to the user. This type declares a number of standard attributes and methods that each *Object* must have.

Timer and *Client* stand for specific objects rather than types. These objects represent some important elements of the protocol environment and are static, in the sense that they exist throughout the entire execution of the simulator. Each of them occurs as exactly one copy. Therefore, the actual types of the above objects are uninteresting and are hidden from the user. Other *Objects* may exist in multiple copies; some of them may be dynamically created and destroyed during the simulation run.

Generally, all objects belonging to the subtype *AI* (for activity interpreters) are models of some entities belonging to the protocol environment. They are responsible for modeling the flow of the time.

The objects shown in figure 3.1 are described in more detail in the following sections of this chapter.

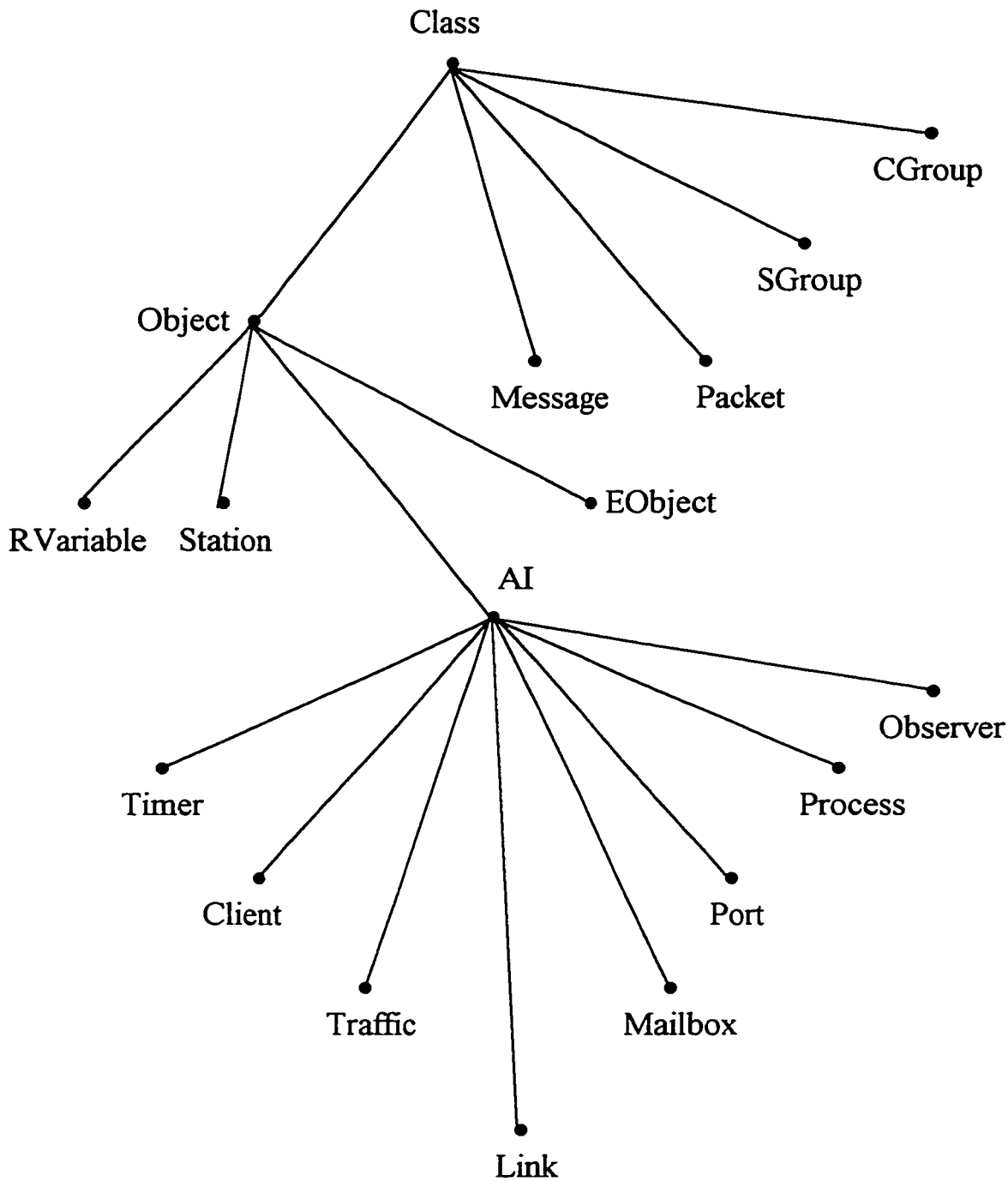


Figure 3.1: The hierarchy of user-visible compound types.

3.4 Defining Network Geometry

The geometry of the modeled network, although static from the viewpoint of the protocol program, is defined dynamically by explicit object creation and calls to some functions and methods. As perceived by SMURPH, the topology of modeled network is defined as interconnection of stations, ports, and links.

Stations are objects of type *Station* (Figure 3.1). A station typically attached to a link (or a number of links) via ports. A port (an object of type *Port*) represents a connection of one station to a link. A link (an object of type *Link*) models a simple communication medium, e.g., a fiber optic carrier, a coaxial cable, or a radio channel. In this way, stations are similar to modules, links are similar to channels, and ports are similar to interaction points in ESTELLE.

In SMURPH, Stations, Ports, and Links are assigned numeric identifiers which generally correspond to the order in which the particular objects have been created.

3.5 Processes

The dynamic part of the user specification, i.e., the protocol program, has the form of a collection of cooperating processes which are executed by the stations. Each process (an object of type *Process*) can be viewed as an *interrupt handler*: its processing cycle consists of being awakened by some event, responding to the event by performing some protocol-related activities, and going back to sleep to await the occurrence of another event.

Each process could be seen as a finite state machine (FSM) in which each state is written in the form of a wait function call.

3.6 Activity Interpreters

Operations of protocol processes are driven by events. These events are generated by objects called *AI* (*Activity Interpreter*). The operation of an AI consists in transforming activities into events. This is what we mean by *interpreting activities*.

A typical example of an AI is a Mailbox (an object of type *Mailbox*), which provides a means for inter-process communication: each process is capable of triggering events that can be perceived by other processes. Mailbox provides a systematic way to synchronize processes, especially those running at the same station.

3.7 Defining traffic conditions

The traffic in the network consists of messages that arrive from "outside" to be processed by stations. By processing a message, we mean splitting it into one or more packets and transmitting it over the network to the destination (a specific station).

The traffic distribution is described by the collections of the so-called *traffic patterns* which are objects of type *Traffic*.

Each traffic pattern is associated with specific message and package types. Messages and packets belonging to different traffic patterns may have different, protocol dependent structures.

SMURPH provides two standard base types *Message* and *Packet* that can be used to define protocol-specific message and packet types. The type *Message* also contains an array, which will hold the additional protocol-specific attributes. Most of the attributes of *Packet* are inherited from *Message*.

3.8 Time

Time in SMURPH 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. SMURPH defines the type *TIME*, whose precision can be specified by the user. Since *TIME* can have a large precision, SMURPH provides standard functions to perform multi-precision arithmetic on variables of that type.

Besides the ITU unit, SMURPH 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 XTP simulation, one virtual second is equal to ten million ITUs, because the ETHERNET transmission rate is 10 million bits per second.

3.9 Performance measures

One important objective of modeling networks and protocols in SMURPH is to investigate their performance. Some performance measures are calculated automatically by the SMURPH package. The user can easily collect additional statistical data which augment or replace the standard measurements.

3.10 SMURPH Debugger

SMURPH offers a dynamic status display (DsD) program where snapshot states of the stations and links can be displayed while the simulator is running. The user has some control over what information is displayed on various windows.

SMURPH also provides some standard functions to let the user display his own variables, messages, and information relating to some objects in the protocol program.

3.11 Discussion

Overall, SMURPH provides minimal features for complex protocol performance simulations. As a result, the implementers of a protocol simulation have to design their protocol program in great detail, in order to obtain a proper simulation results. However, since the coding is now done in the object oriented language C++ , it is easier to maintain and modify a protocol program implemented in SMURPH than the similar one implemented in the old version LANSF.

4. SIMULATION OF XTP USING SMURPH

Our simulation of XTP uses an Ethernet environment within which two stations set up virtual circuit connections to each other, and transfer data using XTP. The work of this thesis is based on the works of Chen [Chen89] and Chung Kam Chung [Chung93]. Chen simulated the revision 3.3 of XTP using LANSF. Chen wanted to find out the maximum throughput of XTP in a no-error environment. Chung Kam Chung extended Chen's work by modeling the error control mechanism of XTP (revision 3.6) in LANSF.

The work of this thesis follows the assumptions that were mentioned in [Chen89] and [Chung93]:

- The datalink layer service uses IEEE 802.2 class I service, which is a connectionless service.
- The limited capacity of the buffers and queues is not simulated.
- Message fragmentation time is not simulated.

4.1 Conceptual Model of XTP Simulator in SMURPH

Figure 4.1 shows the conceptual model of a station architecture of our simulator. There are four simulated chips:

- The main processor chip simulates the single processor that will handle the different tasks of the protocol sequentially.
- The timer chip handles all timer operations.
- The rate control chip handles XTP rate control mechanism.
- The Ethernet chip performs CSMA/CD operations.

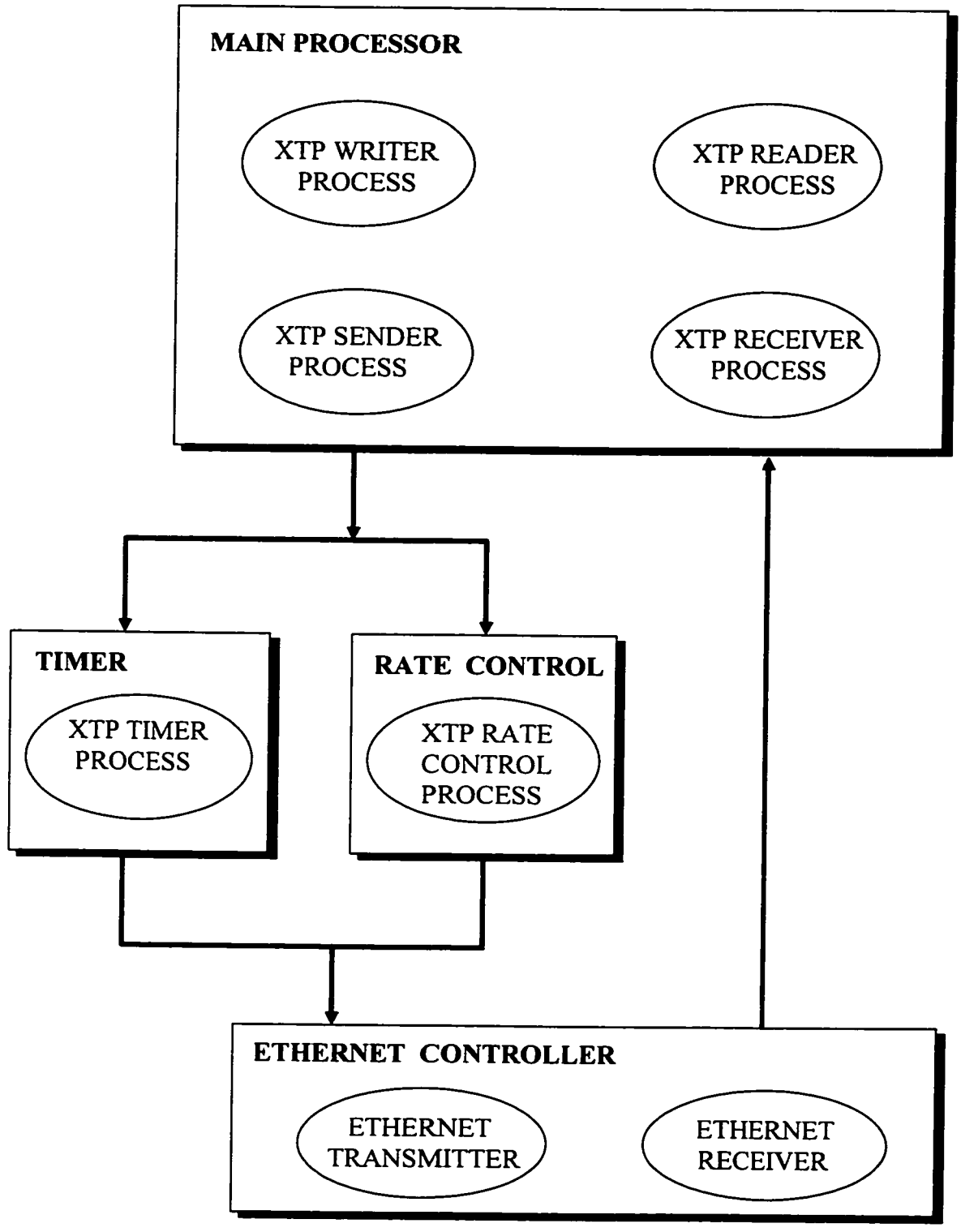


Figure 4.1: Simulated Station Architecture [Chung93].

4.2 XTP Processes in SMURPH

In our simulation, eleven main processes will be used at a **station** to implement XTP. The processes: XTP_Initialization, XTP_Writer, XTP_Sender, Serializer, XTP_RateControl, XTP_Timer, and Ethernet_Transmitter are used for preparing a packet to be sent to the other station. The processes: Ethernet_Receiver, XTP_Receiver, XTP_Reader, and Consumer are used for receiving a packet sent from the other station.

The process XTP_Initialization sets up the virtual circuit environment. The process XTP_Writer simulates the generation of messages arriving at a station. The process XTP_Sender breaks messages into packets, and performs core operations of XTP. The process XTP_RateControl simulates the rate control mechanism in XTP. The process XTP_Timer sets up the timers used, and the process Ethernet_Transmitter is used to transmit a packet on the underlying delivery service.

The process Ethernet_Receiver is used to receive a packet arriving at a station. The processes XTP_Receiver and XTP_Reader are used to implement the error control mechanism of XTP. The process Consumer simulates an XTP user at the receiving end.

The process Serializer simulates the main processor which runs the processes XTP_Sender, XTP_Receiver, and XTP_Reader.

Figure 4.2 shows the flow of a packet through these processes at a station.

SMURPH gives the processes the ability of communicating to each other using signals. SMURPH is deficient in the sense that additional parameters to be passed between processes will require the user to define more complicated data structures. In our simulation, we defined generic queues, which can be used to store parameters passing between processes.

In the following sections, we will describe in more detail the above processes, the signals passed between them, and the generic queues used by these processes.

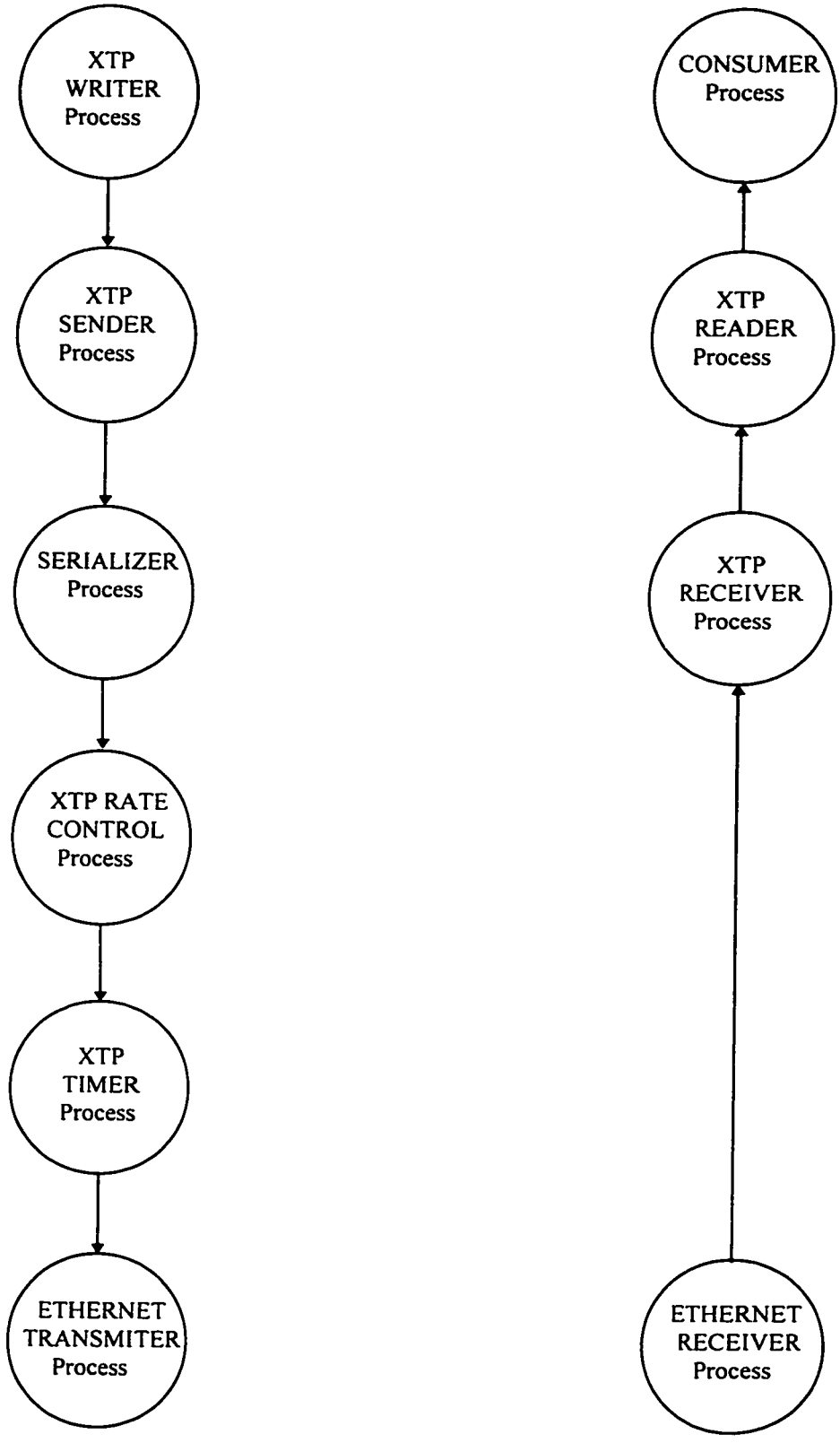


Figure 4.2: Data Flow Diagram at a **station** of the simulation program.

4.2.1 The Initialization Process.

The initialization process at a **station** of the XTP simulator sets up the virtual circuit simulation environment. It creates a context for the station with a receiving station. For example, if our XTP simulator has three stations (identified in SMURPH by station number 1, 2, 3), then the initialization process of station 1 will create two contexts: one context is for the receiving station 2, and the other context is for the receiving station 3. Similarly, the initialization process of station 2 will create two contexts: one context is for the receiving station 1, and the other context is for the receiving station 3. Similarly for the initialization process at the sending station 3.

The context at a station contains the state information of **both the outgoing data stream and the incoming data stream**. From the example above, at the station 1, the context associated with station 2 may contain the following parameters: the sequence number for the next packet (of the outgoing data stream) to be sent to the station 2 by the sender station 1; the WTIMER, RTIMER (Section 2.11); the values of *alloc* (allocation) and *rseq* (section 2.13), etc. The value of the *alloc* field is a do-not-exceed sequence number for the incoming data stream, calculated by the receiver station 1, that limits the amount of data that the sender station 2 may transmit. The value of the *rseq* field is one past the highest consecutive sequence number received without error at station 1. The station 2 can receive values of *alloc* and *rseq* from a returning control packet.

4.2.2 The Writer Process

The process XTP_Writer at a **station** is intended to set up an interface for a user to create an active open virtual circuit and then send data. However, the writer process in our simulation serves merely as a triggering process to start the chain reaction of the entire simulation.

The process XTP_Writer has the following functions: (a) It waits for a packet arrival signal which is generated by the SMURPH traffic scheduler process, (b) When a new packet has arrived, it will send a signal to the process XTP_Sender (which is discussed in the next section), (c) it returns to (a). Packets arriving at a station are saved on a SMURPH **internal packet queue**. Figure 4.3 shows the Finite State Machine (FSM) of the process XTP_Writer.

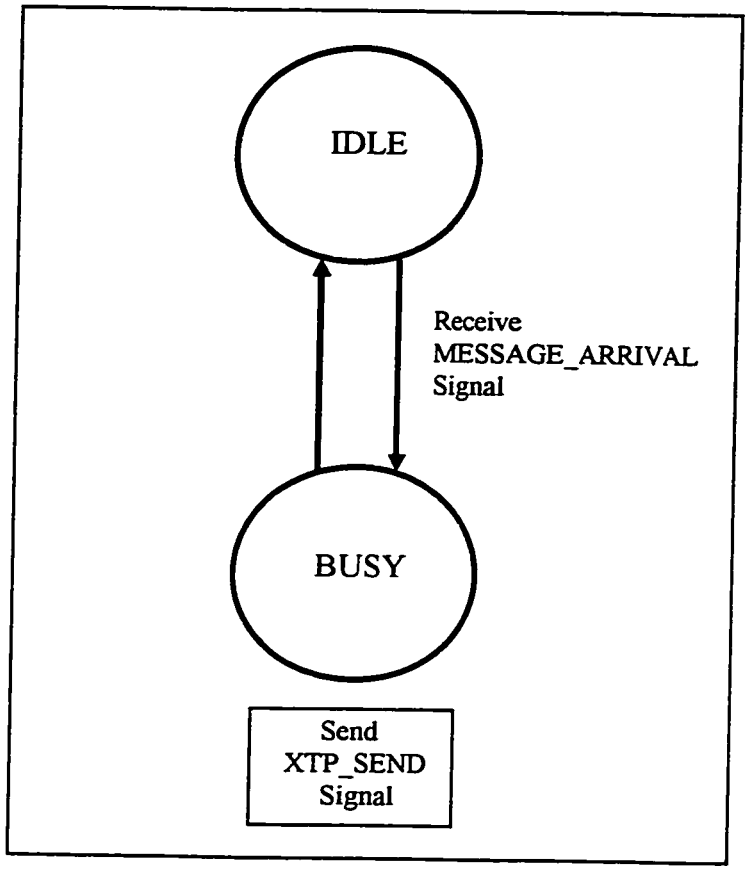


Figure 4.3: Finite State Machine for the process XTP_Writer.

4.2.3 The Sender Process

The process XTP_Sender performs the core operations of the protocol including flow control and rate control.

When the process XTP_Sender receives the signal XTP_SEND (from the process XTP_Writer), it dequeues an arriving packet from the SMURPH internal packet queue (mentioned in section 4.2.2). It then saves the packet in the appropriate context. From the example in section 4.2.1 above, when a packet (to be sent to the station 2) received by the station 1, it will be saved on a waiting queue in the associated context (for the receiving station 2) created in the process XTP_Initilization (section 4.2.1). A context with waiting packets to be sent is saved on a queue called WPX_L. Listing 1 shows the pseudo-code for the process XTP_Sender.

The pseudo-code in listing 1 shows that the process XTP_Sender starts when it receives the signal XTP_SEND. It then checks if there is any context on the WPX_L queue with waiting packets to be sent. If one is found, each packet dequeued from the waiting queue is given a sequence number, packet type, etc., and then it will be enqueued on a queue called PSR_L. The PSR_L queue will be used later by the process Serializer. If a context reaches the flow control limit (section 2.13) or the rate control limit (section 2.14), the context will be blocked, no more packet, are dequeued from the waiting queue. A blocked context can be unlocked when the XTP_SENDER process receives the unblocked signals from other processes.

```
PROCESS XTP_SENDER

    Input signals:
      • XTP_SEND from the process XTP_Writer
      • FLOW_UNBLOCKED from the process XTP_Receiver
      • CREDIT_UNBLOCKED from the process XTP_Credit

    Output signal
      • PROCESSOR_START to the process SERIALIZER

BEGIN

 Start:

    wait for XTP_SEND signal arriving from the process XTP_WRITER.

    If there is a context which is not blocked and with packets to be sent then
       continue at label is_wait
    EndIf

    If all the contexts of the station are blocked then
        continue at Start
    EndIf

    Receive the arriving packet from the SMURPH internal packet queue.
    Save the packet on the waiting queue of the associated context.

Is_wait:

    set context to BUSY
    REPEAT
        'send' a waiting packet (simulate delay time to copy the packet to the buffer of the station;
                                 enqueue the packet, and the associated context to the PSR_L queue)
    UNTIL alloc or credit limit is reached

    Continue at Start

ENDPROCESS
```

Listing 1: pseudo-code for the process XTP_Sender.

The Finite State Machine of the process XTP_SENDER is shown in figure 4.4.

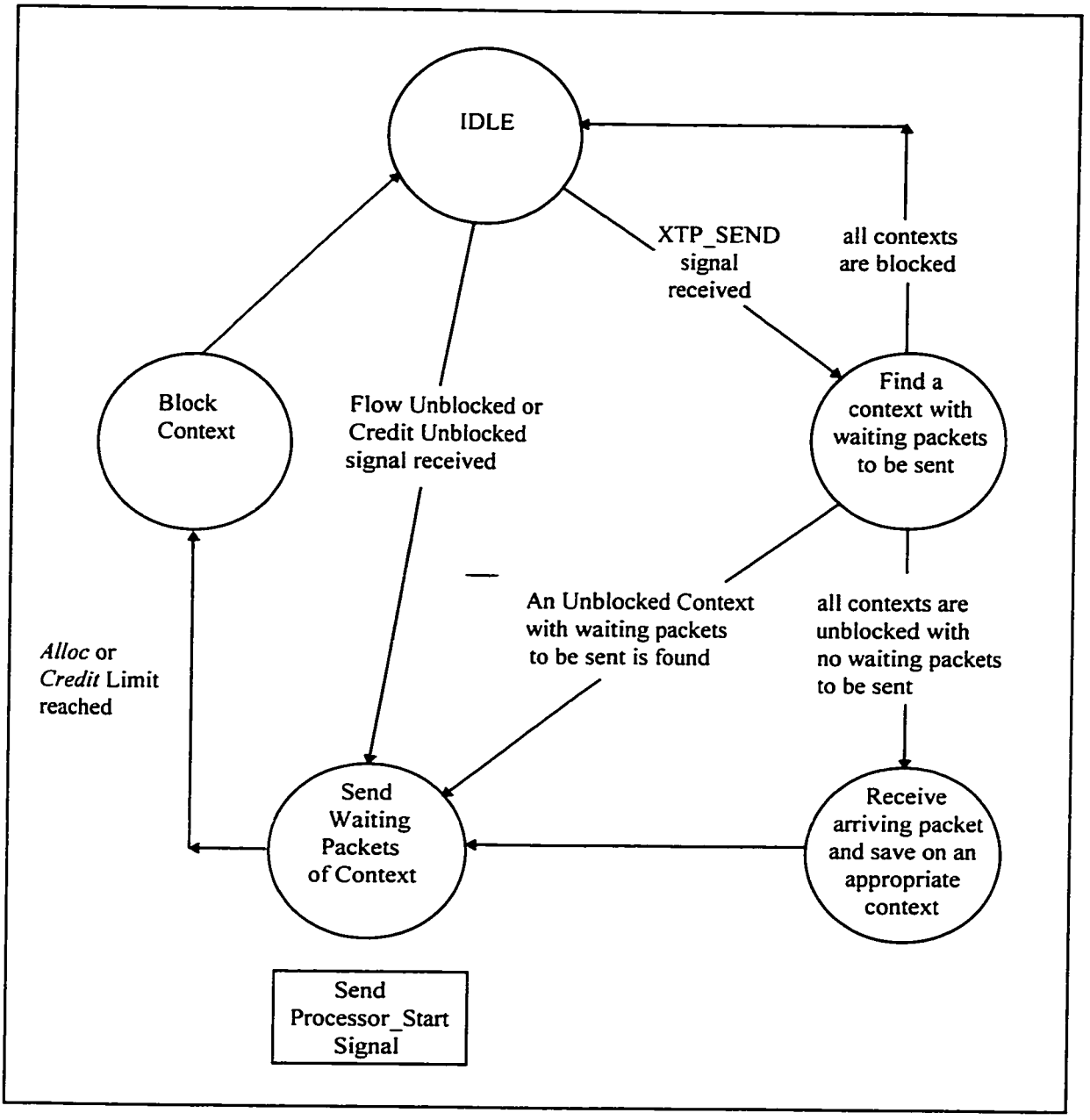


Figure 4.4: Finite State Machine of the process XTP_SENDER.

4.2.4 Process Serializer

The purpose of the process Serializer is to simulate the fact that the sender, receiver, and reader processes are running on a single CPU. The process Serializer schedules the three processes in FIFO order. Whenever a process wishes to use the processor, it sends a signal to the process Serializer and puts an event item on the PSR_L queue. When the process Serializer receives the signal from the sending process, it wakes up and checks the contents of the PSR_L queue, and performs the operations according to the event. The operations are to signal a waiting process or to dequeue the event from the PSR_L queue and to put it on the RCNTL_L queue that is going to be used by the process XTP_RateControl (section 4.2.5). Listing 2 shows the pseudo-code for the process Serializer.

```
PROCESS SERIALIZER
    Input signals:
      • PROCESSOR_START from the XTP_Sender, XTP_Receiver, XTP_Reader processes
   Output signals:
      • COPY_IN_DONE to the process XTP_Sender
      • COPY_OUT_DONE to the process XTP_Reader
      • CHECK_SUM_DONE to the process XTP_Receiver
      • NEW_ENQUEUE to the process XTP_RateControl
BEGIN
Start:
    wait for the PROCESSOR_START signal.

    Dequeue the first event item (a packet and an associated context) from the PSR_L queue
    Wait for the delay time specified by the event

    If the packet type specified by the event item is NONE Then
      SEND signal specified in the event item
    Else If the packet type is FIRST
          Enqueue the packet (and the associated context) to the OUP_L queue
          Signal the process Ethernet_Transmitter
          Mark the context to indicate rate control blocked state
        Else
          Enqueue the packet (and the associated context) to the RCNTL_L queue
          If the context does not indicate the rate control blocked state
            Signal the process XTP_RateControl
          EndIf
        EndIf
    EndIf

   Continue at Start
ENDPROCESS
```

4.2.5 Process XTP_RateControl

The process XTP_RateControl inserts delay between packets that go on the same route. It has the following functions: (a) To wait for a signal (from the process ETHERNET_TRANSMITTER (section 4.2.7)) indicates that a packet has been sent to the receiver, (b) to update the time the last packet has been sent for a context, (c) to look at each event (a context and a packet to be sent) from the RCNTL_L queue, (d) to calculate the time the packet will be delayed before it can be sent, (e) To insert the event and the delay time to the CALLOUT_L queue, and (f) to return to (a). Listing 3 shows the pseudo-code for the process XTP_RATECONTROL.

```
PROCESS XTP_RATECONTROL

    Input signals:
      • NEW_ENQUEUE from the process SERIALIZER
      • ETHER_DONE from the process ETHERNET_TRANSMITTER

    Output signals:
      • ARM_TIMER to the process XTP_Timer

Begin

Start:

    Wait for the signal ETHER_DONE from the process ETHERNET_TRANSMITTER or
          the signal NEW_ENQUEUE from the process XTP_SERIALIZER

    If the received signal is ETHER_DONE then
      Dequeue a context from the RTM_L queue (section 4.2.7)
      Change the state of the context from rate control blocked to rate control unblocked
      Record the time of the last packet has been sent for the current context
      Continue at New_Enqueue
    EndIf

    If the received signal is NEW_ENQUEUE then

New_Enqueue:

    While not at the end of the RCNTL_L queue Do
      Look at the current event (a packet and an associated context) of the RCNTL_L queue
        If the context does not indicate rate control blocked state
          Calculate the time the packet will be delayed before it can be sent
          Subtract the delay time of the packet by the amount of
                              (current time - the time the last packet has been sent for the context)
          Change the state of the context from rate control unblocked to rate control blocked
          dequeue the event (a packet and an associated context) from the RCNTL_L queue


          Insert the event and the delay time to the CALLOUT_L queue (the event with the shortest delay
                                        time is inserted at the head of the CALLOUT_L queue)

          If event was inserted at head of CALLOUT_L
            Send signal to the process XTP_Timer
          EndIf
        Endif
      Point to the next event of the RCNTL_L queue
    END While

    Continue at Start

ENDPROCESS
```

Listing 3: pseudo-code for the process XTP_RATECONTROL.

4.2.6 Process XTP_Timer

The process XTP_Timer maintains a callout list, which keeps track of delay time between packets to be sent on the same route or the time corresponding to various timers: WTIMER, CTIMER, etc. (section 2.11). The callout list is similar to those found in operating systems. The functions of the XTP_Timer process is: (a) Delay for the amount of time specified by the first event found on the CALLOUT_L queue, (b) if before the delay time expires and the process receives a signal informing that a new event has been inserted at the head of the CALLOUT_L queue, the timer process will restart at the new head event of the CALLOUT_L queue, (c) if the delay time expires, the process calls the function Timer_Intr_Handler, which will perform operations specified by the timer involved. For example, if WTIMER expires, a synchronizing handshake is entered. Listing 4 shows the pseudo-code for the process XTP_Timer.

```
PROCESS XTP_TIMER

  Input signals:
     • ARM_TIMER from the process XTP_RateControl or the function Timer_Intr_Handler

  Output signals:
     • PROCESSOR_START to the process Serializer
     • ETHER_SEND to the process Ethernet_Transmitter

BEGIN
   Wait for the ARM_TIMER signal

Start:

   Wait for the delay time specified in the first event of the CALLOUT_L queue

  If the signal ARM_TIMER is received before the delay time expires then
     Continue at Start
  Else (* the delay time expires *)
     dequeue the first event of the CALLOUT_L queue
     pass the event to the function Timer_Intr_Handler
     Continue at Start
  EndIf

ENDPROCESS
```

```
FUNCTION Timer_Intr_Handler

Input parameter:
      • Pointer to an event passed by the process XTP_Timer

BEGIN

    If the event is a timer of type CTIMER then
       enter synchronizing handshake
    EndIf

   If the event is a timer of type WTIMER then
      If endpoint is currently in the synchronizing handshake state then
         decrease the retry_count
         If retry_count < 0 then
            Terminate connection and simulation
         EndIf

         Save current sync value into saved_sync
         Send a CNTL packet with SREQ bit on
         re-insert the event on the CALLOUT_L queue with double delay time
      Else
         Enter Synchronizing handshake state
         Save current sync value into saved_sync
         Send a CNTL packet with SREQ bit on

         re-insert the event on the CALLOUT_L queue with double delay time
         If event was inserted at head of CALLOUT_L
            Send signal ARM_TIMER to the process XTP_Timer
         EndIf

      EndIf
   EndIf

   If the event specifies a packet to be sent
      append the event to the OUP_L queue
      Send the signal ETHER_SEND to the process ETHERNET_TRANSMITTER
   EndIf

END (* Function Timer_Intr_Handler *)
```

Listing 4: pseudo-code of the process XTP_Timer.

The Finite State Machine of the process XTP_Timer is shown in figure 4.5.

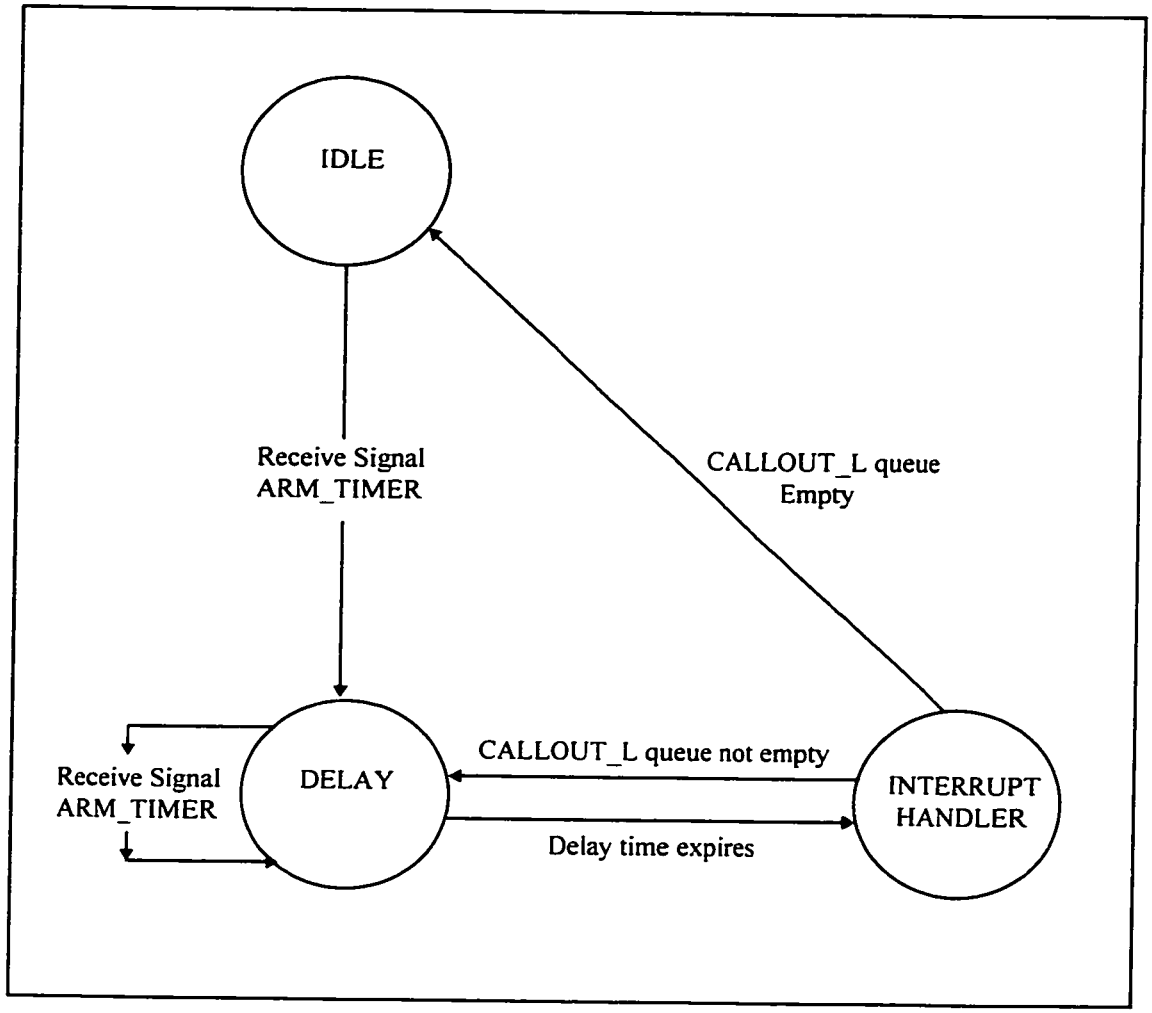


Figure 4.5: Finite State Machine of the process XTP_Timer.

4.2.7 The process Ethernet_Transmitter

The process Ethernet_Transmitter is used to transmit a packet sent via the underlying delivery service. The process has the following functions: (a) when the signal ETHER_SEND from the process XTP_RateContrl is received, the process dequeues the first event (a packet and the associated context) from the OUP_L queue), (b) it then transmits the packet, and (c) it enqueues the remaining context to the RTM_L queue, which is used by the process XTP_RateControl (section 4.2.5), and (d) it goes back to (a). The algorithms used by the process Ethernet_Transmitter to handle the problem of **packet collision** will be discussed in chapter 7.

4.2.8 The process Ethernet_Receiver

The process is used to receive a packet when it arrives at a station. After the receiving the packet is copied to a buffer of the station, the process Ethernet_Receiver enqueues the packet to the INP_L queue, and sends the signal ETHER_ARR to the process XTP_Receiver.

4.2.9 The process XTP_Receiver

The XTP_Receiver process performs the core of XTP packet reception operations. It has the following functions:

- Simulating checksum calculation of the receiving packet.
- Discard if data packet received has sequence number that exceeds buffer allocation of the receiving station, or data packet is a duplicate or packet is corrupted.
- If the receiving packet is a ECNTL packet, and the fields *nspan*, and *spans* of the ECNTL packet indicating lost packets (section 2.7), the process will retransmit the lost packets.

Listing 5 shows the pseudo-code for the process XTP_Receiver.

PROCESS XTP_RECEIVER

Input signals:
- ETHER_ARR from the process Ethernet_Receiver
- CHKSUM_DONE from the process Serializer

Output signals:
- DAT_PACKET_ARR to the process XTP_Reader
- FLOW_UNBLOCKED to the process XTP_Sender

BEGIN

wait for the ETHER_ARR signal from the process Ethernet_Receiver

Start:

Dequeue an event (an arriving packet) from the INP_L queue

Simulate the delay time for check sum calculation by sending a signal to the process SERIALIZER

Wait for the signal CHKSUM_DONE from the process SERIALIZER

Get the appropriate context that associated with the receiving packet.

Pass the context, and the receiving packet to the function **Update_x**

Continue at Start

ENDPROCESS

Listing 5: pseudo-code for the process XTP_Receiver.

```
Function Update_x

Input parameter:
    • Pointer to a context, and a packet

BEGIN

  If packet is a DATA packet then

    If  (Packet's sequence number + packet size) > alloc  OR
          (Packet's sequence < expected sequence number (rseq) ) OR (Packet is damaged) then
                Discard packet
                Return;
    EndIf

    If (packet's sequence number == rseq) then
      extend rseq by packet length

      If the context indicates the station that receives the packet has
                                        some lost packets before (nspan > 0) then
        Update nspan and spans
      endif

      enqueue the packet and the associated context on the RD_LIST queue
      Send the signal DAT_PACKET_ARR to the process XTP_Reader
    EndIf

  Else
    (* Packet is a CNTL packet *)
    If packet has its SREQ bit on then
      Send back a Control Status Packet
    Else
      Copy Control Status packet to the context
      If the field nspan in the Control Status Packet > 0 then
        Retransmit packets specified by nspans and spans fileds of the Control Status Packet
      EndIf

      If alloc limit was extended then
        unblock context if the context is currently in the flow control blocked state
        Send the signal FLOW_UNBLOCKED to the process XTP_Sender
      EndIf

      If the endpoint station was in the Synchronizing handshake state and the Control Status Packet
              received removes the synchronizing handshake state then
                unlock flow of new data packets for the context
      EndIf

      If the Control Status Packet is the reply of the previous request then
          Calculate RTT (Round Trip Time)
          Cancel WTIMER
      EndIf

    EndIf
  EndIf
END (* Function Update_x *)
```

4.2.10 The process XTP_Reader

The process XTP_Reader has the following functions:

- Simulating the delay time for copying data from the buffer of the station to the user's buffer.
- Record the gaps of missing data if the sequence number of the incoming data packet is greater then the expected sequence number (*rseq*).
- Update the *alloc* (section 2.13) value.
- Send a CNTL Status Packet if the incoming DATA packet has its SREQ bit on.
- Pass the non-duplicate data packets to the process Consumer.

Listing 6 shows the pseudo-code for the process XTP_Reader.

```
PROCESS XTP_READER

Input signals:
    • DAT_PACKET_ARR from the process XTP_Receiver
    • COPY_OUT_DONE from the process Serializer

Output signals:
    • PROCESSOR_START to the process Serializer
    • DATA_READY to the process Consumer

BEGIN

  Wait  for the signal DAT_PACKET_ARR from the process XTP_RECEIVER

Start:

  Dequeue an event from the RD_L queue

  Simulate  the delay time for copying data packet to user's buffer by sending a
                                                    signal to the process SERIALIZER

  Wait for the signal COPY_OUT_DONE by the process Serializer

  If packet's sequence number > expected sequence number then
     record the spans list

     If the SREQ bit of the packet is on then
        Send CNTL Status Packet to the Transmitter
     EndIf

     Enqueue event (the packet and the associated context) to the CONSUME_L queue
     Send the signal DATA_READY to the process Consumer

     Continue at Start
  EndIf

  Update alloc value

  If SREQ bit of the packet is on then
     Send CNTL Status Packet to the Transmitter
  EndIf

  Enqueue event (the packet and the associated context) to the CONSUME_L queue
  Send the signal DATA_READY to the process Consumer

  Continue at Start

ENDPROCESS
```

Listing 6: pseudo-code for the process XTP_Reader.

Figure 4.6 shows the Finite State Machine of the processs XTP_Reader.

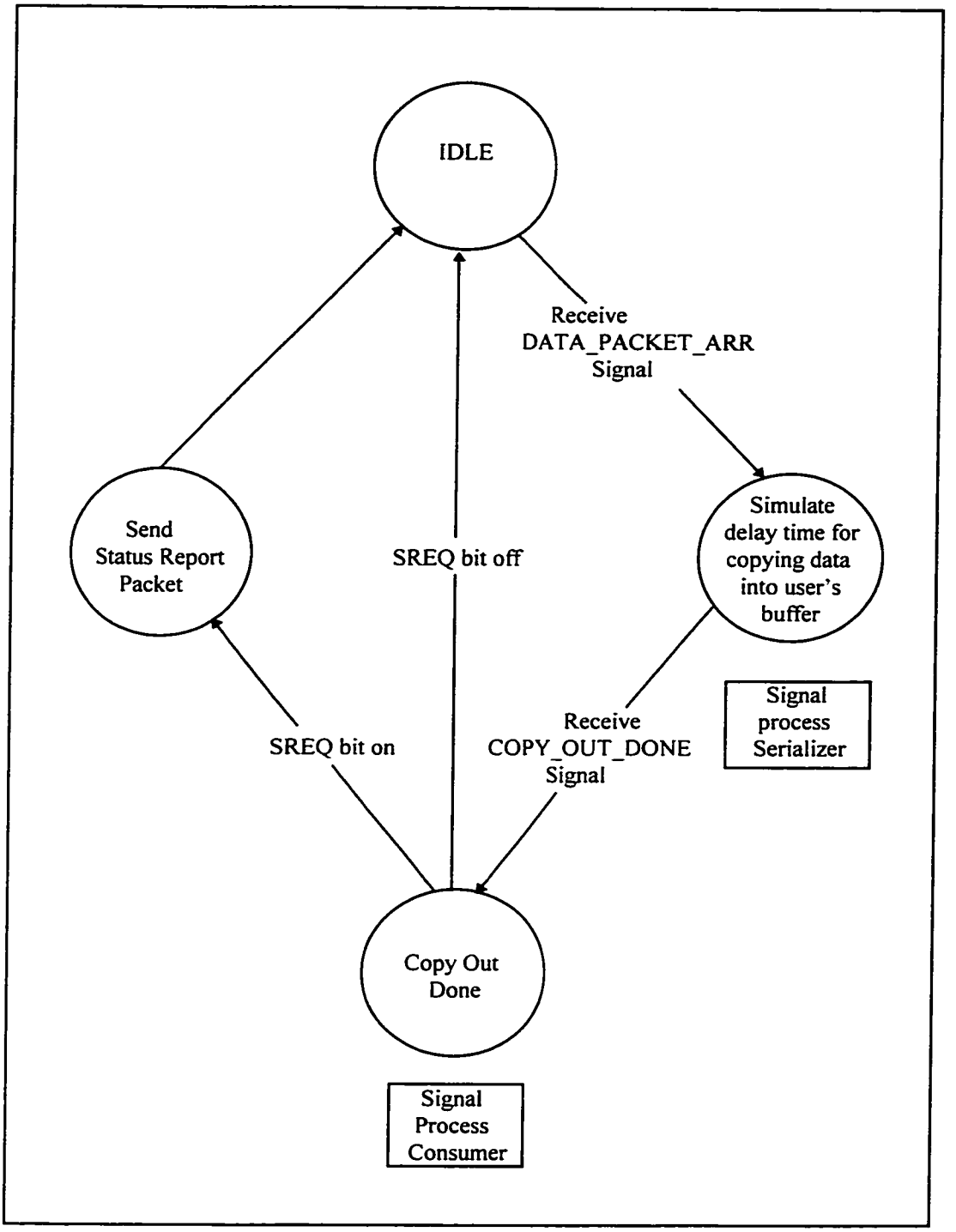


Figure 4.6: Finite State Machine of the process XTP_Reader.

4.2.11 The process Consumer

The process Consumer has the following functions:

- Simulating the delay time of consuming data in user's buffer.
- Send a CNTL Status Packet if the incoming DATA packet has its DREQ bit on.

Listing 7 shows the pseudo-code for the process Consumer.

```
PROCESS CONSUMER

Input signals:
    • DATA_READY from the process XTP_Reader
    • CONSUME_DONE from the process Serializer

Output signals:
    • PROCESSOR_START to the process Serializer

BEGIN

  Wait  for the signal DATA_READY from the process XTP_READER

Start:

  Dequeue an event from the CONSUME_L queue

  Simulate  the delay time for consuming data in user's buffer by sending a
                                                  signal to the process SERIALIZER

  Wait for the signal CONSUME_DONE by the process Serializer


  If DREQ bit of the packet is on then
    Send CNTL Status Packet to the Transmitter
  EndIf

  Continue at Start

ENDPROCESS
```

Listing 7: pseudo-code for the process Consumer.

4.3 Overall data structures used in the simulation

The main processes used in the simulation of XTP in SMURPH have been described in detail in section 4.2 above. These processes communicate with each other by using signals. Additional parameters to be passed between processes will require the using of queues. During process communication, whenever a process places an event on a queue to be used by another process, the initiating process will send a signal to the other process. Figure 4.6 shows the flow of data through the main processes of the simulation. The queues used for passing additional parameters between these processes are also included in figure 4.7. Figure 4.8 shows the signals that are sent and received between these processes.

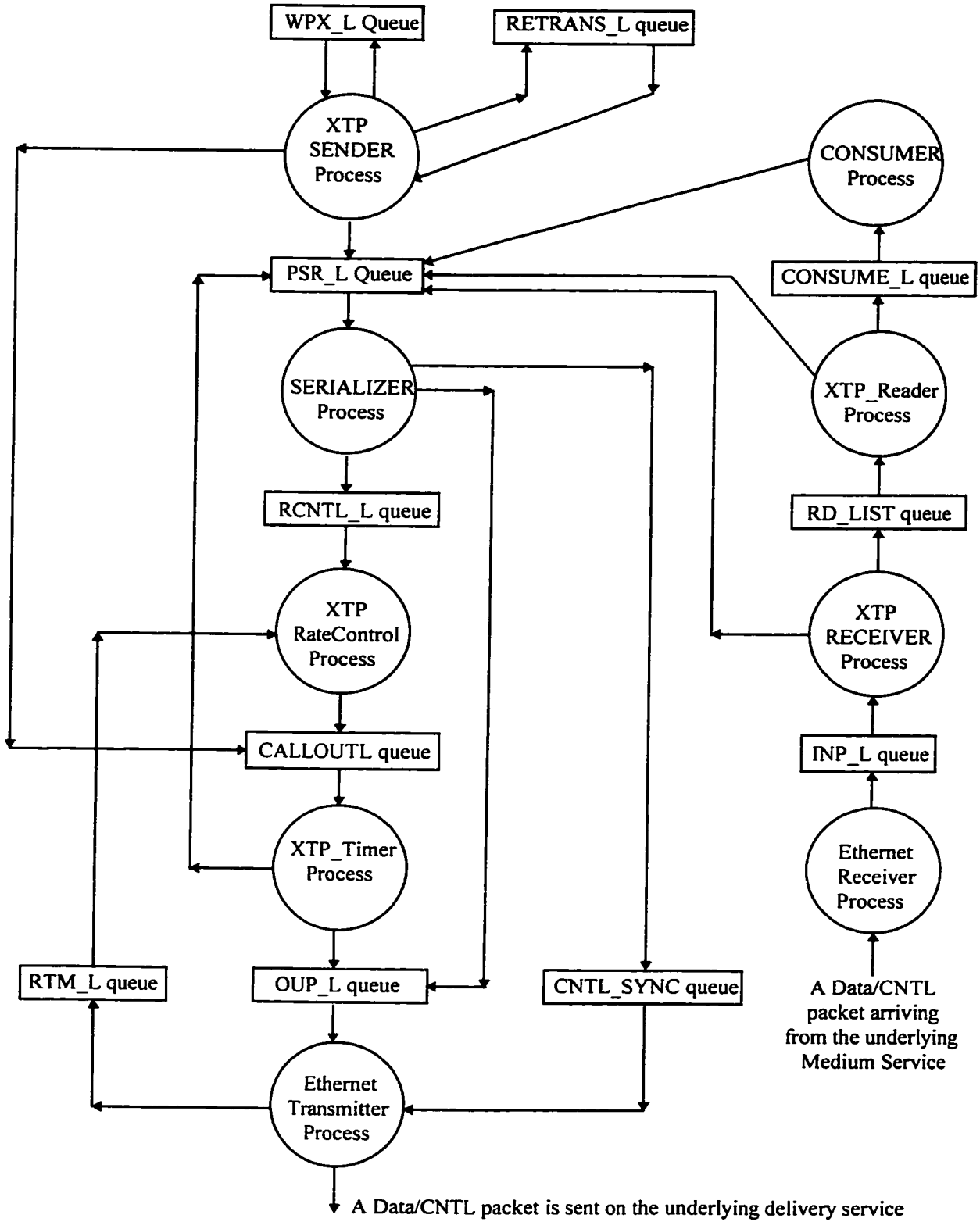


Figure 4.7: Data Flow Diagram between processes of a **station** of the simulation program.

As mentioned in the sections above, the processes XTP_Receiver, XTP_Reader, and Consumer, may have to send a CNTL packet. Similarly, a CNTL packet may be sent by the process XTP_Timer when a Timer expires. CNTL packets created by these processes and their associated contexts will be enqueued on the PSQ_ L queue which is processed by the process Serializer. When the process Serializer processes a CNTL packet whose associated context is in the Synchronizing Handshake state, the CNTL packet (and its associated context) will be enqueued on the CNTL_SYNC queue which has high priority to be processed by the process Ethernet_Transmitter. This will ensure that no data packets are sent (or resent) during a Synchronizing Handshake (section 2.12).

The purposes of the queues shown on figure 4.7 can be summarized as the following:

WPX_L : Used for preserving a pointer to a context that has packets waiting to be sent.

RETRANS_L: Hold copies of packets that have been transmitted, but not yet confirmed of receiving by the receiver.

PSR_L : Hold copies of (data/CNTL) packets (and their associated contexts) which have been processed by the processes: XTP_Sender, XTP_Timer, XTP_Receiver, XTP_Reader, and Consumer. This queue links these processes with the process Serializer.

RCNTL_L : Links the process Serializer and the process XTP_RateControl.

CALLOUTL : Links the process XTP_RateControl and the process XTP_Timer.

OUP_L : Links the processes Serializer, XTP_Timer with the process Ethernet_Transmitter.

CNTL_SYNC: Links the process Serializer and the process Ethernet_Transmitter.

RTM_L : Links the process Ethernet_Transmitter and the process XTP_RateControl. It is used to pass the time the last packet has been sent for a context.

INP_L : Links the process Ethernet_Receiver and the process XTP_Receiver. It is used for passing received (data/CNTL) packets between these processes.

RD_LIST : Links the process XTP_Receiver and the process XTP_Reader. It is used to pass received data packets.

CONSUME_L : Links the process XTP_Reader and the process Consumer.

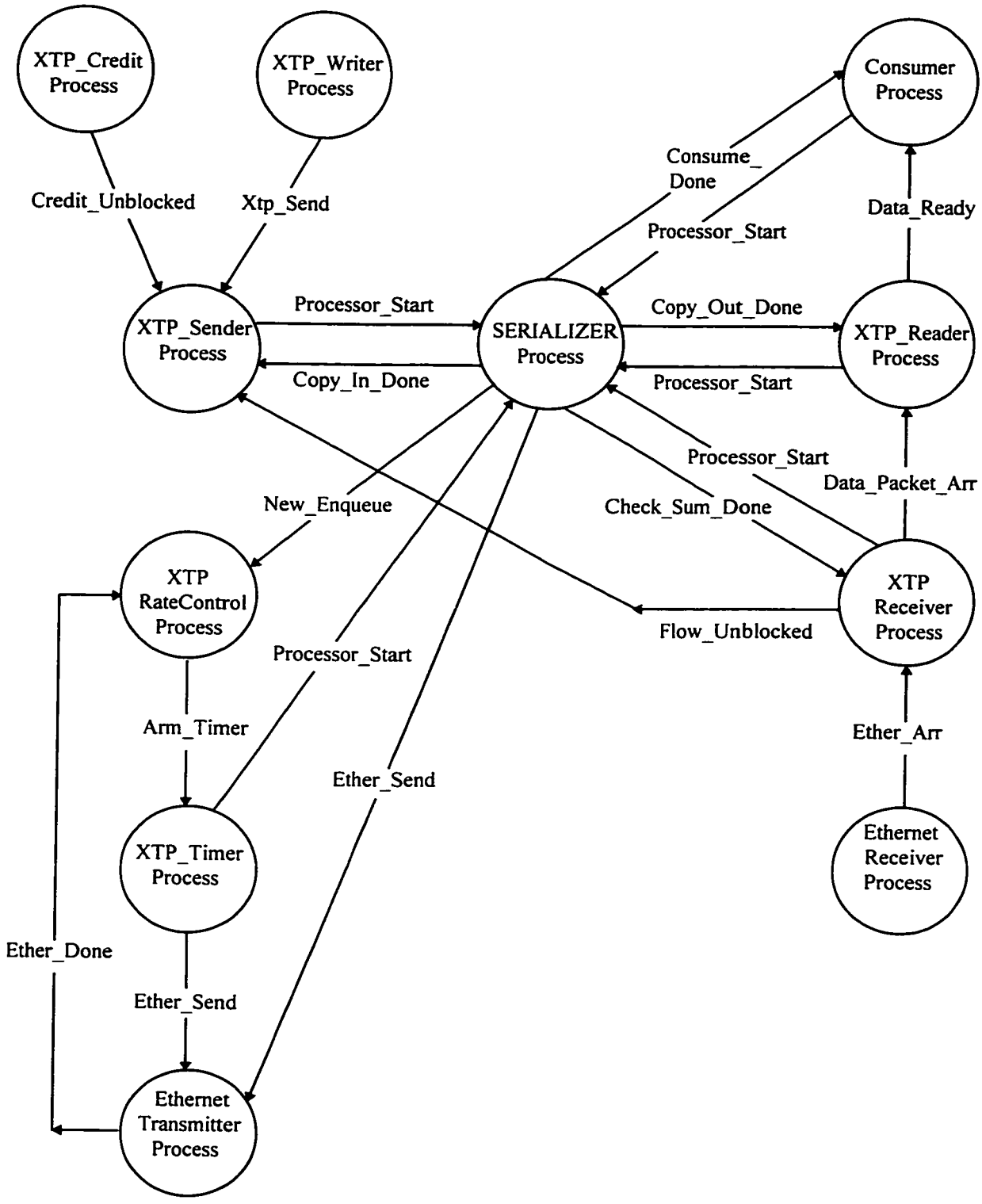


Figure 4.8: Signal Flow Diagram between processes of a **station** of the simulation program.

The purposes of the signals shown on figure 4.8 can be summarized as the following:

XTP_Send : Used by the process XTP_Writer to indicate to the process XTP_Sender that a new packet has been received.

Copy_In_Done : Used by the process SERIALIZER to indicate to the process XTP_Sender that the task of copying data from user's space to XTP's space has been completed.

Consume_Done : Used by the process SERIALIZER to indicate to the process Consumer that the task of consuming delay has been completed.

Copy_Out_Done : Used by the process SERIALIZER to indicate to the process XTP_Reader that the task of copying data from XTP's space to user's space has been completed.

Chk_Sum_Done : Used by the process SERIALIZER to indicate to the process XTP_Receiver that the task of check sum delay has been completed.

Processor_Start : Used by the processes XTP_sender, XTP_Timer, Consumer, XTP_Reader, XTP_Receiver to indicate to the process SERIALIZER that there is a request to use the processor.

New_Enqueue : Used by the process SERIALIZER to indicate to the process XTP_RateControl that a new unblocked packet and its associated context has been enqueued to the RCNTL_L queue.

Arm_Timer : Used by the process XTP_RateControl to indicate to the process XTP_Timer that an event (a packet with its delay time and its associated context) have been inserted on the head of the CALLOUTL queue.

Ether_Send : Used by the processes XTP_Timer, and SERIALIZER to indicate to the process Ethernet_Transmitter that there is an outgoing packet and its associated context have been enqueued to the OUP_L queue.

Ether_Done : Used by the process Ethernet_Transmitter to indicate to the process XTP_RateControl that a packet has been sent successfully and that the associated context of the packet, and the time the packet was sent have been enqueued to the RTM_L queue.

Ether_Arr : Used by the process Ethernet_Receiver to indicate to the process XTP_Receiver that a (data/CNTL) packet has been received and enqueued to the INP_L queue.

Data_Packet_Arr : Used by the process XTP_Receiver to indicate to the process XTP_Reader that a new data packet has been received and enqueued to the RD_LIST queue.

Data_Ready : Used by the process XTP_Reader to indicate to the process Consumer that a new data packet has been enqueued to the CONSUME_L queue.

4.4 Determining the Round Trip Time (RTT)

As discussed in section 2.11, the value of the round trip time (RTT) is needed for the timers. But getting the accurate values for RTT can be very difficult.

In XTP, if SREQ is set in a DATA packet, a CNTL packet is returned immediately, making it is possible to determine the round trip time (the correlation of the returning CNTL packet with the original DATA packet with the SREQ bit set is possible because the *sync* value in the DATA packet is returned as *echo* field in the CNTL packet). If the sending times for all DATA packets with SREQ bit set have been recorded, it is possible to calculate the RTT.

4.5 Policy for the setting of the DREQ bit in bulk transfer

In Chen's simulation [Chen89], the sizes of the messages were smaller than the allocation window, so an update of the receiver status was obtained frequently enough (section 2.8, 2.13). In the simulation of bulk data transfer [Chung93], the message is much longer than the allocation window. Chung Kam Chung [Chung93] has investigated the policy that should be used for the setting of DREQ, to ensure that data flows without interruption.

At a certain point in time, before the allocation window will close, the transmitter should set the DREQ bit on an outgoing DATA packet so that the CNTL status packet sent in reply will reach the transmitter with an updated *alloc* value before the old *alloc* limit is reached. This will ensure that the flow of data will not be blocked. The guiding principle in designing the condition to be satisfied is that there should be enough time to send a status request, receive the response, send the missing packets, and receive the updated *alloc* value, prior to the closure of the *alloc* window. Given a *rate* value for the underlying route, and an observed round trip time (RTT), the condition for setting the DREQ bit is as following:

$$(alloc - (seq + packet\_len)) \leq (2 \times rate \times RTT)$$

Chung Kam Chung [Chung93] has examined a variety of scenarios, using this condition for setting the DREQ bit, and determined that the flow of DATA packets will remained uninterrupted in all error cases.

5. XTP SIMULATION PERFORMANCE

The XTP simulation program is based on the work of Chen [Chen89], and Chung Kam Chung [Chung93]. Chen simulated the revision 3.3 of XTP using LANSF. Chen wanted to find out the maximum throughput of XTP simulation in a no error environment. Chung Kam Chung extended Chen's work by modeling the error control of XTP (revision 3.6) in LANSF.

The XTP simulation program in SMURPH has been run on a SUN processor using SUN OS 4.1.3. The standard UNIX debugging tool dbx was used during the process of implementing the XTP simulation program.

5.1 The SMURPH Tunable Parameters

There are many tunable parameters in SMURPH. We are only interesting in the following four:

- The Message Length, which specifies the length of a message which is generated by SMURPH. This parameter is specified in bits.
- The Message Interarrival Time, which specifies the rate of messages arriving at a station of the XTP simulation.
- The Number of Stations of the XTP simulation.
- The Maximum Number of Messages that SMURPH can generate for a station of XTP simulation. Once this number is reached, an XTP simulation run is finished.

5.2 The XTP Tunable Parameters

The tunable parameters of the XTP simulation program are:

- *alloc*, which defines the allocation window size in bytes (section 2.13).

- *wtimer*, which defines the initial value of WTIMER (section 2.11). WTIMER is updated when a new value is calculated for RTT (section 4.4).

- *ctimer*, the context life timer value (section 2.11).

- *copy_delay*, which specifies the delay time for copying data from user's buffer space to XTP's buffer space, or from XTP's buffer space to user's buffer space.

- *checksum_delay*, which specifies the check sum delay time for outgoing and incoming packets of a station.

- *consume_delay*, which specifies the delay time caused by the consumption of a a data packet by an XTP's user of the receiving station.

- *XTP_ERROR*, *XTP_FASNAK*, *XTP_GO_BACK_N*, *XTP_SELECT_RETRANS* are Boolean variables stating whether error control (with Go-Back-N (section 2.7) or selective retransmission (section 2.7)), or FASNAK (section 2.6.1) should be enabled in a run of the simulation program. If XTP_ERROR is not enabled, the the simulation program will disable the error control processes.

5.3 Measurements

There are two measurements that we wish to obtain from a run of the simulation program: the **actual throughput** and average **message delay**. The throughput of the simulation is obtained by using the following formula [Chen89]:

$$\textit{Actual Throughput of the Simulation} = \frac{\textit{Delivered Sequence} \text{ [Bytes]}}{\textit{Last Sent Completed} \text{ [seconds]}}$$

where *Delivered Sequence* is the sequence number of the last packet successfully delivered to the XTP's user, and *Last Sent Completed* is the simulation time when that delivery occurred.

Message delay is the difference between the time when the message was queued by the process XTP_Writer (section 4.2.2), which simulates the XTP's user at the transmitting end, and the time when the message is received by the process Consumer (4.2.11), which simulates the XTP's user at the receiving end.

5.4 The Simulation Plan

The XTP simulation involves two stations using a single virtual circuit. The circuit setup is done at the initialization processes of the simulation program.

The message sizes used for different runs of the simulation are 6 Byte, 128 Byte, 1KB (1,024 Bytes), 8 KB (8,192 Bytes), and 1 MB.

- 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 a terminal accessing activity.
- The 128 Byte message represents a Remote Procedure Call (RPC) activity.
- The 1KB (1,024 Bytes) message represents a page fetch operation.
- The 8 KB (8,192 Bytes) message represents a file transfer operation.
- The 1 MB (1,024 KB) message represents a large file transfer operation. This message size is much larger than the allocation window size, and thus requires the policy outlined in section 4.5 to be used.

There are different load numbers used for the simulation runs of a message size. The offered load numbers were calculated by partitioning the expected maximum throughput of the simulation into equal intervals. The load numbers are equal to 10%, 20%, 30%, etc., of the maximum throughput. Four more offered load numbers were added to the simulation of the message size of 1MB. They are equal to 110%, 120%, 130% and 140 % of the expected

maximum throughput. These four added load numbers are used to determine the performance of XTP when it is saturated.

The expected maximum throughput for a message size is given by [Chen89]:

$$Maximum\ Expected\ Throughput = \frac{User\ Data\ Length}{Total\ Packet\ Length} \times Total\ Bandwidth$$

The XTP simulation run for a message size with a typical load is carried out twice: the first run assumes that there is no delay for copying data from and to user's buffer space, while the second run assumes there is a delay. The checksum delay is assumed to be included in the copy delays, because it is possible for the XTP entities to calculate the checksum for a packet while it is being copied.

SMURPH generates 10,000 messages for a XTP simulation run of a message size. For message size of 1MB (bulk transfer), only 100 messages are used since larger number of messages results in simulation crash due to the lack of memory. Given that the object transferred is much larger for this case than it is for the other cases, this reduction is not significant.

For an XTP simulation run with error control, the bit error rate is 10^{-6} (each bit of the Data/CNTL packet has a chance of 10^{-6} to be corrupted).

For a typical message size, under a specific load, the performance of the XTP simulation is measured in three different cases:

1) No error occurred during the Simulation.
2) Error may occur during the simulation. Lost data packets are retransmitted by Selective Retransmission Error Control Mechanism.

3) Error may occur during the simulation. Lost data packets are retransmitted by Go-Back-N Error Control Mechanism.

5.5 The Simulation Results

Tables 5.1 to 5.5 show the performance of XTP during the transfer of messages of sizes 6 Byte, 128 Byte, 1KB, 8KB, and 1 MB, with no delay in copying data from and to user's buffer space, under different loads specified in section 5.4. Figures 5.1 to 5.5 show the corresponding graphs for the tables 5.1 to 5.5.

Similarly, tables 5.6 to 5.10 show the performance of XTP during the transfer of messages of sizes 6 Byte, 128 Byte, 1KB, 8KB, and 1 MB, with delay in copying data from and to user's buffer space, under different loads specified in section 5.4. Figures 5.5 to 5.10 show the corresponding graphs for the tables 5.5 to 5.10.

The effective throughputs in tables 5.1 to 5.10 are derived from the formula discussed in section 5.3. The effective throughputs may be greater than the message arrival rates (offered loads). The reason is that, the total delivered sequence number (in Bytes) of a station of the XTP simulation run (as reported by SMURPH) will also include the headers of all the data packets arriving at the station.

High throughputs are observed from figures 5.1 to 5.10. The maximum throughputs observed for 8 KB and 1MB messages are approximately 85% of the available bandwidth.

From figures 5.1 to 5.10, we notice that the decrease in the performance of XTP due to errors being allowed to occur, is minimal for message sizes of 6 Byte, 128 Byte and 1 KB. For message sizes of 8 KB, and 1MB, there is approximately 25% decrease in performance when errors are introduced. This is mainly due to the fact that smaller message sizes have a smaller probability that the message will be corrupted during the simulation. A 6 Byte

message has a probability of 4×10^{-3} of getting corrupted, whereas a 8 KB message, which is broken into data packets, has a probability of 6.76×10^{-1} of getting corrupted.

Tables 5.11 to 5.14 show the average message delay for the 6 Byte, 128 Byte, 1KB and 8 KB messages. Since saturation is already reached in bulk transfer (message size of 1 MB), recording the average message delay is meaningless. Figures 5.11 to 5.14 show the corresponding graphs for the tables 5.11 to 5.14.

The performance of the underlying Ethernet has some impact on the performance of XTP [Chung93]. The average message delay is expected to grow exponentially in the Ethernet environment once the load offered reaches a certain point [Gburzynski91]. Due to the probabilistic nature of the Ethernet protocol, it may happen that two or more stations will collide for an arbitrarily long time. This probability is described as the Ethernet Capture Effect, and it is increased tremendously when the load offered reaches a certain threshold with large message sizes. As a result, the backoff algorithm, which is used when a collision has occurred, plays a large role in the performance of Ethernet, especially when the message size is large (for example, message sizes of 8KB, or 1MB). A good backoff algorithm will decrease the probability that the stations resend their packets at the same time after a collision has occurred. The Ethernet Capture Effect and an enhanced Backoff algorithm will be discussed in chapter 7.

[Chen89] mentioned that the copy and checksum delays play an important role in the performance of XTP. From the results of the XTP simulation runs with error control, we can conclude that these delays are not as important as the performance of the underlying Ethernet. Ethernet was found to bottleneck, limiting the transfer of data. The delay resulting from the collisions occurring in the underlying network was, occasionally, 50 times higher than the copy delay. A good backoff algorithm should be able to decrease the number of collisions, and thus increase the performance of XTP (Chapter 7).

Message Length 6 Bytes (48 bits)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
2,730	22,300	22,100	22,200
5,328	44,700	44,700	44,500
7,986	66,700	67,000	66,500
10,644	89,300	89,700	90,400
13,302	112,000	112,000	112,000
15,960	135,000	133,000	135,000
18,618	156,000	157,000	154,000
21,276	178,000	177,000	178,000
23,934	201,000	202,000	200,000
26,582	223,000	222,000	220,000
29,250	247,000	240,000	249,000
31,908	268,000	266,000	260,000
34,566	284,000	274,000	257,000
37,244	307,000	267,000	246,000
39,894	315,000	269,000	258,000
42,553	316,000	271,000	263,000

Table 5.1: Throughput vs. Offered loads for 6 Byte Messages (no delay).

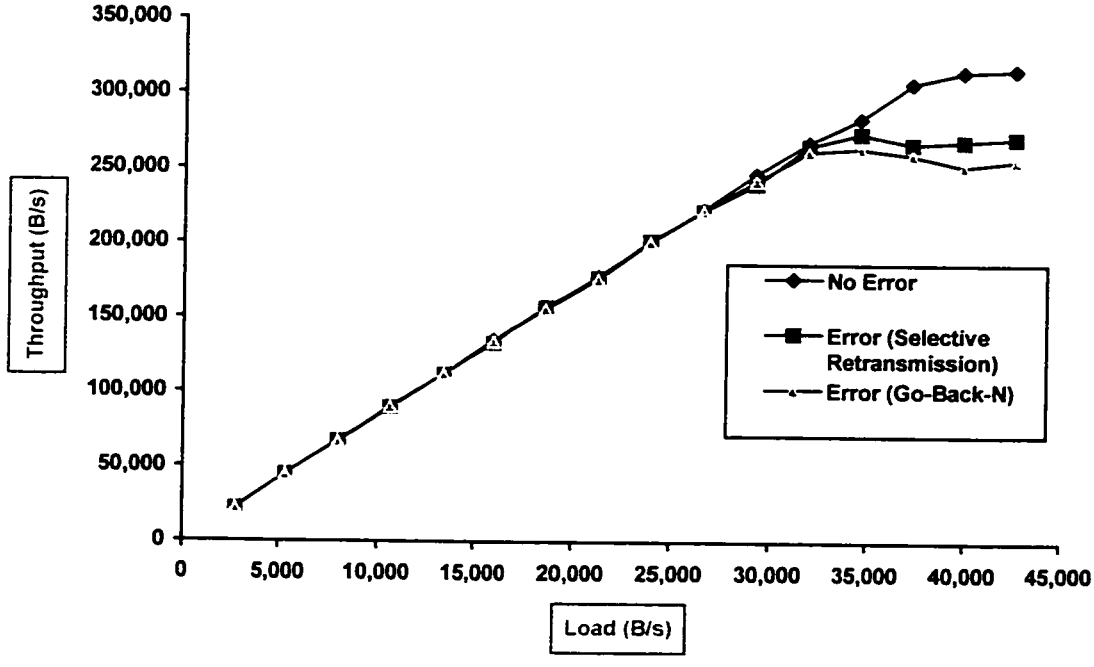


Figure 5.1: Throughput vs. Offered loads for 6 Byte messages (no delay).

Message Length 128 Bytes (1024 bits)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
36,680	53,400	52,900	53,300
79,616	107,000	107,000	107,000
119,170	160,000	160,000	161,000
158,730	213,000	213,000	214,000
198,295	267,000	269,000	270,000
237,829	320,000	314,000	321,000
277,356	373,000	372,000	376,000
316,989	426,000	429,000	437,000
356,447	479,000	487,000	477,000
396,040	533,000	531,000	540,000
435,819	586,000	573,000	559,000
475,483	639,000	573,000	545,000
515,091	675,000	574,000	540,000
554,353	677,000	575,000	549,000

Table 5.2: Throughput vs. Offered loads for 128 Byte Messages (no delay).

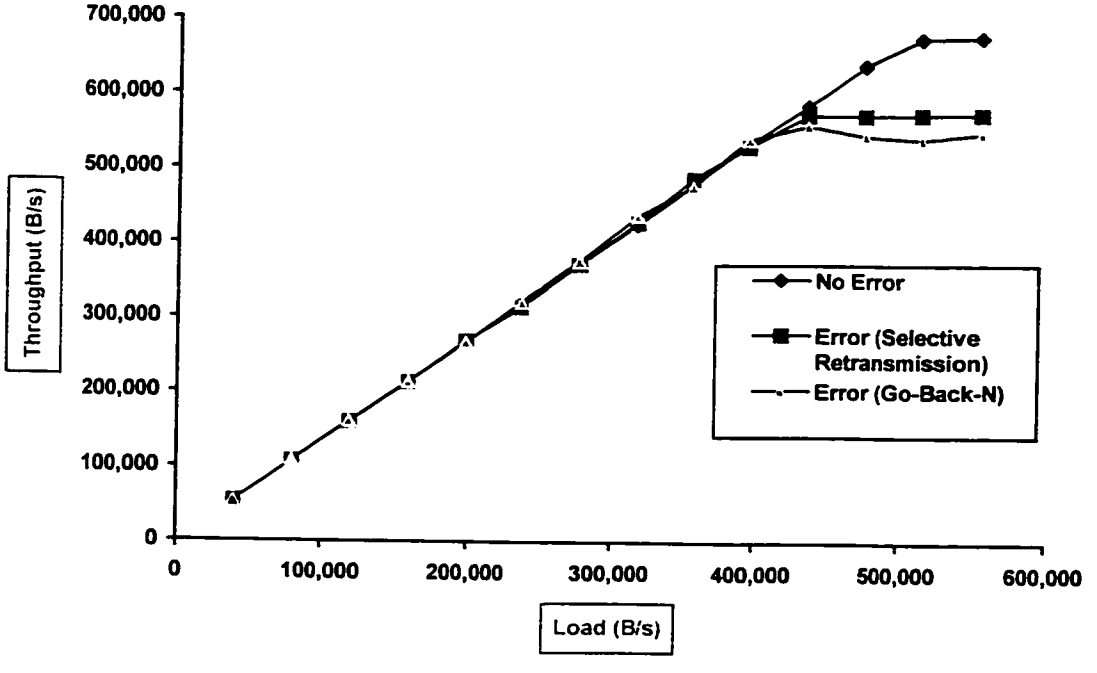


Figure 5.2: Throughput vs. Offered loads for 128 Byte messages (no delay).

Message Length 1 KByte (1024 Bytes)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
98,305	103,000	101,000	103,000
197,634	206,000	201,000	211,000
295,937	309,000	308,000	315,000
394,240	411,000	416,000	418,000
492,544	514,000	517,000	519,000
590,849	617,000	625,000	632,000
689,191	719,000	713,000	729,000
787,510	822,000	821,000	847,000
879,347	918,000	917,000	920,000
984,142	1,030,000	967,000	909,000
1,082,452	1,100,000	961,000	923,000
1,180,676	1,100,000	964,000	917,000
1,279,041	1,100,000	962,000	900,000

Table 5.3: Throughput vs. Offered loads for 1 KByte Messages (no delay).

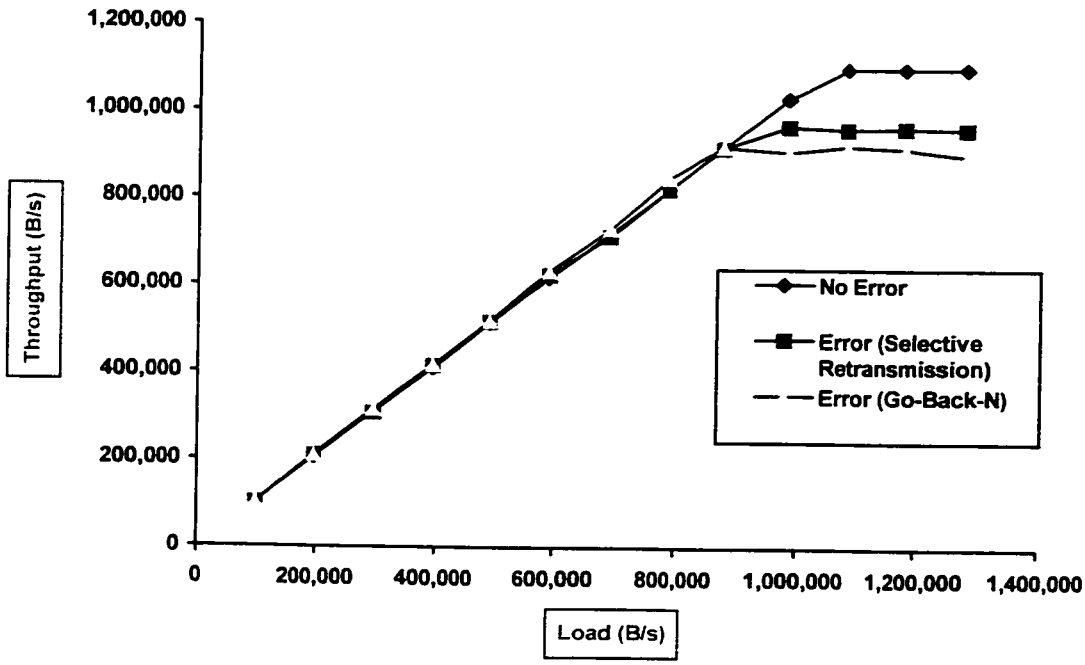


Figure 5.3: Throughput vs. Offered loads for 1 KByte messages (no delay).

Message Length 8 KByte (8192 Bytes)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
114,688	115,000	120,000	120,000
229,377	235,000	242,000	238,000
344,064	355,000	359,000	354,000
458,753	474,000	479,000	480,000
573,441	592,000	597,000	594,000
688,132	711,000	710,000	712,000
786,437	812,000	812,000	818,000
907,511	948,000	944,000	951,000
1,032,219	1,070,000	1,030,000	992,000
1,148,889	1,130,000	1,030,000	994,000
1,261,569	1,130,000	1,030,000	993,000
1,376,252	1,130,000	1,030,000	991,000
1,490,946	1,130,000	1,030,000	992,000

Table 5.4: Throughput vs. Offered loads for 8 KByte Messages (no delay).

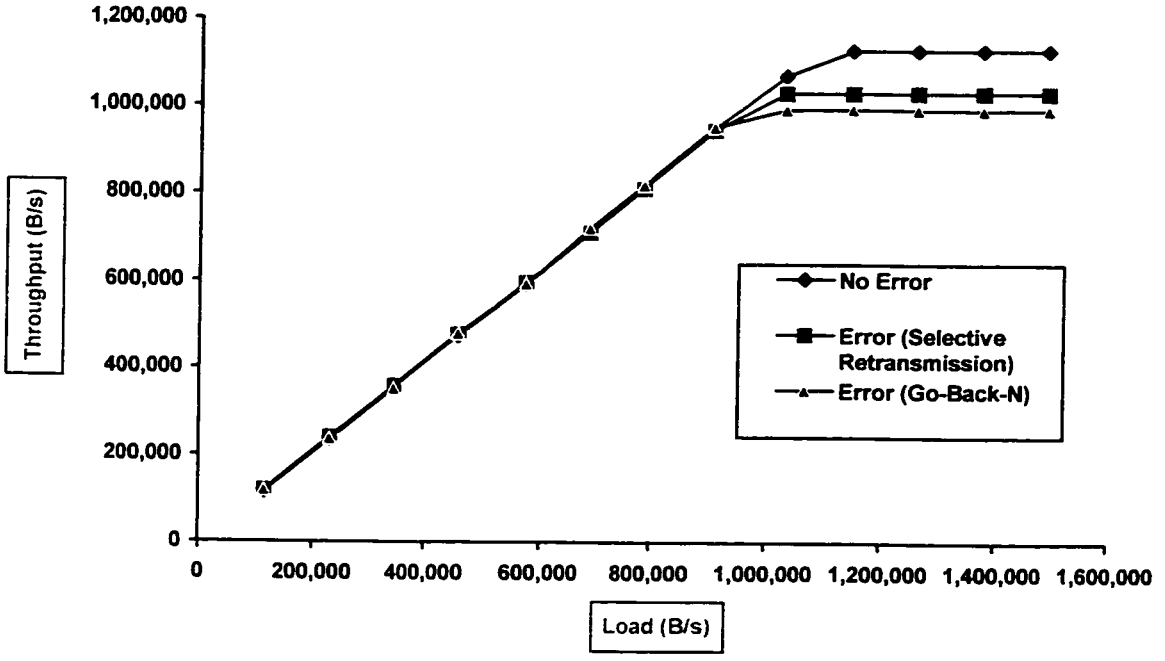


Figure 5.4: Throughput vs. Offered loads for 8 KByte messages (no delay).

Message Length 1 MByte (1024 KByte)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
114,688	112,000	144,000	112,000
196,608	231,000	226,000	248,000
327,680	317,000	349,000	376,000
458,752	467,000	474,000	493,000
589,824	650,000	597,000	545,000
720,896	737,000	761,000	747,000
851,968	820,000	908,000	740,000
917,507	858,000	785,000	702,000
1,048,576	987,000	988,000	724,000
1,179,648	977,000	1,080,000	741,000
1,245,184	1,070,000	1,080,000	760,000
1,310,720	1,080,000	1,050,000	731,000
1,441,792	1,090,000	1,080,000	751,000
1,572,864	1,130,000	1,080,000	754,000
1,703,936	1,130,000	1,090,000	740,000
1,835,008	1,130,000	1,090,000	745,000

Table 5.5: Throughput vs. Offered loads for 1 MByte Messages (no delay).

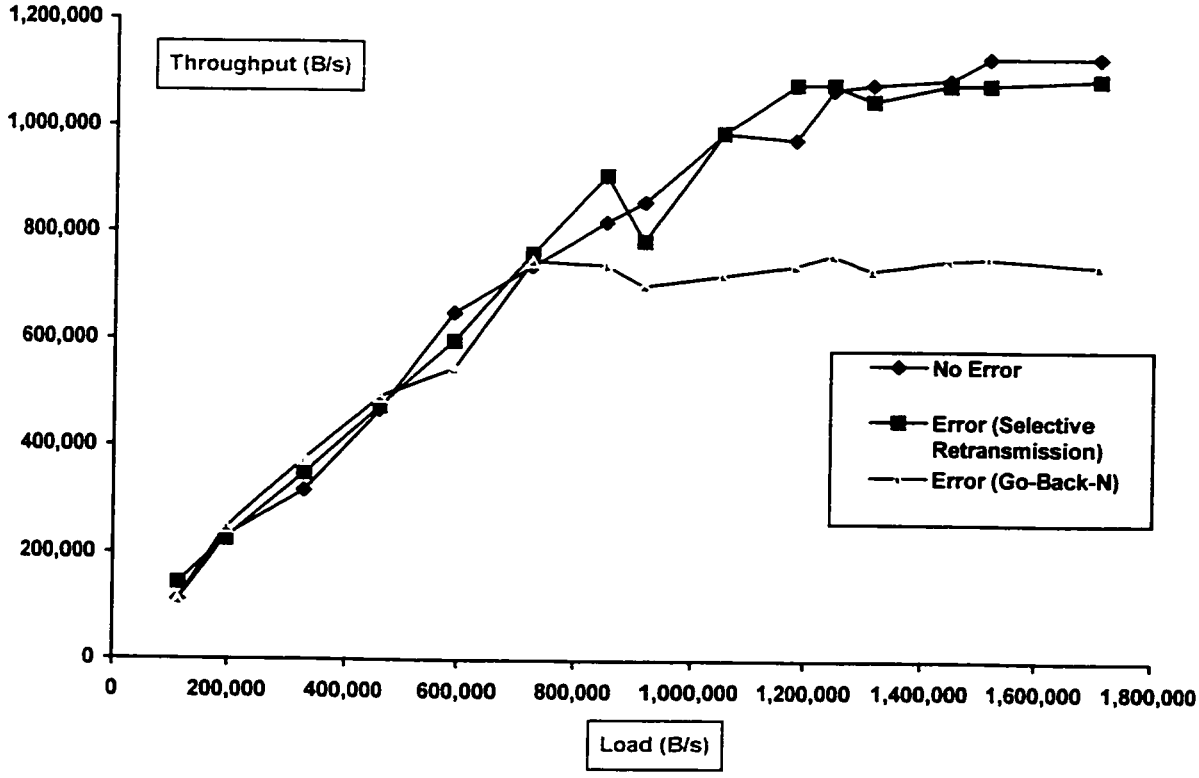


Figure 5.5: Throughput vs. Offered loads for 1 MByte messages (no delay).

Message Length 6 Bytes (48 bits)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
2,730	22,300	22,300	22,100
5,328	44,200	44,200	44,600
7,986	66,300	66,700	66,700
10,644	89,700	89,500	90,300
13,302	112,000	112,000	112,000
15,960	137,000	136,000	134,000
18,618	157,000	154,000	156,000
21,276	178,000	178,000	176,000
23,934	196,000	201,000	201,000
26,582	221,000	221,000	220,000
29,250	243,000	246,000	242,000
31,908	265,000	268,000	262,000
34,566	286,000	270,000	264,000
37,244	311,000	270,000	260,000
39,894	315,000	273,000	252,000
42,553	316,000	272,000	256,000

Table 5.6: Throughput vs. Offered loads for 6 Byte Messages (with delay).

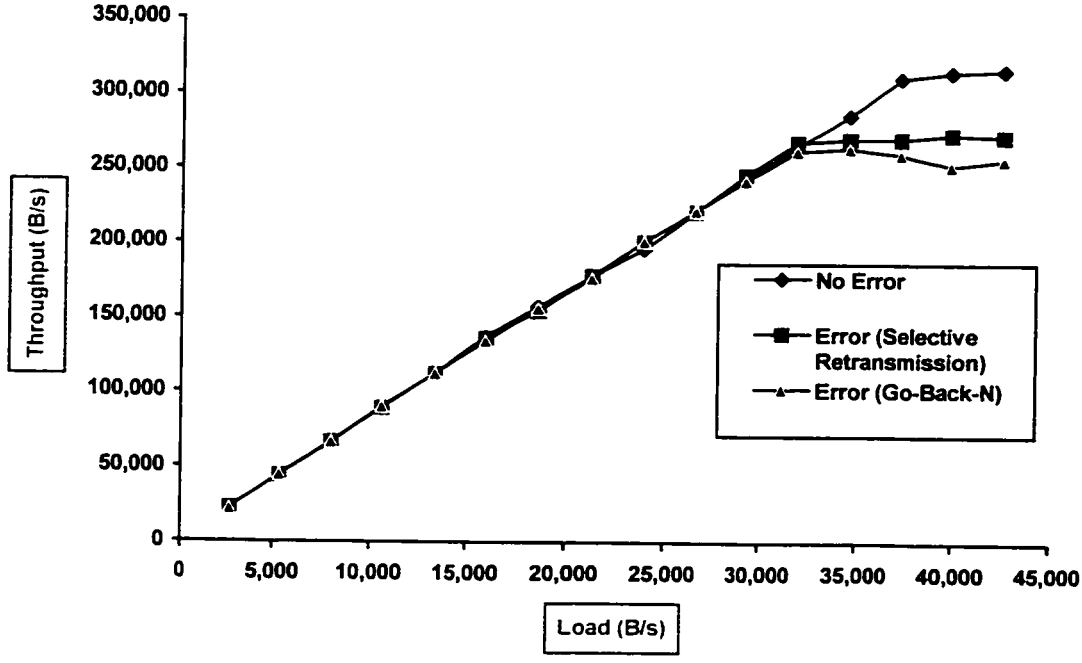


Figure 5.6: Throughput vs. Offered loads for 6 Byte messages (with delay).

Message Length 128 Bytes (1024 bits)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
36,680	53,400	53,400	53,600
79,616	108,000	107,000	109,000
119,170	160,000	163,000	163,000
158,730	215,000	213,000	214,300
198,295	268,000	268,000	265,000
237,829	322,000	320,000	322,000
277,356	372,000	372,000	376,000
316,989	426,000	426,000	427,000
356,447	483,000	476,000	481,000
396,040	536,000	533,000	535,000
435,819	591,000	575,000	541,000
475,483	645,000	576,000	537,000
515,091	675,000	576,000	538,000
554,353	677,000	575,000	538,000

Table 5.7: Throughput vs. Offered loads for 128 Byte Messages (with delay).

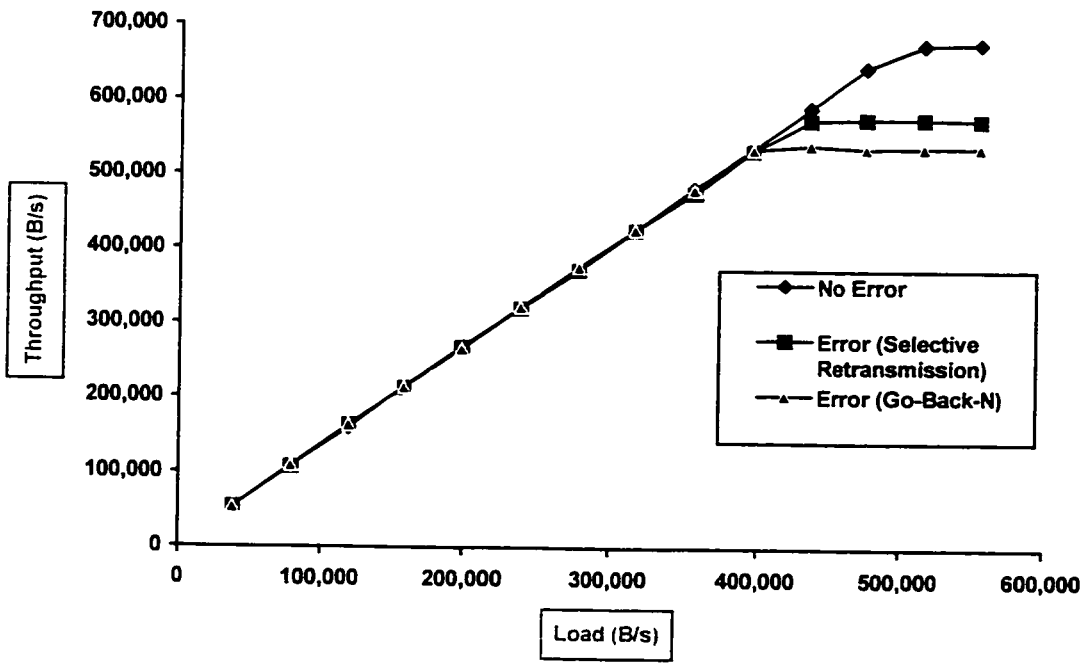


Figure 5.7: Throughput vs. Offered loads for 128 Byte messages (with delay).

Message Length 1 KByte (1024 Bytes)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
98,305	103,000	103,000	102,000
197,634	206,000	208,000	209,000
295,937	312,000	311,000	311,000
394,240	411,000	417,000	422,000
492,544	516,000	514,000	516,000
590,849	617,000	612,000	616,000
689,191	717,000	726,000	725,000
787,510	822,000	821,000	821,000
879,347	918,000	944,000	908,000
984,142	1,030,000	963,000	920,000
1,082,452	1,100,000	962,000	921,000
1,180,676	1,100,000	964,000	917,000
1,279,041	1,100,000	964,000	922,000

Table 5.8: Throughput vs. Offered loads for 1 KByte Messages (with delay).

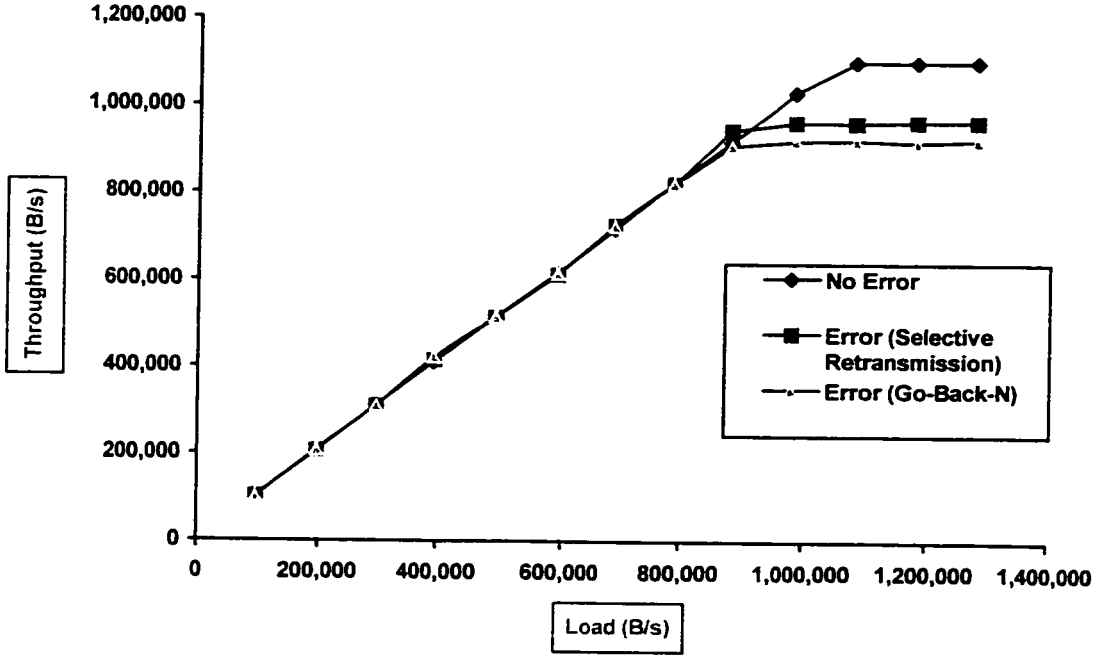


Figure 5.8: Throughput vs. Offered loads for 1 KByte messages (with delay).

Message Length 8 KByte (8192 Bytes)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
114,688	115,000	122,000	119,000
229,377	235,000	243,000	239,000
344,064	360,000	352,000	352,000
458,753	474,000	473,000	473,000
573,441	592,000	585,000	588,000
688,132	711,000	715,000	704,000
786,437	812,000	830,000	814,000
907,511	948,000	953,000	953,000
1,032,219	1,070,000	1,060,000	1,010,000
1,148,889	1,130,000	1,060,000	1,010,000
1,261,569	1,130,000	1,060,000	1,020,000
1,376,252	1,130,000	1,060,000	1,010,000
1,490,946	1,130,000	1,060,000	1,010,000

Table 5.9: Throughput vs. Offered loads for 8 KByte Messages (with delay).

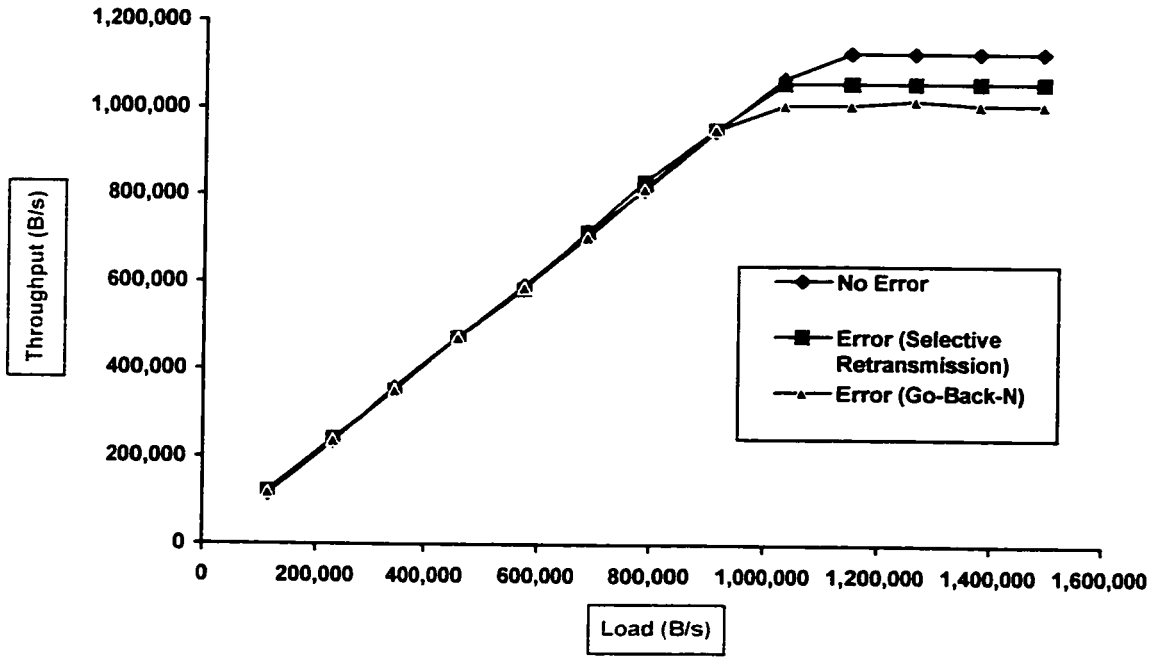


Figure 5.9: Throughput vs. Offered loads for 8 KByte messages (with delay).

Message Length 1 MByte (1024 KByte)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (Selective Retransmission)	Error (Go-Back-N)
114,688	137,000	102,000	118,000
196,608	168,000	250,000	259,000
327,680	367,000	355,000	328,000
458,752	429,000	461,000	441,000
589,824	553,000	621,000	530,000
720,896	704,000	693,000	679,000
851,968	1,070,000	895,000	746,000
917,507	928,000	967,000	947,000
1,048,576	1,120,000	955,000	907,000
1,179,648	1,110,000	1,100,000	954,000
1,245,184	1,130,000	1,100,000	949,000
1,310,720	1,130,000	1,090,000	976,000
1,441,792	1,130,000	1,100,000	982,000
1,572,864	1,130,000	1,100,000	983,000
1,703,936	1,130,000	1,100,000	973,000
1,835,008	1,130,000	1,100,000	981,000

Table 5.10: Throughput vs. Offered loads for 1 MByte Messages (with delay).

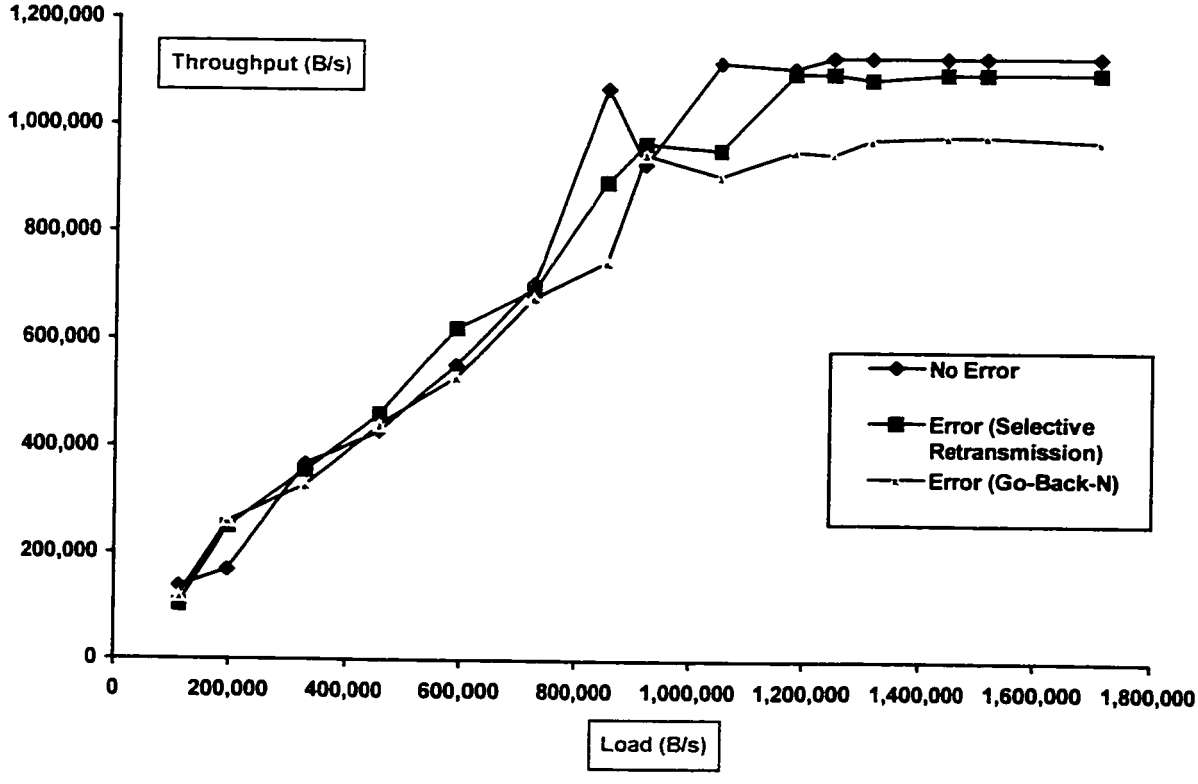


Figure 5.10: Throughput vs. Offered loads for 1 MByte messages (with delay).

Message Length 6 Bytes (48 bits) Message Copy delay 16 Time Unit	Average Message Delay (TU) TU = Time Unit 1 second = 10,000,000 TU			
Message Rate (B/s)	No Error		Error	
	No Delay	Delay	No Delay	Delay
2,730	640	724	645	735
5,328	741	784	751	793
7,986	810	842	818	863
10,644	893	944	921	958
13,302	978	1,030	1,030	1,090
15,960	1,090	1,130	1,190	1,210
18,618	1,170	1,260	1,410	1,440
21,276	1,290	1,310	1,670	1,680
23,934	1,400	1,410	2,070	2,170
26,582	1,170	1,530	3,010	2,930
29,250	1,670	1,700	5,350	8,220
31,908	1,750	1,790	65,200	86,400
34,566	1,850	1,870	510,000	475,000
37,244	35,600	30,400	1,150,000	1,260,000
39,894	271,000	352,000	1,860,000	1,690,000
42,553	811,000	757,000	2,220,000	2,120,000

Table 5.11: Average Message Delay vs. Offered Loads for 6 Byte Messages.

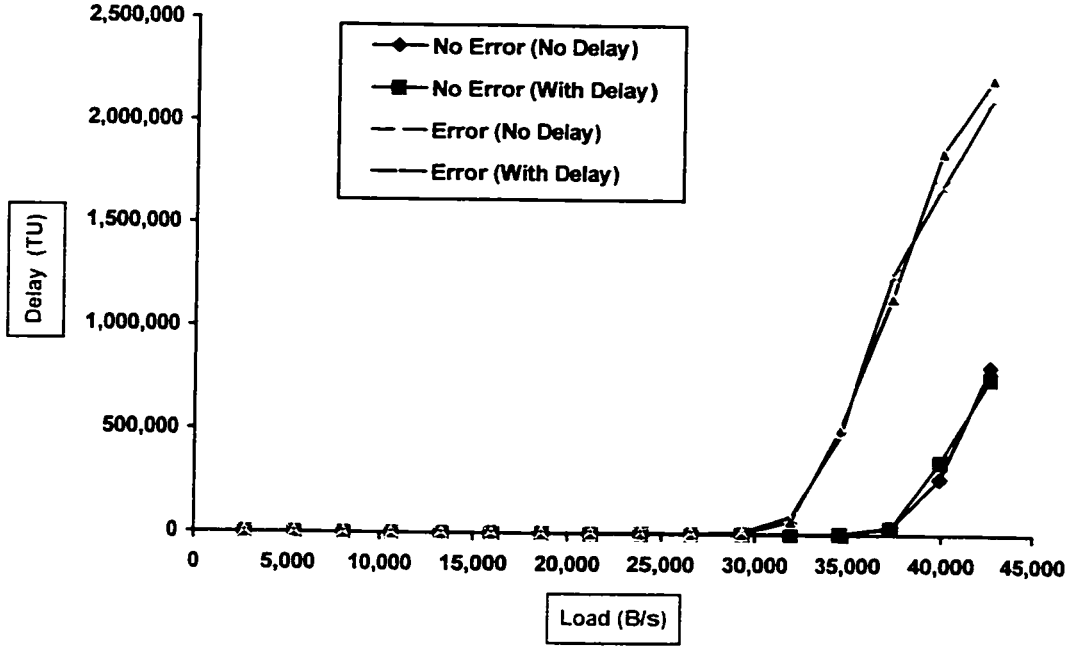


Figure 5.11: Average Message Delay vs. Offered Loads for 6 Byte Messages.

Message Length 128 Bytes (1024 bits) Message Copy delay 336 Time Unit	Average Message Delay (TU) TU = Time Unit 1 second = 10,000,000 TU			
Message Rate (B/s)	No Error		Error	
	No Delay	Delay	No Delay	Delay
36,680	1,700	2,670	1,770	2,730
79,616	1,840	2,760	1,850	2,770
119,170	2,000	2,880	1,980	2,880
158,730	2,200	3,020	2,160	3,000
198,295	2,450	3,170	2,360	3,170
237,829	2,770	3,510	2,660	3,420
277,356	3,210	3,860	3,100	3,790
316,989	3,830	4,390	3,780	4,820
356,447	4,770	5,330	6,790	5,770
396,040	6,470	6,960	15,000	12,000
435,819	10,100	10,200	352,000	259,000
475,483	23,300	26,400	1,450,000	1,700.000
515,091	274,000	275,000	2,420,000	2,290,000
554,353	1,090,000	1,100,000	3,580,000	3,470,000

Table 5.12: Average Message Delay vs. Offered Loads for 128 Byte Messages.

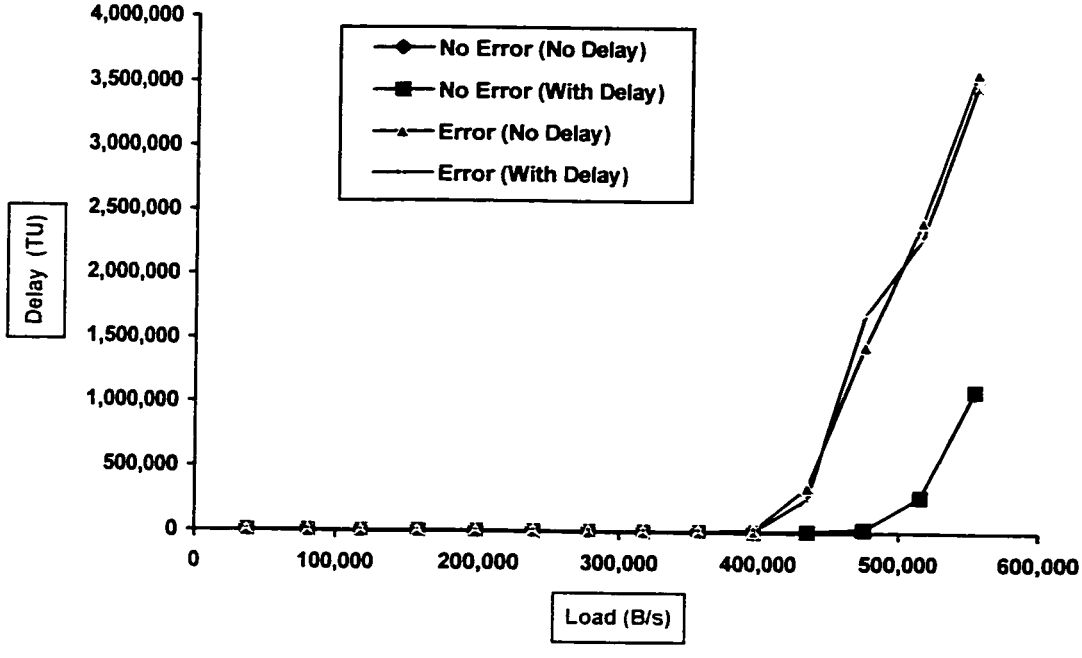


Figure 5.12: Average Message Delay vs. Offered Loads for 128 Byte Messages.

1 KByte (8192 bits) Message Copy delay 2694 Time Unit	Average Message Delay (TU) TU = Time Unit 1 second = 10,000,000 TU			
Message Rate (B/s)	No Error		Error	
	No Delay	Delay	No Delay	Delay
98,305	9,260	17,300	10,200	18,500
197,634	9,910	17,800	10,550	18,700
295,937	10,700	18,600	11,100	19,300
394,240	11,700	19,500	12,400	20,700
492,544	13,200	20,900	14,100	22,600
590,849	15,100	22,900	18,000	25,300
689,191	18,200	25,500	23,300	32,300
787,510	23,600	31,300	35,100	47,900
879,347	34,300	41,900	130,000	268,000
984,142	77,500	85,000	4,130,000	3,900,000
1,082,452	1,150,000	1,160,000	8,790,000	7,400,000
1,180,676	4,950,000	4,960,000	11,900,000	12,300,000
1,279,041	8,300,000	8,310,000	15,900,000	15,200,000

Table 5.13: Average Message Delay vs. Offered Loads for 1 KByte Messages.

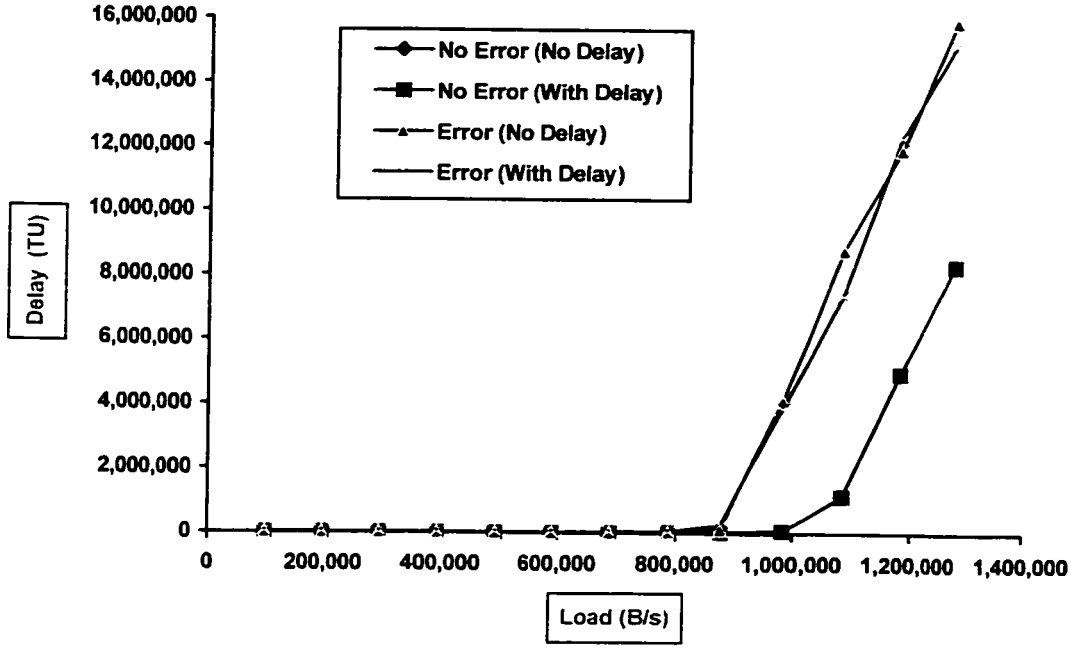


Figure 5.13: Average Message Delay vs. Offered Loads for 1 KByte Messages.

8 KByte (65,536 bits) Message Copy delay 21,559 Time Unit	Average Message Delay (TU) TU = Time Unit 1 second = 10,000,000 TU			
Message Rate (B/s)	No Error		Error	
	No Delay	Delay	No Delay	Delay
114,688	74,300	90,500	83,500	90,900
229,377	79,800	96,100	88,500	94,500
344,064	87,000	103,000	96,700	102,000
458,753	96,900	114,000	111,000	113,000
573,441	111,000	128,000	133,000	132,000
688,132	133,000	150,000	164,000	168,000
786,437	165,000	182,000	211,000	235,000
907,511	266,000	284,000	485,000	454,000
1,032,219	665,000	683,000	13,400,000	16,300,000
1,148,889	14,200,000	14,200,000	46,200,000	42,900,000
1,261,569	46,800,000	46,800,000	84,200,000	74,400,000
1,376,252	74,000,000	74,000,000	112,000,000	102,000,000
1,490,446	97,000,000	97,000,000	137,000,000	131,000,000

Table 5.14: Average Message Delay vs. Offered Loads for 8 KByte Messages.

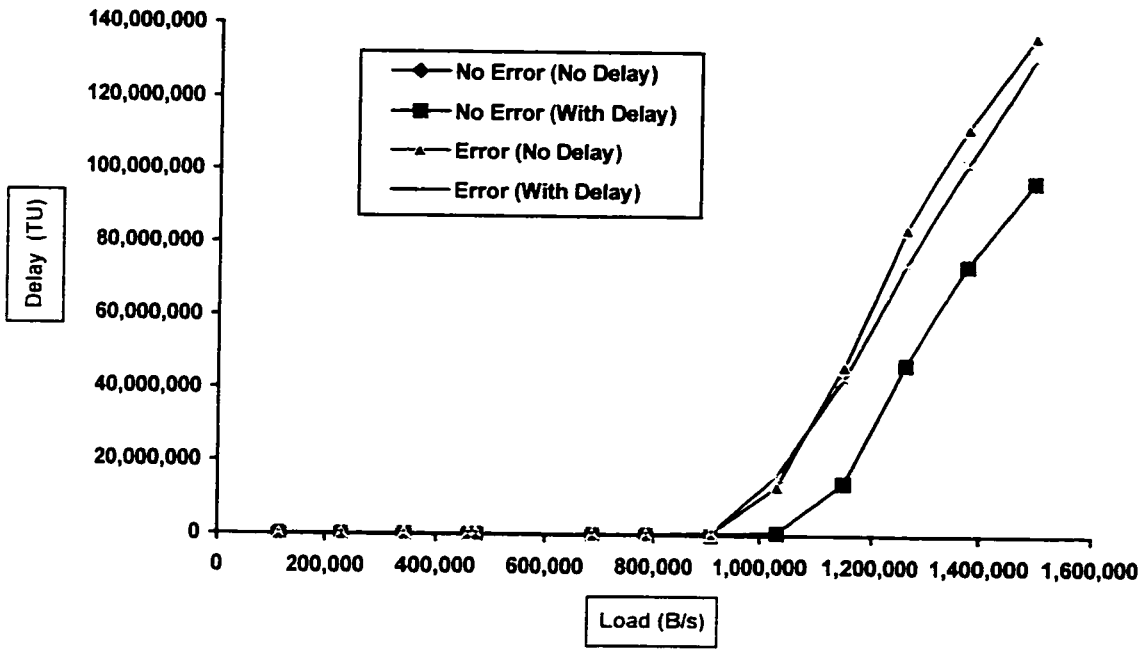


Figure 5.14: Average Message Delay vs. Offered Loads for 8 KByte Messages.

6. XTP IN LOW ERROR RATE ENVIRONMENT

As we approach the year 2000, fiber optic transmission media are rapidly becoming widely available for communication networks. The low bit error rates are a beneficial result of the move toward fiber optic networks. It is now more realistic to assume that the physical transmission of data is basically reliable.

In this chapter we will investigate the performance of XTP and the behavior of the error control in XTP for bulk transferring in a highly reliable environment, i.e., the environment with a very low bit error rate.

6.1 The XTP simulation plan in low error rate environment.

The XTP simulation is carried out on message size of 1MB with a low bit error rate of 1/4,000,000 (each bit of a packet has 1/4,000,000 chance of getting corrupted).

6.2 XTP simulation results in low error rate environment.

Table 6.1 and Figure 6.1 show the throughput of XTP Simulation for 1MB message (with delay) with Error Control (Selective Retransmission), and with Error Rate of 1/4,000,000 and 1/1,000,000.

Table 6.2 and Figure 6.2 show the throughput of XTP Simulation for 1MB message (with delay) with Error Control (Go-Back-N), and with Error Rate of 1/4,000,000 and 1/1,000,000.

From the figure 6.1 to 6.2, we see that the performance of XTP in the low error rate of environment (bit error rate of 1/4,000,000) is close to its performance in the environment with no errors. We also see that the performance of XTP in low error rate environment,

with **Go-Back-N** error control, is improved more significantly, in the environment with low error rate (bit error rate of 1/4,000,000), compared with the environment with higher error rate (bit error rate of 1/1,000,000).

Similarly results can be obtained for XTP simulation for 1MB messages (no delay).

With the bit error rates of fiber optic channels are on the order of 10^{-12} (1/1,000,000,000,000), we expect that the performance of XTP on fiber optic channels is very close to its performance in a no error environment.

Message Length 1 MByte (1024 KByte)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (err rate = 1/4,000,000)	Error (err rate = 1/1,000,000)
114,688	137,000	110,000	102,000
196,608	168,000	259,000	250,000
327,680	367,000	266,000	355,000
458,752	429,000	502,000	461,000
589,824	553,000	689,000	621,000
720,896	704,000	788,000	693,000
851,968	1,070,000	939,000	895,000
917,507	928,000	1,010,000	967,000
1,048,576	1,120,000	1,070,000	955,000
1,179,648	1,110,000	1,070,000	1,100,000
1,245,184	1,130,000	1,040,000	1,100,000
1,310,720	1,130,000	1,120,000	1,090,000
1,441,792	1,130,000	1,120,000	1,100,000
1,572,864	1,130,000	1,120,000	1,100,000
1,703,936	1,130,000	1,120,000	1,100,000
1,835,008	1,130,000	1,120,000	1,100,000

Table 6.1: Throughput vs. Offered loads for 1 MByte Messages (with delay) with error control (**Selective Retransmission**), and different bit error rates.

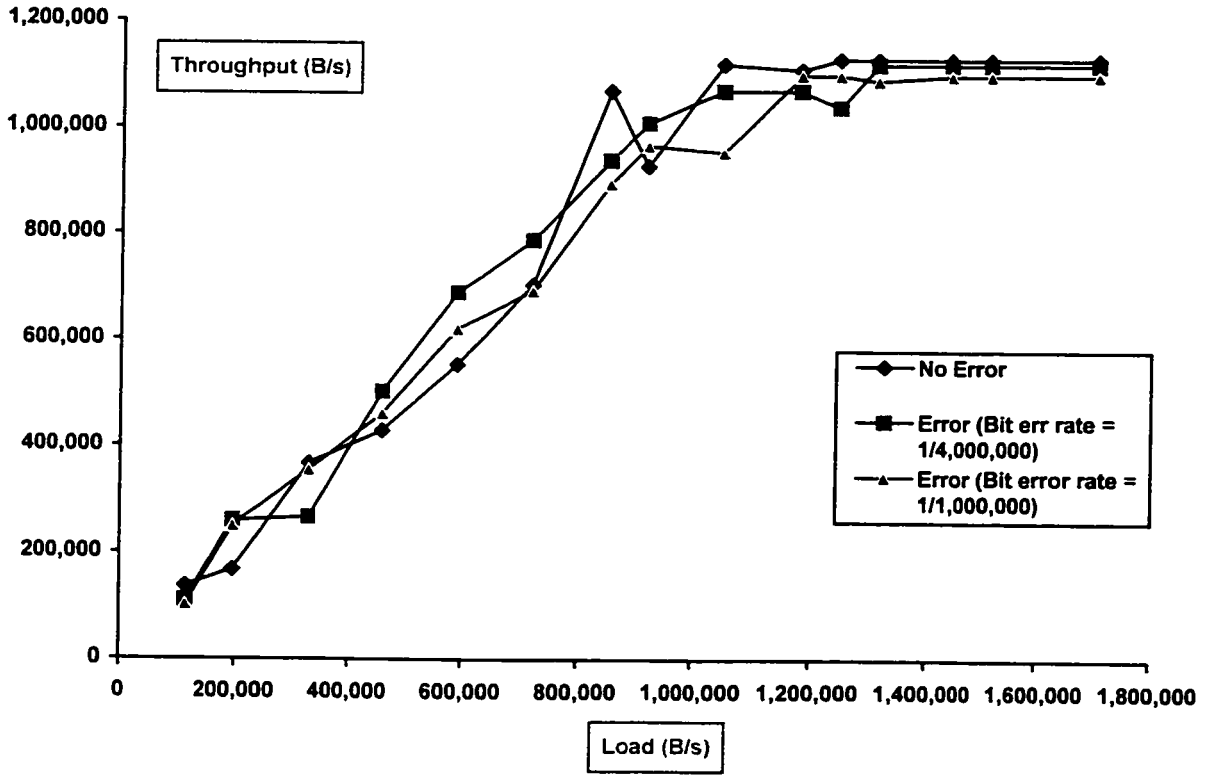


Figure 6.1: Throughput vs. Offered loads for 1 MByte Messages (with delay) with error control (**Selective Retransmission**), and different bit error rates.

Message Length 1 MByte (1024 KByte)	Effective Throughput (B/s)		
Message Rate (B/s)	No Error	Error (err rate = 1/4,000,000)	Error (err rate = 1/1,000,000)
114,688	137,000	97,400	118,000
196,608	168,000	195,000	259,000
327,680	367,000	338,000	328,000
458,752	429,000	428,000	441,000
589,824	553,000	590,000	530,000
720,896	704,000	792,000	679,000
851,968	1,070,000	843,000	746,000
917,507	928,000	929,000	947,000
1,048,576	1,120,000	974,000	907,000
1,179,648	1,110,000	1,080,000	954,000
1,245,184	1,130,000	1,080,000	949,000
1,310,720	1,130,000	1,080,000	976,000
1,441,792	1,130,000	1,060,000	982,000
1,572,864	1,130,000	1,080,000	983,000
1,703,936	1,130,000	1,080,000	973,000
1,835,008	1,130,000	1,080,000	981,000

Table 6.2: Throughput vs. Offered loads for 1 MByte Messages (with delay) with error control (**Go-Back-N**), and different bit error rates.

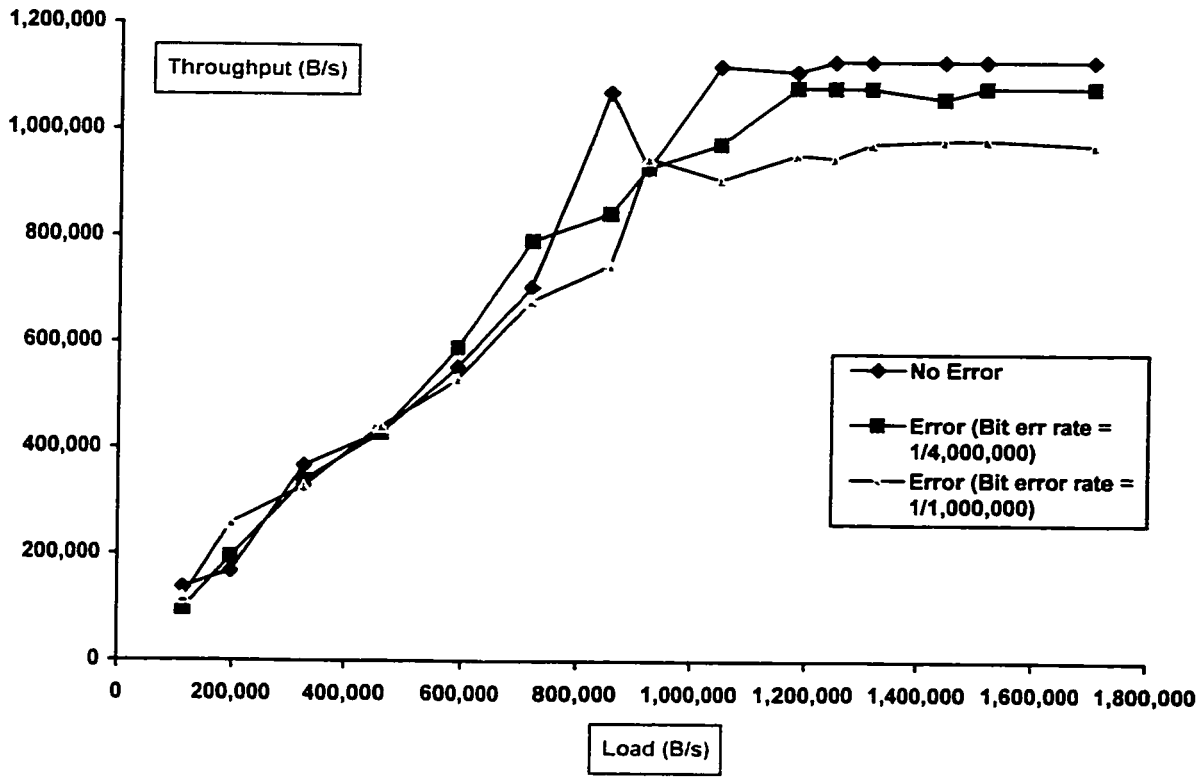


Figure 6.2: Throughput vs. Offered loads for 1 MByte Messages (with delay) with error control (**Go-Back-N**), and different bit error rates.

6.3 The behavior XTP's error control in a low error rate environment

In this section we will study the behavior of XTP's error control (selective retransmission) in a low error rate environment.

We start this study by looking at a XTP simulation run for 1MB messages (with delay) with load number (message arriving rate) that is equal to 100% of the expected maximum throughput (1,310,720 B/s).

At the simulation time of 8,101,571 (a time unit of a XTP simulation run is equal to 10^{-7} s), the data packet with sequence number of 868,500 is damaged when it is transmitted by the Transmitter. As a result, the Receiver (which is expecting to receive the next data packet with sequence number of 868,500) will not be able to receive this packet. When the Receiver is receiving the next data packet with sequence number of 870,000, the Receiver Error Control will recognize that the data packet with sequence number 868,500 has been lost, and it will update the value of *spans* and the *nspans* from 0 to 1.

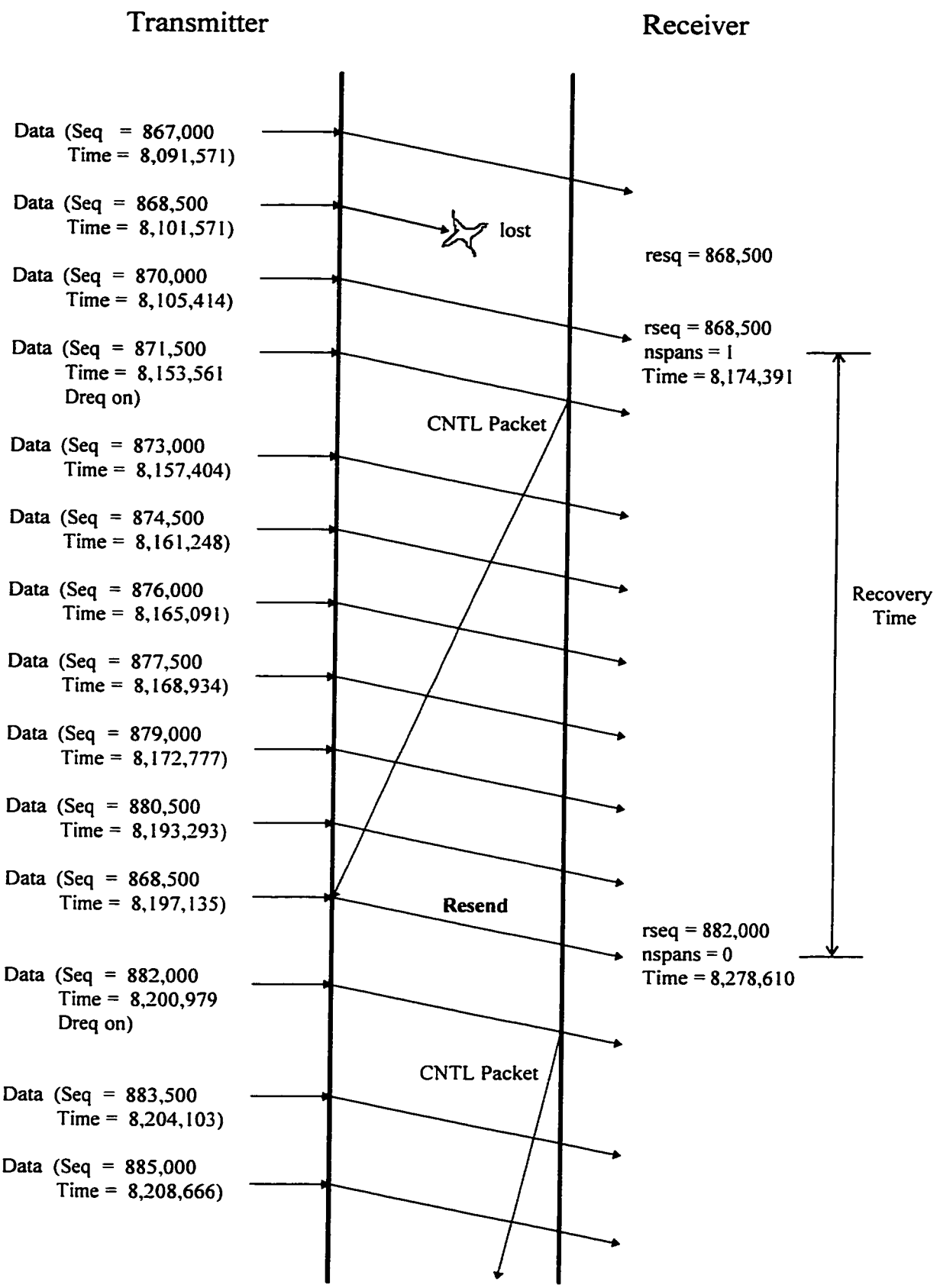
Upon receiving the data packet with the sequence number 871,500, with the DREQ bit on, the Receiver will send back a CNTL packet that contains its current status (*rseq*, *nspans*, *spans*, etc.) to the Transmitter. When the Transmitter receives this CNTL packet at simulation time of 8,197,135, it will figure out that the data packet with the sequence number 868,500 has not been received by the Receiver, and the Transmitter will resend this packet.

Upon Receiving the data packet with the sequence number 868,500 the Receiver will update the values of *rseq*, *nspans*, *spans*, etc.

At the simulation time of 8,313,042 the Transmitter receives the next CNTL packet sent by the receiver. At this time, the Transmitter recognizes that the resent data packet (with sequence number of 868,500) has been received by the receiver. The gap has been filled (*nspans* = 0).

The Transmitter continues to transmit a number of data packets before the next error occurs (a data packet is damaged). When this happens, the whole error recovery process is repeated again.

Figure 6.3 and 6.4 show the flow of data and CNTL packets discussed above.


Transmitter
Receiver
Data (Seq = 867,000
Time = 8,091,571)
Data (Seq = 868,500
Time = 8,101,571)
lost
Data (Seq = 870,000
Time = 8,105,414)
resq = 868,500
Data (Seq = 871,500
Time = 8,153,561
Dreq on)
rseq = 868,500
nspans = 1
Time = 8,174,391
CNTL Packet
Data (Seq = 873,000
Time = 8,157,404)
Data (Seq = 874,500
Time = 8,161,248)
Data (Seq = 876,000
Time = 8,165,091)
Recovery
Time
Data (Seq = 877,500
Time = 8,168,934)
Data (Seq = 879,000
Time = 8,172,777)
Data (Seq = 880,500
Time = 8,193,293)
Data (Seq = 868,500
Time = 8,197,135)
Resend
rseq = 882,000
nspans = 0
Time = 8,278,610
Data (Seq = 882,000
Time = 8,200,979
Dreq on)
CNTL Packet
Data (Seq = 883,500
Time = 8,204,103)
Data (Seq = 885,000
Time = 8,208,666)

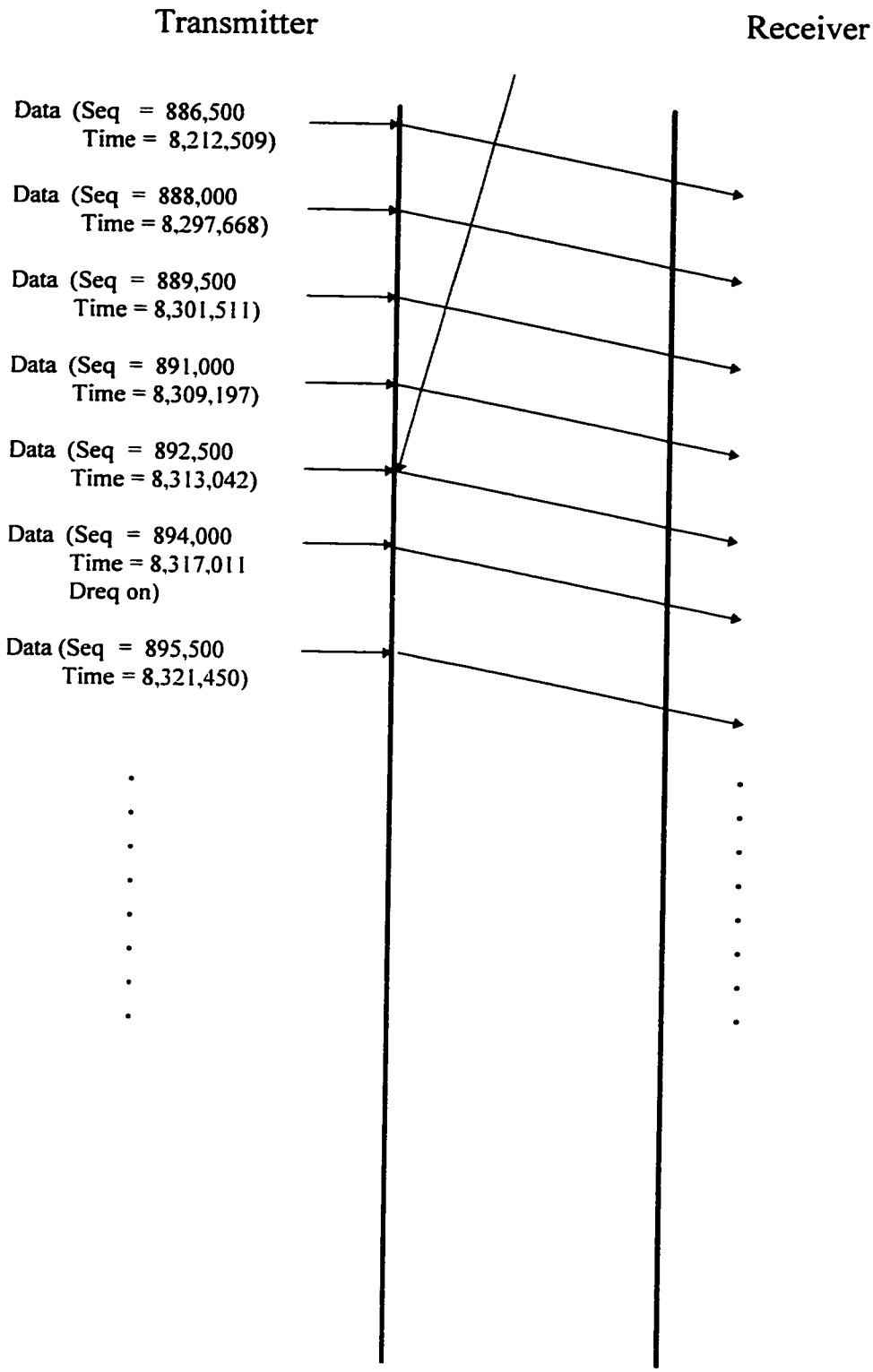


Figure 6.3: Time diagram

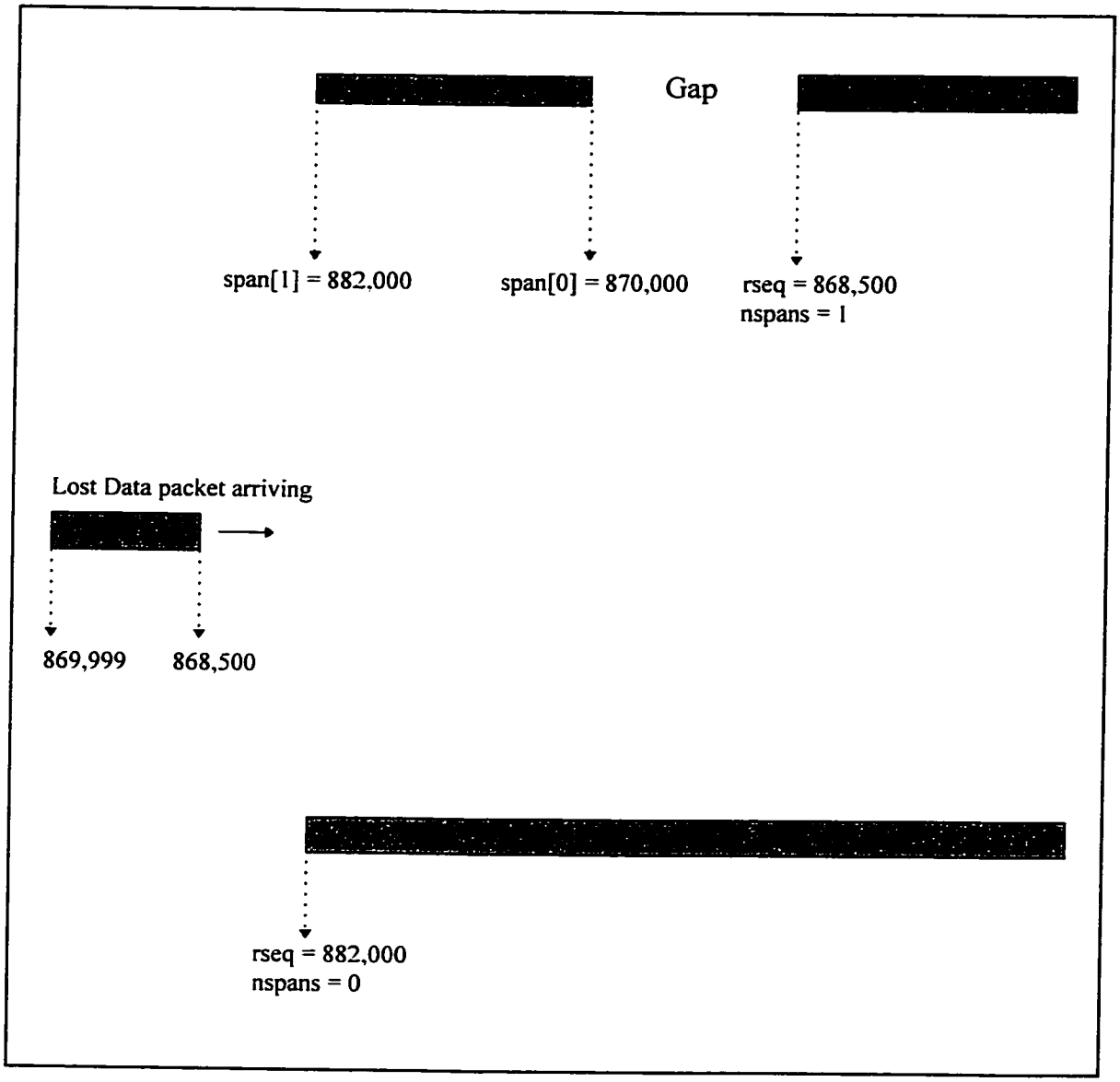


Figure 6.4: Receiver's specification of errors.

Figure 6.3 shows that after transmitting the data packet (with sequence number of 868,500) that will be corrupted, the Transmitter will transmit 8 more data packets (sequence number 870,000 to 880,500) before it receives the CNTL packet sent back by the Receiver, to inform it that there was a data gap at the Receiver end. After resending the lost data packet (with sequence number of 868,500), the Transmitter will transmit 8 more data packets before it receives the next CNTL packet sent back by the Receiver to inform it that the data gap has been filled.

6.4 The pattern of packets

We set up the XTP simulation runs with message size of 1MB (delay and no delay), with bit error rate of 1/4,000,000 and with different loads. We recorded the number of data packets and CNTL packets sent and received by the Transmitter and Receiver, when a data packet is damaged. Based on the results of these simulation runs, we come up to the following pattern of data and CNTL packets sent and received by the Transmitter, after a data packet is damaged:

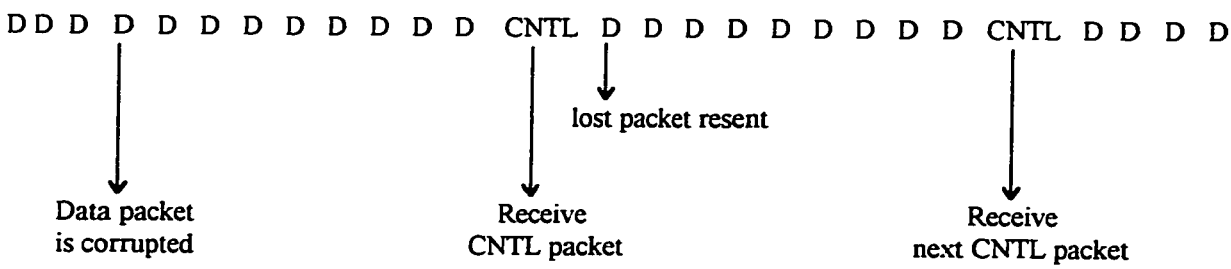


Figure 6.5: The pattern of data and CNTL packets sent and received by the Transmitter.

Figure 6.5 shows that **on the average**, when a data packet is damaged, the Transmitter will transmit 8 more data packets before it receives a CNTL packet sent back by the receiver to inform it that there was a data gap at the Receiver end. After resending the lost data packet, the Transmitter will transmit an average of 8 more data packets before it receives the

next CNTL packet sent back by the Receiver to inform it that the data gap had been filled at the Receiver end. With low bit error rate of 1/4,000,000, the Transmitter will continue to transmit a number of data packets with no damaged packet for a period of time. When the next data packet is damaged, the Transmitter will follow a similar pattern of packets described in figure 6.5.

As described in chapter 5 (and in chapter 7), the **number of collisions** at a station of XTP simulation runs with large message size (for example, 1MB), will be significant higher compared with XTP simulation runs with smaller message sizes, and with the same loads. A CNTL packet sent back by the Receiver in XTP simulation runs with large message size of 1MB, will have higher chance of colliding with a data packet arriving at the Receiver. As a result, in XTP simulation runs with large message size of 1MB, it will take the Transmitter, on the average, longer time to receive a CNTL packet sent back by the Receiver to inform it if there was a gap at the Receiver end.

Table 6.3 shows all the patterns of packets for XTP simulation runs with message size of 1MB (with delay), with different loads, and with bit error rate of 1/4,000,000.

For similar XTP simulation runs with smaller message size (for example, 8KB), the patterns of packets will be shorter, as expected.

Message Rate (B/s)	The number of data packets the Transmitter continues to send after a data packet is damaged, and before the Transmitter receives the next CNTL packet.	The number of data packets the Transmitter continues to send after a damaged is resent, and before the Transmitter receives the next CNTL packet.
114,688	7	7
196,608	7	7
327,680	8	8
458,752	8	8
589,824	8	8
720,896	8	8
851,968	8	8
917,507	8	9
1,048,576	9	9
1,179,648	9	9
1,245,184	9	9
1,310,720	8	8
1,441,792	8	8
1,572,864	8	8
1,703,936	8	8
1,835,008	8	8

Table 6.3: The patterns of data packets sent by the Transmitter after a data packet is damaged.

6.5 The Recovery Time.

We also measure the **average recovery time** for the XTP simulation runs of table 6.3. (as shown in figure 6.3, the recovery time is the time the Receiver takes to recover a lost data packet). Table 6.4 shows the average recovery time for the XTP simulation runs of message size 1MB (delay and nodelay), with bit error rate of 1/4,000,000.

Figure 6.6 shows the distribution of the average recovery time for the XTP simulation runs of 1MB message (with delay), and with bit error rate of 1/4,000,000.

	Average Recovery Time (ms)
Delay	18
No Delay	70

Table 6.4: Transmitter's average recovery time for XTP simulation runs of 1MB messages (delay and no delay), with bit error rate of 1/ 4,000,000, and with Selective Retransmission Error Control .

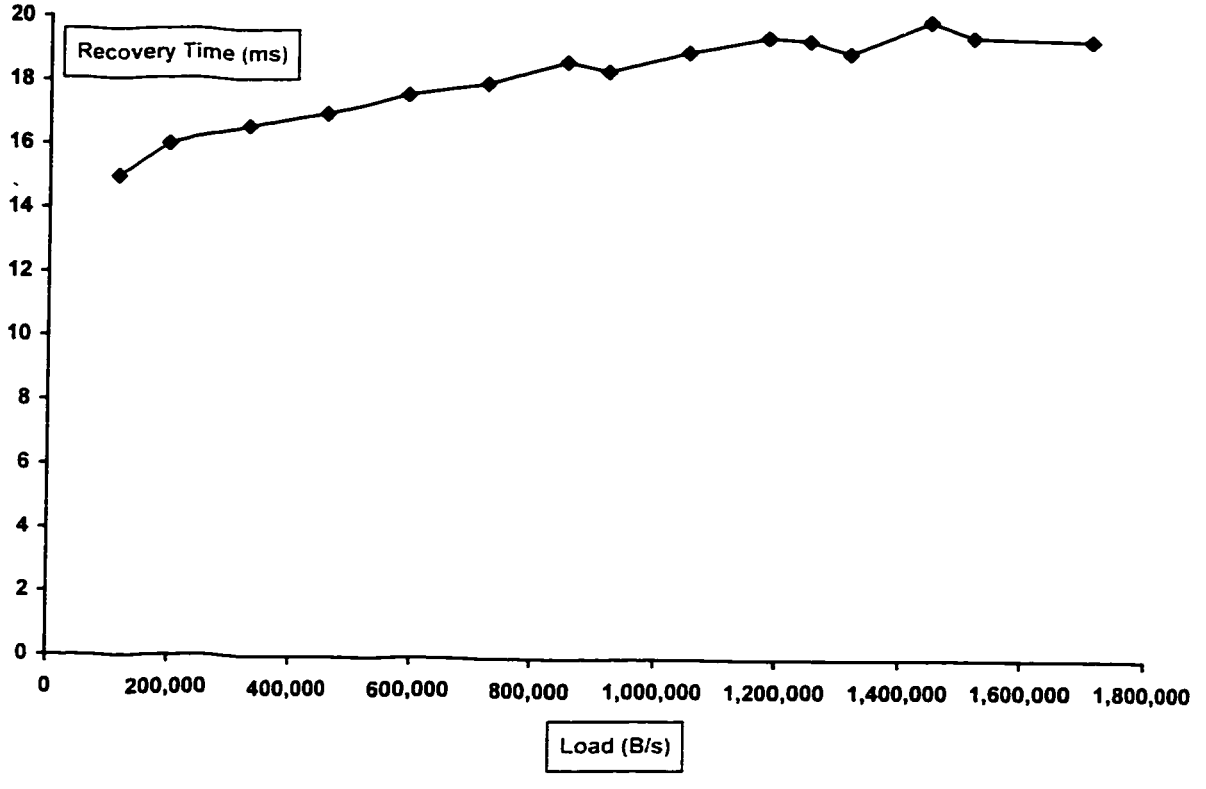


Figure 6.6: The distribution of Average Recovery Time of XTP simulation runs of 1MB messages (with delay) with bit error rate of 1/ 4,000,000 and Selective Retransmission Error Control.

In the next chapter, we will discuss an enhanced backoff algorithm, that can reduce the number of collisions at a station during an XTP simulation run. This may lead to the decreasing of the average recovery time at the Transmitter station, and the improvement of the XTP throughput.

7. THE ETHERNET CAPTURE EFFECT

Ethernets have been in widespread use for over a decade now. It has been the core technology enabling distributed computing, and network for the desktop, server and backbone over the years. It has met the performance needs of applications and systems, and has been a robust and inexpensive local area network so that computer systems have begun to have an interface to it as a default. The algorithms for controlling access to the Ethernet networks have been examined by a large number of researchers over the years. One of these algorithms is the well known Binary Exponential Backoff (BEB) algorithm, which is used in Ethernet/802.3 network, for collision resolution.

If two stations both detect an idle Ethernet and begin transmitting at the same time, a collision will occur. After the first collision, each station waits either 0 or 1 slot time (a slot time is 512 bit time, e.g., 51.2 μseconds for 10 Mbps system, and 5.12 μseconds for 100 Mbps system) before trying again. If two stations collide and each picks the same random number, they will collide again. After the second collision, each one picks either 0, 1, 2 or 3 at random and waits that number of slot times. If the third collision occurs (the probability of this happening is 0.25), then the next time the number of slot times to wait is chosen at random from the interval 0 to $2^3 - 1$.

In general, after n collisions, a random number between 0 and $2^n - 1$ is chosen, and that number of slot times is skipped. However, after 10 collisions have been reached, the randomization interval is frozen at a maximum of 1023 time units ($2^{10} - 1$). After 16 collisions, the retransmission is aborted on any given packet. Further recovery is up to higher layers.

The BEB algorithm was chosen to dynamically adapt to the number of stations trying to send. If the randomization intervals for all collisions was 1023, the chance of the stations colliding for a second time would be negligible, but the average wait after a collision would

be hundreds of time units, introducing significant delay. By having the randomization interval grow exponentially as more and more collisions occur, the BEB algorithm ensures a low delay when only a few stations collide, but also ensures that the collision is resolved in a reasonable interval when many stations collide. However, BEB algorithm can lead to the problem of Ethernet Capture Effect, which is discussed in the next section.

7.1 The Ethernet Capture Effect

The Ethernet Capture Effect is the behavior wherein under high load, one station is able to hold on to the channel to transmit packets consecutively, in spite of other station(s) contending for access. This is particular acute in the case of a 2-node network, with one station receiving an unfair share of the channel bandwidth over a transient period. The number of packets consecutively transmitted by the node capturing the channel can potentially be hundreds of packets or more, if the station has this large number of appropriate size packets to transmit. The Capture Effect is due to the *transient unfairness* of the standard BEB algorithm used in the Ethernet/802.3 network. The degree of transient unfairness is severe for a small number of contending stations and it reduces quickly as the number of active stations increases. It also has severe performance repercussions even with protocols using window flow control with modest maximum window size of 32K or 64K. The Capture Effect results in the channel being unnecessary idle, thus reducing the overall throughput achieved by the applications. The *transient unfairness* also results in the access latency seen by a station having substantially increased variability, which is bad for emerging applications such as multimedia networking.

Ramakrishnan introduced the *Capture Avoidance Binary Exponential Backoff* algorithm (CABEB) [Ramakrishnan94], which he states has all the properties needed to overcome the performance degradation without violating in any way the spirit of the Ethernet/802.3 standards for access to the network. The CABEB is an extension of the standard BEB, where it uses the standard BEB for all cases of collision resolution except for one. The CABEB for collision resolution minimizes the occurrence of capture by a station by

using an enhanced backoff algorithm when a collision occurs. The CABEB algorithm is discussed in the next section.

7.2 The CABEB Algorithm

The CABEB algorithm executes the standard BEB for all the collision cases except for one special case. If a station transmits a first packet and begins to transmit a second packet and the station has not received a collision or a packet between the first packet and the beginning of the second packet, we call this an *uninterrupted consecutive transmit*. The CABEB algorithm processes this uninterrupted consecutive transmit using an enhanced BEB algorithm to solve the Capture Effect problem. The CABEB algorithm is compliant with the Ethernet/802.3 standard.

There are several back to back cases of interest. The first case is the uninterrupted consecutive transmit, where a station transmits a second packet after it has successfully transmitted a first packet and there were no other stations contending for the channel prior to the beginning of the second packet. In the uninterrupted consecutive transmit case, the second packet could encounter a collision. So, the conditions required for this case are a station successfully transmitting a first packet, the channel is idle (i.e., no packet is received, or collision encountered) for an arbitrary period of time after the first packet, and the station begins to transmit a second packet.

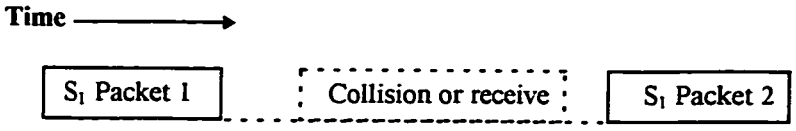
The second case of interest is where a station transmits a first packet, then it receives one or more packets or collisions, and then the station transmits a second packet. We call this case an *interrupted consecutive transmit*. For an interrupted consecutive transmit, the station of interest (i.e., the one that just transmitted a packet) is not involved in the collision.

The third case of interest here is when the second packet of an uninterrupted consecutive transmit is involved in a collision and the station wins the collision resolution to complete the transmission of the second packet. An uninterrupted consecutive transmit is

the first portion of a captured transmit, where the second packet of the uninterrupted consecutive transmit is involved in a collision. We will call this case a *captured transmit*. When a station does consecutive captured transmits for multiple packets, we say that the station has captured the channel. A captured transmit occurs when the station winning the collision resolution transmits the second packet. Figure 7.1 illustrates these cases.

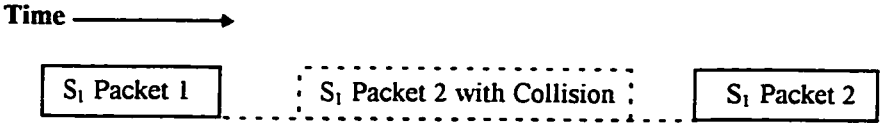
1. Uninterrupted consecutive transmit

2. Interrupted consecutive transmit



Note: Station S_1 is not involved in the collision.

3. Captured transmit



Note: Station S_1 is involved in the collision.

Figure 7.1: Examples of Transmit Cases.

The CABEB algorithm solves the Capture Effect problem by minimizing the occurrence of captured transmits. This is archived by using an enhanced backoff algorithm when a collision occurs on the second packet of an uninterrupted consecutive transmit. For

an interrupted consecutive transmit, it uses the standard BEB algorithm for collision resolution.

The enhanced backoff algorithm can be described as follows: When transmitting a second packet of an uninterrupted consecutive transmit, the station takes a backoff of 2 slot times on the first collision. If the other colliding station is transmitting a fresh packet (i.e., one that has not experiencing any collision), that station will draw a backoff of 0 or 1 slot time (according to the BEB) and hence its packet is guaranteed to be transmitted.

The CABEB algorithm [Ramakrishnan94] is described in more detail on the next page.

CABEB Algorithm [Ramakrishnan94]

n = collision attempts (0-15)

r = standard BEB uniform distributed random number

k = BEB backoff range variable

backoff = the number of slot times to backoff

For collision resolution, the following procedure determines the backoff time for CABEB algorithm:

For an uninterrupted consecutive transmission:

If n = 1 then backoff = 2;

if n = 2 then backoff = 0;

if n > 2 then backoff = r;

where $0 <= r < 2^k$, k = min (n,10)

For an interrupted consecutive transmission:

backoff = r; where $0 <= r < 2^k$, k = min (n, 10)

Thus, The CABEB algorithm guarantees that a capture transmit does not occur when both the colliding packets are expecting their first collision. After the backoff of 2 slot times, the station is ready to retransmit the packet. If the packet collides for the second time, the station draws a backoff of 0 slot times. Thus, the station will retransmit immediately. The selection of 0 slot time on a second collision allows the station to have a higher probability of winning the second collision. If the same packet experiences a third or subsequent collision, the station uses the standard BEB for collision resolution.

The CABEB algorithm is not meant to be effective when one of the packets involved in the collision has advanced beyond its second collision attempt. In this case, the CABEB's backoff 2 slot times does not help because the other station may be backing off a random number of slot times based on the collision attempt and the standard BEB. For those cases where a collided packet has advanced beyond its second collision attempt, the CABEB behavior is similar to that of the standard BEB. This is important as we would like the performance of the CABEB algorithm to be no worse than the standard BEB algorithm when the number of stations in the network is large.

7.3 The Performance of XTP with CABEB Algorithm

As discussed in section 5.5, the Ethernet Capture Effect has some impact on the performance of XTP simulation, especially when the loads offered reaches a certain threshold in XTP simulation runs with large message size of 1MB. As a result, the backoff algorithm, which is used when a collision has occurred, plays a large role in the performance of XTP simulation [Chung93].

To overcome the Ethernet Capture Effect in XTP simulation, we implemented the CABEB algorithm [Ramakrishnan94] in the XTP simulation program.

In this section, we consider the comparison between the CABEB algorithm and the BEB algorithm for XTP simulation with the **message size of 1 MB (with delay), and with errors present**.

The graph in figure 7.2 shows the performance of XTP for 1 MB messages, with error control (selective retransmission mechanism) using either the BEB algorithm, or the CABEB algorithm. The graph in figure 7.3 shows the number of collisions at a station for the XTP simulation in figure 7.2.

From figure 7.2, we find that the throughput of XTP has been increased between 2 - 10% (depend on the load numbers) with the CABEB algorithm. Figure 7.3 shows that the number of collisions at a station of the simulation has been reduced between 12 - 20 %.

Similarly results can be obtained from figure 7.4 and figure 7.5, when using the Go-Back-N mechanism for error control, instead of using the Selective Retransmission mechanism. We observe that the XTP throughput (figure 7.4) using CABEB algorithm was increased by more than 10% for the test cases with load numbers greater than 70% of the maximum expected throughput. This can be explained by the large difference in number of collisions between CABEB algorithm and BEB algorithm for test cases with high load numbers.

Similarly results can be obtained for XTP simulation for 1MB messages (**no delay**).

The CABEB enhanced the standard BEB algorithm by modifying the early stages of collision resolution (i.e., the first and second collision of a packet). As a result, the enhanced algorithm is most effective for a network with small number of stations, where the enhanced algorithm avoids a given packet from advancing its number of collisions beyond two. This is guaranteed for a two node case, as the algorithm allows the two stations to alternate their transmit. In addition, the CABEB maintains the same mean as the standard BEB for the retransmission backoff time for multiple collisions on any given packet. This allows the algorithm to be compliant to the IEEE 802.3 standard. Another added advantage of

maintaining the same mean value for multiple collisions is that the behavior of CABEB converges to the standard BEB for network with a large number of stations, at high load when all stations are active.

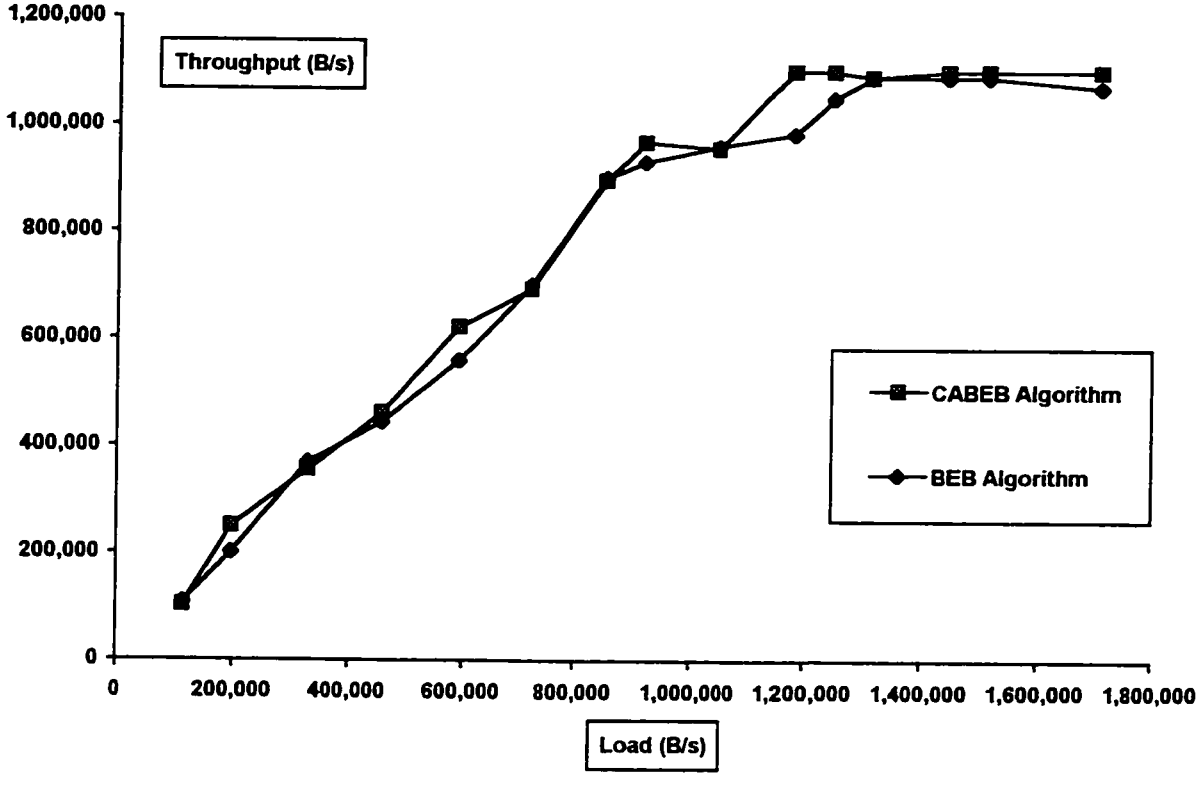


Figure 7.2: Throughput vs. Offered loads for 1 MB messages (with delay), with Error Control (Selective Retransmission).

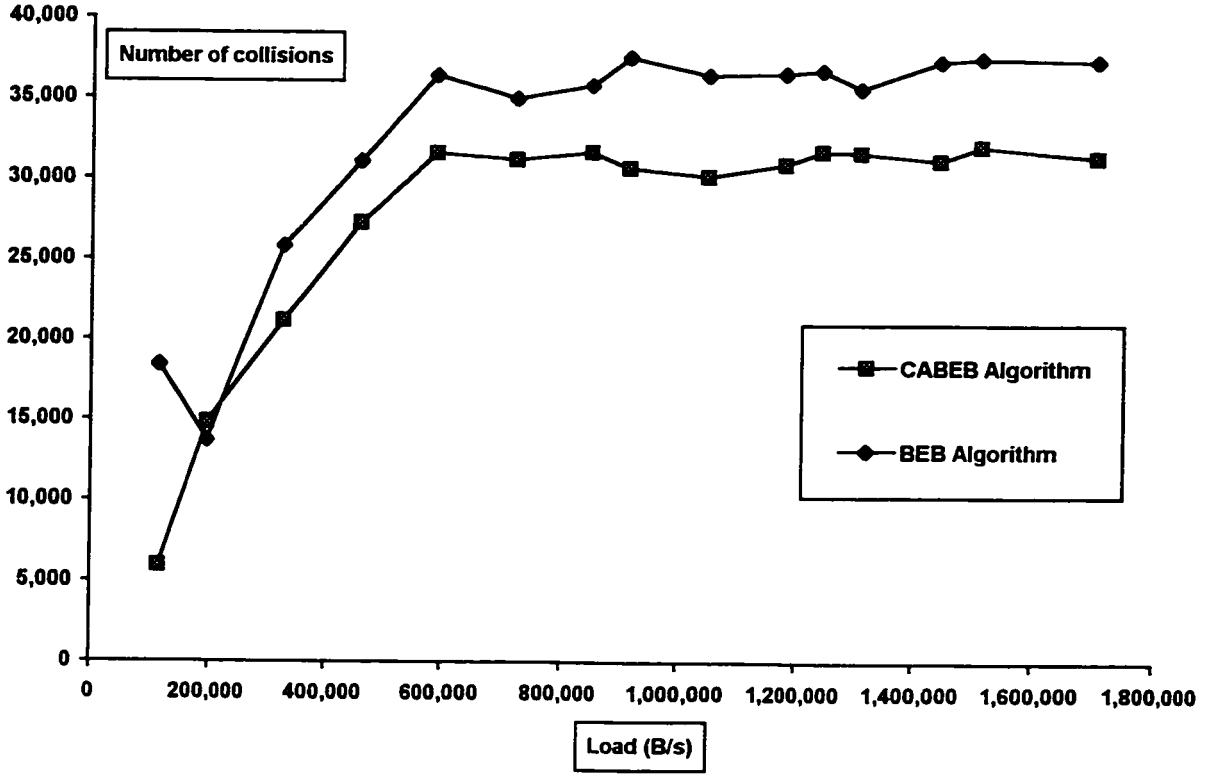


Figure 7.3: The number of collisions vs. Offered loads for 1 MB messages (with delay), with Error Control (Selective Retransmission).

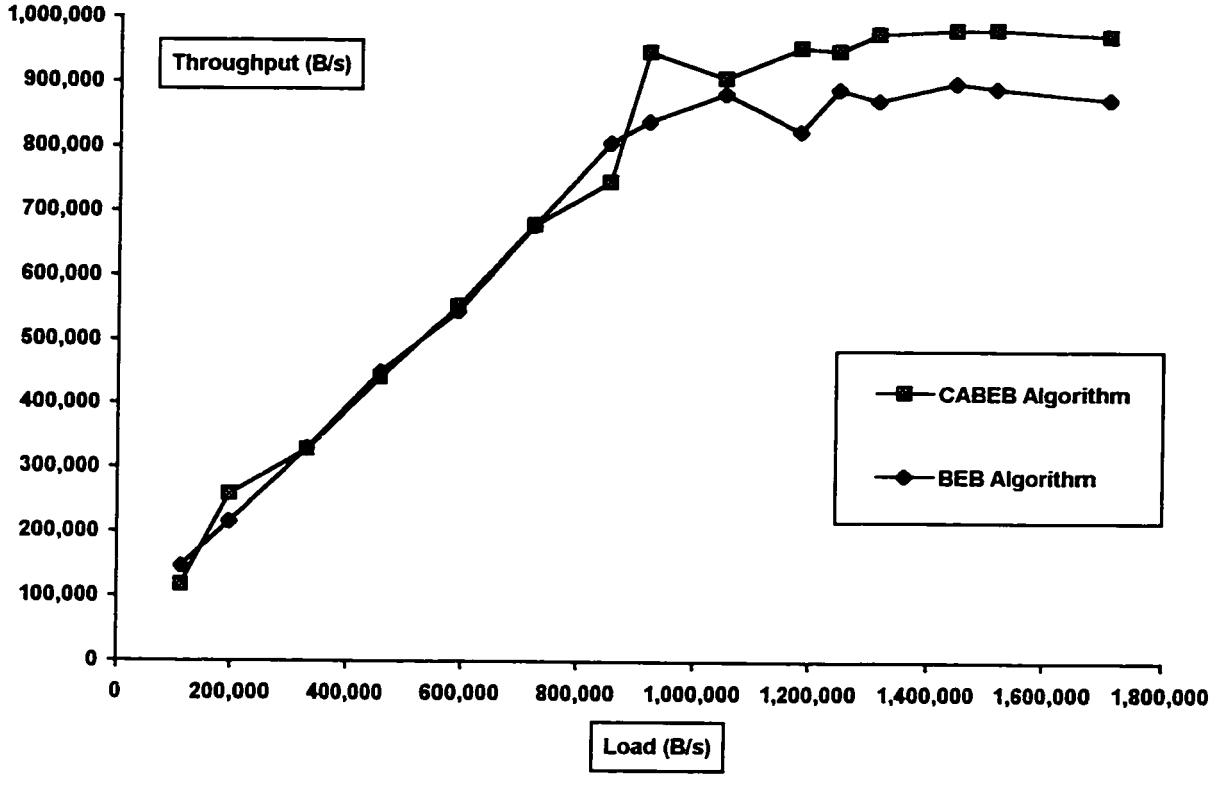


Figure 7.4: Throughput vs. Offered loads for 1 MB messages (with delay), with Error Control (Go-Back-N).

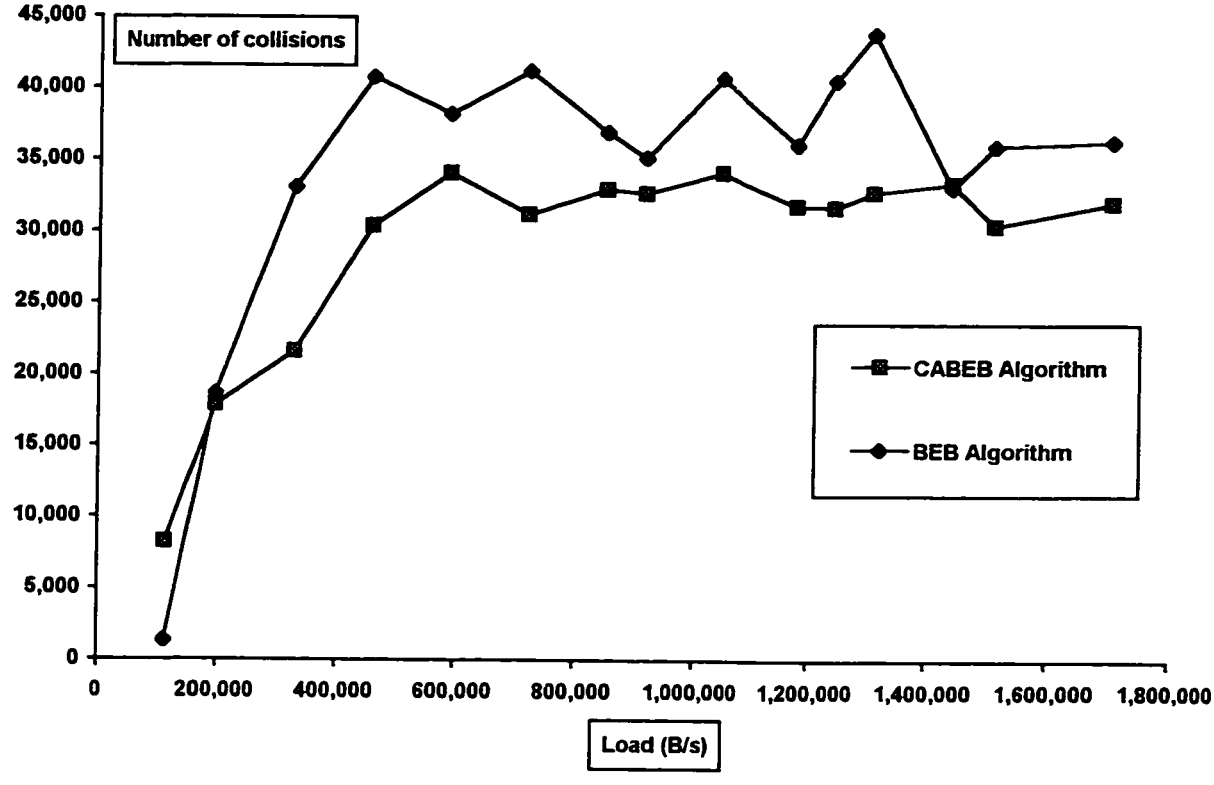


Figure 7.5: The number of collisions vs. Offered loads for 1 MB messages (with delay), with Error Control (Go-Back-N).

8. CONCLUSION

8.1 Thesis goals

The object of this thesis was to study the performance the XTP with error control, using SMURPH as the simulation tool. We benefit from the works done by [Chen89] and [Chung93], and the new formal specification of XTP [XTPForum95]. However, it still took us a lot of time for implementing the XTP simulation in SMURPH. We hope that the new versions of SMURPH in the future, will have better debugging tools. This may save a lot of time. The simulation runs also consume a lot of time, for example, a typical XTP simulation run for the message size of 1MB will take at least 45 minutes to complete.

8.2 XTP Performance

From the XTP simulation runs with high bit error rates, we find that XTP with error control is capable of providing excellent performance. In the environments with low bit error rates, XTP has the performance that is very close to its performance in the no-error environments. With bit error rates of fiber optic channels on the order of 10^{-12} (1/1,000,000,000,000), we expect that the performance of XTP on fiber optic channels will be excellent.

By carrying out many simulation runs with low bit error rates, the behavior of the XTP error controls was also studied in more detail.

[Chen89] mentioned that the copy and checksum delays play an important role in the performance of XTP. From the results of the XTP simulation runs with error control, we can conclude that these delays are not as important as the performance of the underlying Ethernet. Ethernet was found to be a bottleneck, limiting the transfer of data. The delay resulting from the collisions occurring in the underlying network was, occasionally, 50 times higher than the copy delay. A good backoff algorithm should be able to decrease the number of collisions, and thus increase the performance of XTP.

The CABEB algorithm (discussed in chapter 7) enhances the standard BEB algorithm, and reduces the number of collisions in the underlying network. As a result the number of times that the protocol entering the synchronizing handshake is also reduced.

The performance of XTP simulation runs, with error control, using the CABEB algorithm, is improved significantly. For XTP simulation runs, using Go-Back-N error control, and CABEB algorithm, at high load numbers, the number of collisions are reduced by more than 30% and the performance of XTP is improved by more than 10%. The CABEB algorithm solves the problem of Ethernet Capture Effect and it is compliant to the IEEE 802.3 standard.

At a conclusion, we believe that XTP, which is designed to fix some deficiencies in TCP, and to provide high efficiency in transport service, has an excellent performance, even in the high error rate environment, and with large message sizes. The excellent performance of XTP is due to the excellent design decisions made for the error control mechanisms.

8.3 Future work

The current XTP simulator in SMURPH can be extended by increasing the number of stations using the channels. The expected decreasing in the performance of XTP should be investigated. The performance of XTP with both CABEB and BEB algorithms should also be studied.

The next logical step is to study the performance of XTP with error control in a multicast environment, since XTP multicasting is now formally specified in the XTP specification, version 4.0 [XTPForum95]. The study of XTP performance in the multicast environment is currently carried out by Mr. Kun Guo, a graduate student, under the supervision of Dr. J.W. Atwood.

It would also be interesting to study the performance of XTP in other real-life network configurations such as token ring, token bus, or FDDI. Simulation of XTP with FDDI configuration in SMURPH may not be possible because the resources required might exceed the ones available. More packets will be transmitted, and thus, more state information will have to be kept at each station of the simulation.

The XTP simulation, with error control, is now implemented in SMURPH, which is an **object-oriented** configurable simulator, and thus, make the design and implementation of some of the works descried above, possible for the future.

REFERENCES

[Chen89] Chen, J. Design and Validation of an XTP Simulator. M.Comp.Sc Thesis, Concordia University, Department of Computer Science, 1989.

[Chung93] Chung Kam Chung, G. Design and Validation of Error Control in an XTP Simulator. M.Comp.Sc Major Technical Report, Concordia University, Department of Computer Science, 1993.

[Danthine80] Danthine, A. Protocol Representation with Finite-State Models. IEEE Transactions on Communications, VOL. COM-28, NO.4, April 1980, pages 632-643.

[Gburzynski91a] Gburzynski, P. and P. Rudnicki. LANSF: A Protocol Modeling Environment and its Implementation. Software - Practice and Experience, 21, 3, 51 - 76, 1991.

[Gburzynski91b] Gburzynski, P. and P. Rudnicki. An Overview of SMURPH: an object oriented configurable simulator for low-level communication protocols. University of Alberta, Department of Computer Science, 1992.

[Holzmann91] Holzmann, J. Design and Validation of Computer Protocols. Prentice Hall, 1991.

[Institut93] Institut National des Telecommunications, Evry, France. Formal specification, validation and performance evaluation of the Xpress Transport Protocol (XTP), 1993.

[PEI92a] Protocol Engines Incorporated. XTP Protocol Definition, Revision 3.6, 1992.

[Ramakrishnan94] Ramakrishnan, K., and Yang, H. The Ethernet Capture Effect: Analysis and Solution. Digital Equipment Corporation, Littleton, MA, 1994.

[Stallings91] Stallings, W. Data and Computer Communications. Third edition, Macmillan Publishing Company, 1991.

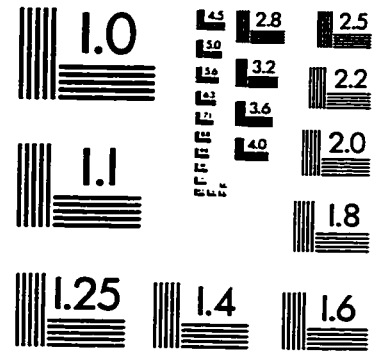
[Strayer92] Strayer, W., Dempsey B., and Weaver, A. XTP: The Xpress Transfer Protocol. Addison Wesley, 1992.

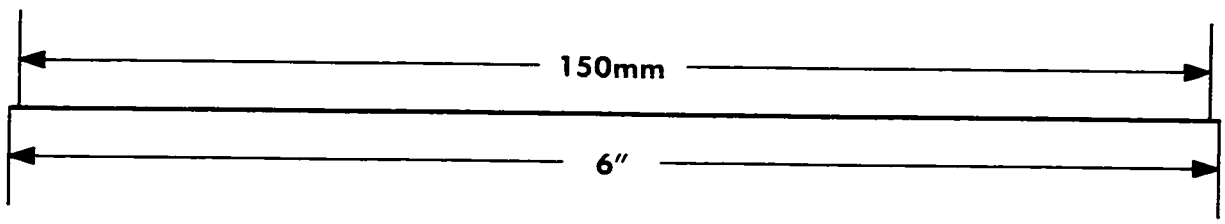
[Tanenbaum88] Tanenbaum, A. Computer Networks. Second edition, Prentice Hall Inc., 1998.

[Whiten94] Whiten, B., Steinberg, S., and Ferrari, D. The Packet Starvation Effect in CSMA/CD LANs and a solution. University of California at Berkley, 1994.

[XTPForum95] XTP Forum. Xpress Transport Protocol Specification, Revision 4.0. 1995.

IMAGE EVALUATION
TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved