The Xpress Transfer Protocol (XTP)-
A Tutorial

Robert M. Sanders

Computer Science Report No. TR-89-10
Revised January 15, 1990

The Xpress Transfer Protocol (XTP) — A Tutorial

Robert M. Sanders

Computer Networks Laboratory
Department of Computer Science
University of Virginia

January 15, 1990

The Xpress Transfer Protocol (XTP) — A Tutorial

CONTENTS

LIST OF FIGURES

LIST OF TABLES

The Xpress Transfer Protocol (XTP) — A Tutorial

Robert M. Sanders

Computer Networks Laboratory
Department of Computer Science
University of Virginia

ABSTRACT

XTP is a reliable, real-time, light weight *transfer*[1] layer protocol being developed by a group of researchers and developers coordinated by Protocol Engines Incorporated (PEI).[1,2,3] Current transport layer protocols such as DoD's Transmission Control Protocol (TCP)[4] and ISO's Transport Protocol (TP)[5] were not designed for the next generation of high speed, interconnected reliable networks such as FDDI and the gigabit/second wide area networks. Unlike all previous transport layer protocols, XTP is being designed to be implemented in hardware as a VLSI chip set. By streamlining the protocol, combining the transport and network layers and utilizing the increased speed and parallelization possible with a VLSI implementation, XTP will be able to provide the end-to-end data transmission rates demanded in high speed networks without compromising reliability and functionality.

This paper describes the operation of the XTP protocol and in particular, its *error, flow* and *rate* control, inter-networking addressing mechanisms and multicast support features, as defined in the XTP Protocol Definition Revision 3.4.[1]

1. The *transfer* layer is formed by combining the functionalities of both the network and transport layers of the ISO OSI model into a single layer.

1. Introduction

Future computer networks will be characterized by high reliability and very high data transmission rates. Traditional transport layer protocols, such as TCP and TP4, which were designed in an era of relatively slow and unreliable interconnected networks, may be poorly matched for the emerging environment. Although they contain many necessary features, such as error detection, retransmission, flow control and data resequencing, they are deficient in many respects — they do not provide rate control and selective retransmission, reliable multicast is not supported, their packet formats are complex and require extensive parsing due to variable header lengths and support of complex modes. These protocols manage many timing events at both the sender and the receiver — for example, since the sender does not initiate receiver data acknowledgements, both the receiver and sender require an additional timer. The data transmission rates assumed are no longer valid and may limit the scalability of the protocols — in TCP, for example, which was designed in an era of 56Kbps data transmission rates, the flow window size is small, and based on 16 bit byte sequencing. Finally, the state machines for these transport protocols were intended for sequential rather than parallel execution. For example, the placement of the checksum field was considered arbitrary and so it was placed in the header.

XTP provides for the reliable transmission of data in an inter-networked environment, with real-time processing of the XTP protocol — i.e., the *processing* time for incoming or outgoing packets is no greater than *transmission* time. XTP contains error, flow and rate control mechanisms similar to those found in other more modern transport layer protocols[2] in addition to multicast capability. Timer management is minimized — in XTP there is only one timer at the

2. Specifically, two other modern transport layer protocols — Versatile Message Transaction Protocol (VMTP) developed at Stanford University by David Cheriton, and Network Bulk Transfer (NETBLT) developed at MIT by David Clark.

receiver, used in closing the context. XTP has a 32 bit flow window. XTP's state machine is specifically designed for parallel execution. Address translation, context creation, flow control, error control, rate control and host system interfacing can all execute in parallel.

The XTP protocol is considered a *lightweight* protocol for several reasons. First, it is a fairly simple yet flexible algorithm. Second, packet headers are of fixed size and contain sufficient information to screen and steer the packet through the network. The core of the protocol is essentially contained in four fixed-sized fields in the header — KEY, ROUTE, SEQ and the command word. Additional mode bits and flags are kept to a minimum to simplify packet processing.

The XTP subsystem can be decomposed into four processes as shown in Figure 1 and described in Table 1.[3] These processes are the *reader, receiver,* the *writer* and the *sender.* In Figure 1, one end of a full-duplex *connection* is depicted. A connection can be considered as a pair of *contexts,* with one context at each end of the connection.

---

3. Note that other implementation schemes are possible. This particular scheme was taken from the example implementation described in the XTP Protocol Definition Revision 3.4.

Process	Accesses	Description
Reader	control blocks input buffer	Interface between XTP receiver and host operating system. Transfers data and commands from receiver to host through control blocks.
Writer	control blocks output buffer	Interface between XTP sender and host operating system. Transfers data and commands to sender from host through control blocks.
Receiver	network interface translation map input buffer context records	Parses packets received from the network. Queues data for reader in the input buffer. Uses translation map to determine context owning packet. Updates context record to maintain state of receiver.
Sender	network interface translation map output buffer context records	Prepares packets for transmission. Uses translation map to determine outbound network address. Transmits packets onto the network. Updates context record to maintain state of sender and manages XTP timers.

TABLE 1. XTP Processes

In this hypothetical implementation, control blocks are used to pass data and commands between the host operating system/user application and the XTP subsystem. Each control block corresponds to one XTP service request, such as read or write a block of data to or from the remote process. Each control block is associated with a context, whose state is contained in a context record.

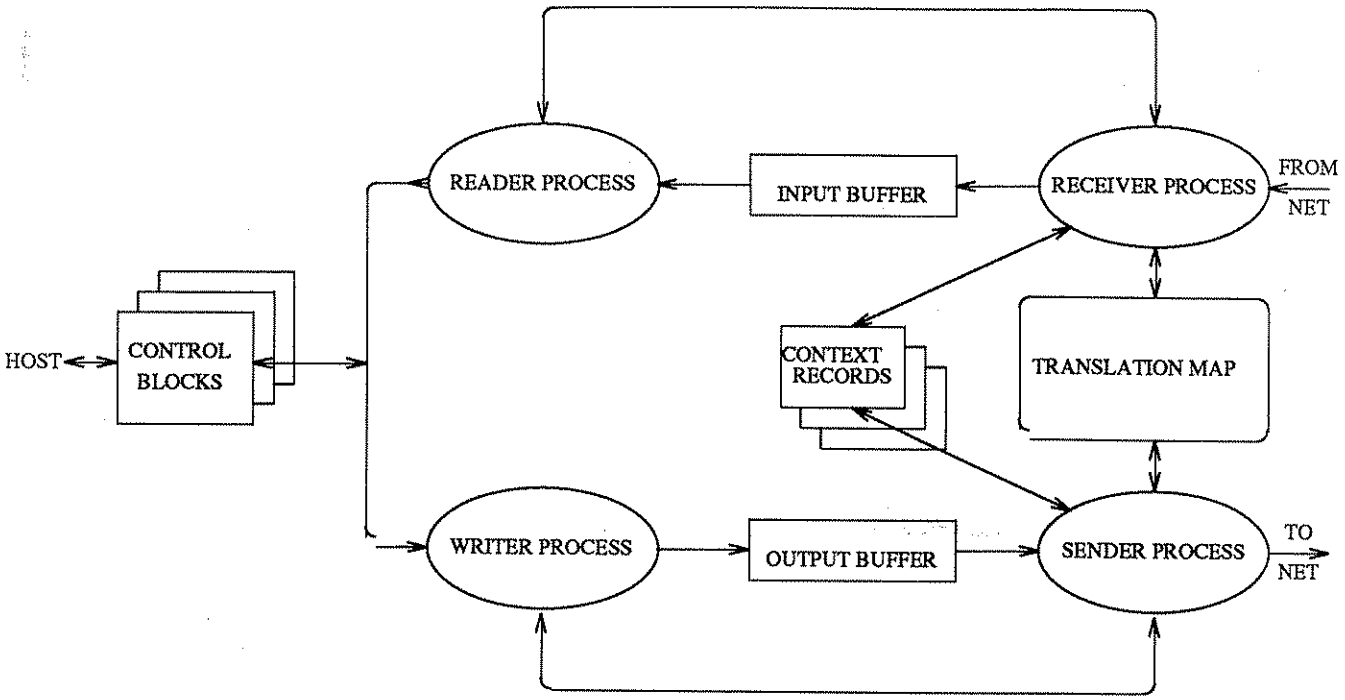


Figure 1. XTP Process Structure

Consider the sequence of events occurring when an application initiates a data transmission. The host operating system prepares a control block containing the write command and pointing to the data to transmit. The control block is then passed to the XTP subsystem. In the subsystem, the writer process examines the control block and responds by queueing the appropriate data into the output buffer for the associated context. Ultimately, the sender process prepares the data into one or more packets, which it transmits to the remote destination network address. At the destination, the XTP subsystem receiver parses the incoming packet, and queues the data for delivery to the destination's host operating system. Presently, the destination's reader process extracts the data from the input buffer, and transfers it to the host. Once the data have been delivered, the reader updates the associated control block to indicate that the data have been received.

The companies belonging to the Technical Advisory Board (TAB) developing XTP are: AMD, Apollo, Artel/NASA, Boeing, Brooktree, Concurrent, DY-4, IBM, Intergraph, Interphase, Mentat, SBE, Silicon Graphics, Synernetics, Unisys and Xerox. Research affiliates include the

University of Virginia, Concordia University, Naval Surface Warfare Center (NSWC) and Naval Ocean Systems Center (NOSC).

2. XTP Protocol Overview

A protocol specifies how data are exchanged between two or more user entities using sequences of protocol data units (PDUs). User entities for XTP are referred to as client or application processes, and may be located at or above the session layer of the OSI Reference Model. Each PDU consists of a sequence of fields laid out in a specific format. Some fields contain control information, some contain data. Some fields are optional. The length of a field may be variable or constant. Different PDU types have different packet format specifications.

2.1 Types of XTP PDUs

XTP utilizes two frame formats, one for *control* packets and one for *information* packets (see Figure 2). XTP packets can also be typed. XTP *information* packet types are DATA, FIRST, PATH, DIAG (Diagnostic), MAINT (Maintenance), ROUTE and MGMT (Management). DATA and FIRST packets both can contain user data. Also, an experimental information packet for coalescing data packets at a router is being studied and is called a SUPER packet. *Control* packets have two types: Control (CNTL) and Route Control (RCNTL). Table 2 describes the function of each of these packet types.

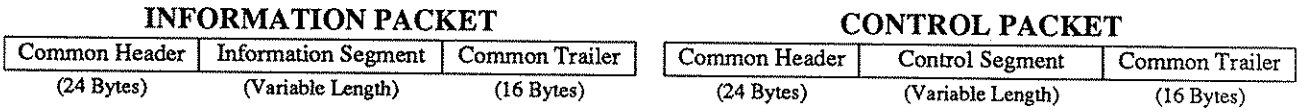


Figure 2. General Frame Formats

Both packet type formats (control and information) share a common *header* segment and a common *trailer* segment, each of constant length. The common header is 24 bytes long, while the common trailer is 16 bytes in length. Each XTP packet includes a variable length segment between the header and trailer whose segment type determines the packet type (i.e., in information packets the variable length segment is known as the *information* segment; in control packets the variable length segment is referred to as the *control* segment). The important fields

are aligned on 8 byte boundaries so that they can be quickly accessed by any machine with 2 byte, 4 byte or 8 byte alignment. The formats are described in greater detail later.

PACKET	TYPE	CODE	GENERATOR	DESCRIPTION
FIRST	INFO	00010	sender	Initiates context establishment, contains *address* segment and may contain client data.
DATA	INFO	00000	sender	Contains client data.
PATH	INFO	00110	sender	Establishes path to receiver. Used in *inter-net* connections.
DIAG	INFO	01000	receiver, router	Indicates error condition at receiver or router. (Example: destination unknown)
MAINT	INFO	01010	sender	Gathers end-to-end diagnostic data. (Example: determine hoptimes on route)
MGMT	INFO	01110		Not defined in the XTP Protocol Definition version 3.4
CNTL	CNTL	00001	sender, receiver	Used by receiver to return status; contains receiver's *error*, *flow* and *rate* parameters. Used by sender when re-synchronizing with the receiver.
SUPER	INFO	10000	router	Experimental packet format used for coalescing data packets at a router using the same route.
ROUTE	INFO	10010	sender, router	Used for route control. Sent by context originator to request that route be released. Sent by Router to acknowledge that route has been released.
RCNTL	CNTL	10011	router	Router generated CNTL packet. May be generated by router at any time.
Packet type indicated by 5 bit code in common header's *command word* field.				
Least significant bit in type field code is set for control packets.				

TABLE 2. XTP Packet Types

The common *header* specifies the packet type and identifies what portion of the data stream, if any, is included in the information segment. Optional modes, such as disabling error checking or multicast transmission, are indicated in the packet header's *control flags* field. The common *trailer* contains two checksum fields, identifies how much of the data stream has been delivered to the receiving client application, and also contains a *flags* field. These flags generally control state changes, for example closing the data transmission connection or requesting data acknowledgement. Message boundaries are also specified in the trailer by setting the *end of message* flag (EOM).

The *information* segment contains the user data being transferred, and is also used to pass addresses and other miscellaneous data when appropriate. In general, user data bytes are

streamed to the receiver application process in the order generated by the sending application process (a bit pipe). Each *data* packet contains a contiguous subset of the data stream being transferred. In XTP, there is no protocol-imposed upper limit on the number of bytes included in each data packet — each implementation is bounded by the underlying datalink layer. For each implementation this limit is known as the *maximum transmission unit* (MTU) and is found by subtracting the XTP header and trailer sizes from the datalink's maximum data field size. XTP supports two additional modes of data transfer which allow out-of-band, tagged data of constant length (8 bytes) to be included in the data packet along with the user's data. These additional data bytes also appear in the *information* segment, either at the beginning or at the end of the usual user data. Their presence is indicated by flags in the header and trailer (the *tag).* Beginning tagged data are indicated by the BTAG flag in the common header. Ending tagged data are specified with the ETAG flag in the common trailer.

The *control* segment contains the receiver's *error, flow* and *rate* control parameters' values. This segment also contains fields used to resynchronize the transmitter and receiver when necessary.

2.2 Multi-Packet Handshaking

Sequences of PDU exchanges between the user entities must correspond to a protocol-defined handshake. The handshake requires multiple packet exchanges in both directions and perhaps involving different types of packets. Two-way communication is necessary to establish end-to-end data transmission reliability levels in XTP as in other protocols.[4] In XTP, multi-packet exchange sequences provide user applications with both a transport-level *virtual circuit* capability and a transport-level *datagram* service. For example, in XTP a connection may consist of an exchange of three packets, as shown in Figure 3.

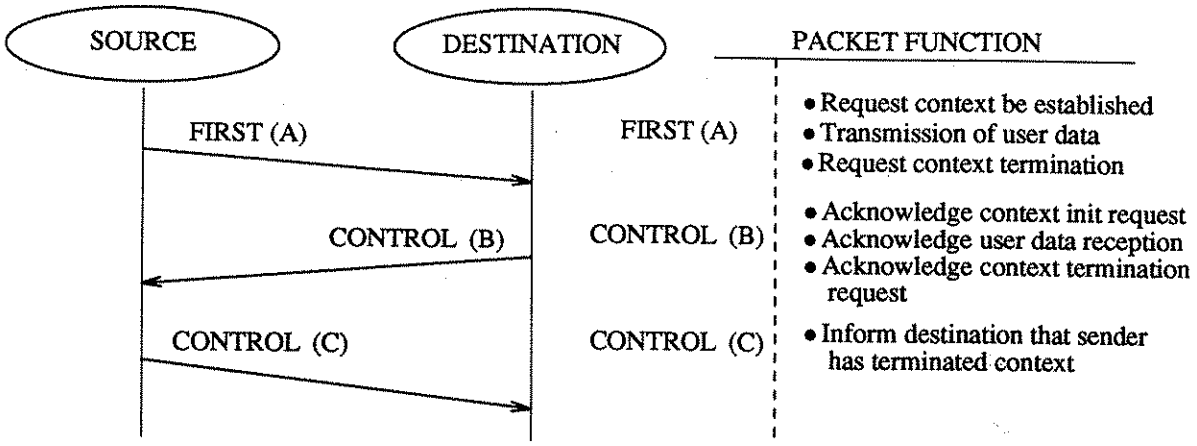


Figure 3. Three Packet Connection-Mode Handshake

The scenario above depicts how XTP can reliably set up a connection between two user processes, transmit data, and close the connection with a minimum of three packets. In this scenario, the source initially transmits packet (A). At the destination the header is examined and it is determined that the source wishes to establish a send connection. If the destination wishes to comply, a context is established. The packet's data are then queued for transfer to the waiting destination user process.

Within packet (A)'s header and trailer is encoded the current status of the connection at the source from which it is deduced that packet (A) is the last data packet to be transferred, and that the source is ready to close the connection. Also, the source requests that a control packet be returned with the current status of the destination.

After successfully transferring the received data to the host, the destination complies by sending control packet (B). This packet acknowledges the receipt of the data, and indicates that the destination is also ready to close the connection. On receiving packet (B), the sender emits control packet (C), and closes its side of the connection — thus completing the three way handshake. Any buffers still associated with the connection are freed, and the sender will no longer respond to control packets arriving for the context. When packet (C) is received by the

destination, the connection is closed.

Since the basic network may be unreliable, packets may be dropped in transit, become corrupted, arrive out of order, or may be duplicated. Packet reception is not guaranteed. The sender must assume that packet reception failed until directed otherwise by the receiver. Thus, the receiver is required to *positively acknowledge* correctly received data from the network as in TCP. This acknowledgment is contained in the control packet sent from the destination back to the sender. By a similar argument, the receiver can not be sure that the acknowledgement arrived safely at the sender unless the sender acknowledges the receiver's acknowledgement. To avoid the recursive trap of acknowledging acknowledgements, and acknowledging acknowledgements of acknowledgements, the protocol must resort to a different mechanism for guaranteeing delivery of the acknowledgement. It is more efficient for the receiver to assume that the sender received the acknowledgement unless informed by the sender otherwise. This shifts the burden of lost acknowledgments onto the sender. In XTP, the sender can request acknowledgement of all currently received data by setting the *status request* bit (SREQ) in the XTP common trailer, as described in Table 3. A timer (WTIMER) is used by the sender to determine if the receiver has failed to respond to a sender-generated request for current status and data acknowledgement. If the timer expires before an acknowledgement arrives, the sender assumes the acknowledgment was lost, and sends another request for a control packet acknowledging the received data. The only exception is in closing the connection. When closing, the source acknowledges context termination, so that the receiver can be sure that the context is closed. If this last packet gets corrupted or lost, the receiver will eventually timeout and close the connection.

This timeout method differs from the approach taken by TCP in that the XTP timer is only needed when the sender is expecting a return control packet rather than implicitly with each data packet. This significantly reduces the number of packet retransmissions when multiple data

packets are issued for each SREQ and the WTIMER times out due to a sudden increase in the round trip latency time. Unlike TCP, where each data packet would be retransmitted after the timeout, in XTP only a CNTL packet containing the SREQ would be sent. The corresponding returned CNTL packet would indicate which data packets, if any, to retransmit. This is a conservative procedure which forces a "synchronizing handshake" before retransmitting except when retransmission is explicitly indicated by the receiver.

The XTP receiver only sends an acknowledgment when the sender requests one. Thus, a range of data packets may be acknowledged by one CNTL packet. This reduces the overhead of generating and receiving extraneous CNTL packets and the number of interrupts which must be serviced per context.

Some transport protocols require elaborate packet exchanges to establish, maintain and terminate a connection. The ISO TP4 protocol, for instance, requires that six logical packets be exchanged for a single exchange of data.[5] The first pair negotiates the connection creation, the second pair sends the data packet and acknowledges the correct receipt of data, and the last pair close the connection. This additional packet ping-ponging is undesirable in a real-time environment.

Closing an XTP connection is coordinated using the three flags RCLOSE, WCLOSE and END. These flags are listed in Table 3. The local host sets the RCLOSE or WCLOSE flags in an out-going packet to inform the remote host that it has completed all reading or writing it intends to perform on the shared connection. Note that in a full duplex connection between two nodes A and B data would be transmitted in both directions ($A \rightarrow B$ and $B \rightarrow A$). Using RCLOSE and WCLOSE, each direction can be shut down independently. Suppose $B \rightarrow A$ completes first. In the last data packet from B to A the WCLOSE flag is set, indicating no more data will be sent from B to A on the connection. A responds by acknowledging the received data and the WCLOSE request

by setting RCLOSE in the subsequent CNTL packet. Meanwhile, data packets from *A* are still being generated and transmitted to *B* on the same connection. Packets from *B* to *A* now have WCLOSE set, and are only CNTL packets acknowledging data sent from *A* to *B*. Packets sent from *A* to *B* do not have WCLOSE set, but do have RCLOSE set. Finally, when *A* is preparing its final data packet, it sets the WCLOSE flag in the outgoing packet to inform *B* that *A* has also completed writing, and is ready to close the $A \rightarrow B$ transmission (RCLOSE is also set in this packet). *B*'s acknowledging CNTL packet also contains both RCLOSE and WCLOSE.

The END flag is set in an outgoing packet to signal to the remote host that the local host released or closed its end of the connection. Thus, END is set in the final packet transmitted, and indicates that the context has been terminated — i.e., that it is guaranteed that no further packets can be exchanged. If, at any time, a packet is received with the END bit set, the context is assumed closed at the remote end, and the local host releases the context. This means that the receiver/sender will not generate further packets, unless errors occur requiring retransmission. In the previous paragraph, *B*'s last CNTL packet would have the END flag set.

Parameter	Location	Description
SREQ	trailer *flags* field (1 BIT)	(Immediate Status Request) Set by sender when requesting receiver's status. Effect: Receiver immediately returns a CNTL packet containing up-to-date *error*, *rate* and *flow* parameter values.
DREQ	trailer *flags* field (1 BIT)	(Delayed Status Request) Set by sender when requesting receiver's status. Effect: Receiver delays returning CNTL packet until all queued data has been delivered to receiving client.
END	trailer *flags* field (1 BIT)	(End of Connection) Set in the last packet for each connection. Effect: No more packets will be transmitted.
RCLOSE	trailer *flags* field (1 BIT)	(Read Side Closed) Set when closing connection. Effect: Future incoming packets will be ignored, even if some data has not been acknowledged.
WCLOSE	trailer *flags* field (1 BIT)	(Write Side Closed) Indicates that all user data has been transmitted. Effect: New output commands are aborted but retransmission of unacknowledged data may occur.
EOM	trailer *flags* field (1 BIT)	(End of Message) Marks the end of the current message transmission. Future packets (except for retransmissions) will pertain to the next message. Effect: EOM indication passed to receiving client.
BTAG	header *flags* field (1 BIT)	(Beginning Tagged Data) Signifies presence of user-tagged data in first 8 bytes of the *information* segment. Effect: BTAG indication and associated data are passed to receiving client.
ETAG	trailer *flags* field (1 BIT)	(Ending Tagged Data) Signifies presence of user-tagged data in last 8 bytes of the *information* segment. Effect: ETAG indication and associated data are passed to receiving client.

TABLE 3. XTP Protocol Control Flags

Referring to Figure 3, Packet (A) requests context termination by setting the WCLOSE bit. The destination notes that WCLOSE has been set, and acknowledges the context termination request by setting RCLOSE in control packet (B). In the final packet (C), the sender sets all three flags (END, WCLOSE and RCLOSE) to terminate the connection.

The "close" protocol based on END, RCLOSE and WCLOSE can uniformly support the three packet graceful termination of Figure 3, an abbreviated termination, transactions, and abort

situations without modification.

The two-packet, transaction-like packet exchange sequences are referred to as *fast handshakes*. For *full duplex* connections, these modes are less reliable than the three packet connection and appear in Figure 4. The two packet fast close can be considered a transport level datagram service or the basis for simple request/response operations.

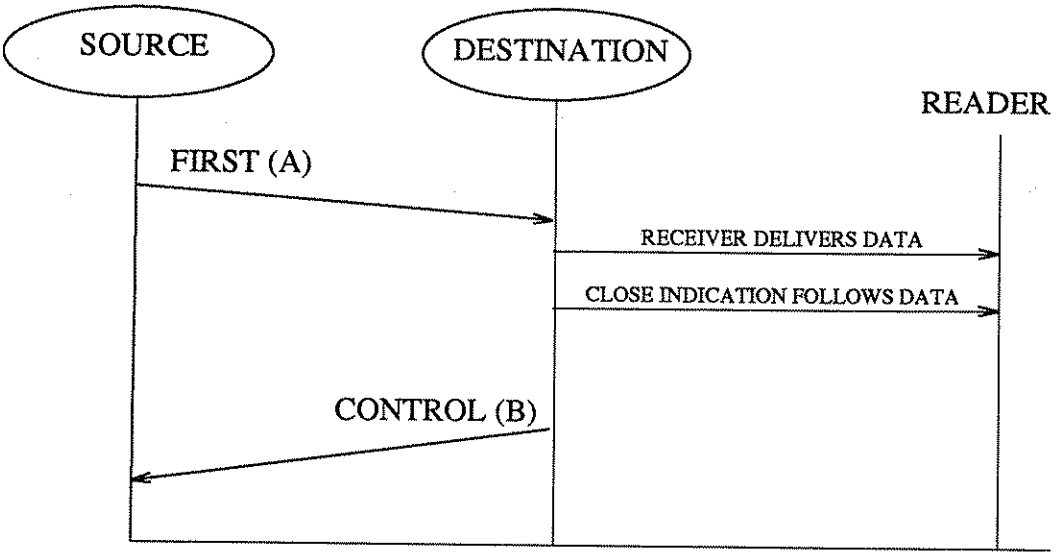


Figure 4. Two Packet Transaction-Mode Handshake

In one fast close mode, the source informs the destination that a final close acknowledgment packet will not be sent by setting both RCLOSE and WCLOSE in packet (A)'s trailer. The source also sets SREQ, as discussed previously, to request acknowledgement of the data it transmitted. Since the source has set RCLOSE, the destination knows that the source will not transmit a final acknowledgement after receiving control packet (B). The advantage of this mode is that the destination doesn't have to wait to close the context after issuing the closing control packet. Control packet (B) sets RCLOSE, WCLOSE and END.

In another fast close mode, the sender sets WCLOSE and SREQ in packet (A), and the receiver returns with WCLOSE, RCLOSE, and END set in packet (B). The relationship between

the XTP closing flags can be illustrated as shown in Figure 5. In Figure 5, six different paths for closing a connection are depicted.

In the paths marked a) and b), the local host operation system or XTP client application has requested a graceful close of the XTP connection. Path a) corresponds to the case where a sending context has completed data transmission. In the first step, the host informs the XTP subsystem that closing has been requested. As in Figures 3 and 4, the XTP subsystem responds by setting the WCLOSE flag in the next outgoing packet. At this point, the sending context enters the next step in closing. In this step, the local XTP subsystem waits for the remote system to acknowledge all data and, *specifically*, to acknowledge the write close request (WCLOSE) with a remote read close acknowledge (RCLOSE received). At this point, the WTIMER is started, as shown in the picture by a loop back to the same state. If the acknowledgement occurs before WTIMER expires, the connection can now be closed by sending the final *sender*-generated CNTL packet of Figure 3. Otherwise, the WTIMER expires, a second WCLOSE request is sent, and the WTIMER is restarted. Path b) is a symmetric case when the host at the reader requests a graceful close.

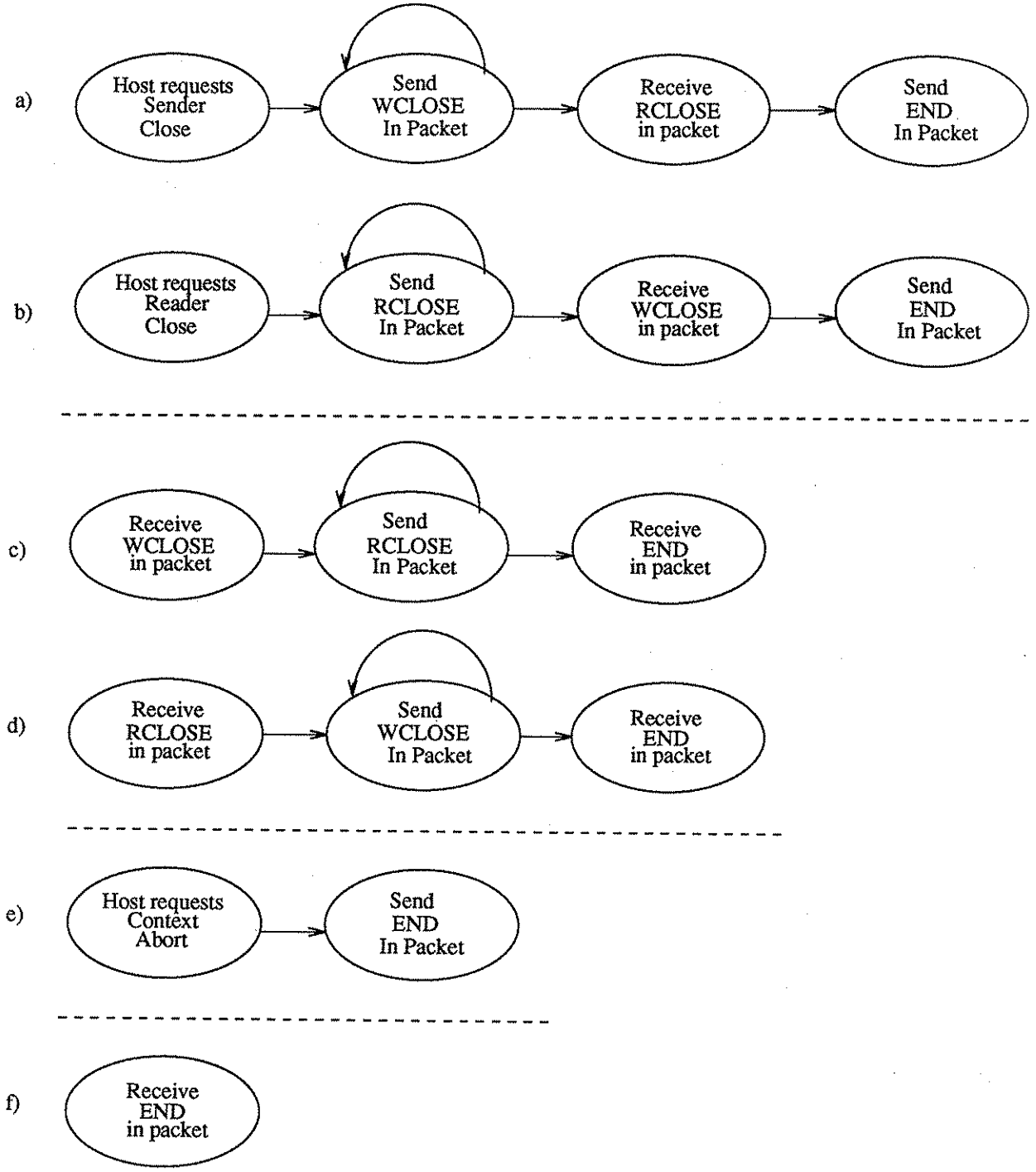


Figure 5. XTP Closing Connection Modes and the WCLOSE/RCLOSE/END Flags

In parts c) and d), the local XTP subsystem has detected a close request from the remote end of the connection, and responds by closing the local end gracefully. As in the host initiated close modes, the WTIMER is used to retransmit the close acknowledgement until a packet containing a

set END bit is received. (Once again, the third and final packet in Figure 3.)

The three packet close can at any time be short-circuited by sending the END bit prematurely, from either end of the connection. As shown in part e), this may have been initiated by an "impatient" host. At the connection end receiving the END packet, closing is abruptly terminated as in part f).

Note that when a *data* acknowledgment is requested in XTP, as in the FIRST packets of Figures 3 and 4 (packet (A) in both figures), the acknowledgment is not necessarily provided immediately. In the fast close cases, the receiver delays acknowledgment until all data received prior to the SREQ have been processed. This includes the data contained in the packet with SREQ.

XTP contains a second status request flag in the common header flags field which is called DREQ. DREQ differs from SREQ in that SREQ requests a response immediately from the receiver, and DREQ requests it after the currently queued data have been received at the receiver. This is useful because the acknowledgement is delayed until the receiver has freed the buffer space associated with the queued data and is capable of accepting more data from the sender. Flow control blocking is minimized.

In closing, SREQ behaves like DREQ — if this were not so, the protocol would behave as follows in minimal packet exchange scenarios such as in Figure 3. Packet (A)'s SREQ could generate CNTL packet (B) before the data from packet (A) had been delivered to the receiving client. Thus, CNTL packet (B) doesn't acknowledge the data sent in packet (A). At the sender, the context could not be closed because the data was not acknowledged. After the WTIMER expires, a new packet would be generated at the sender, say packet (C') requesting the receiver once again return its status. The final control packet, say (C''), would acknowledge that all data had been successfully transferred. In this scenario, an extra packet, Packet (C') has been sent, and

the WTIMER has been forced to expire at least once, both of which are unnecessary and undesirable. Thus, in closing, SREQ responses are delayed until all data have been processed.

3. Error, Rate And Flow Control in XTP

XTP includes substantial *error, rate* and *flow* control mechanisms which all require feedback from the remote XTP receiver process to the local XTP sender process. This feedback is contained inside CNTL packets and guides the sender on *what* and *when* to transmit. In this section, each of these control mechanisms is explored. In particular, *flow* control is presented within the context of an example, depicted in Figure 6. This figure illustrates a situation where the sender intends to transmit a total of 27 packets containing user data to the destination. At the point in time depicted, the data in the first seven packets have been transmitted, has arrived correctly at the destination's receiver, is queued for delivery to the destination host receiving process, and has been accepted by the host — processing on these seven packets has thus been completed. The receiver has detected a *gap* in the data stream occuring over the bytes in packets 11 through 14. A *gap* is detected when out of sequence data are received and accepted. The missing data bytes may be lost or delayed. The XTP packet format has provisions for reporting up to 16 separate gaps that are outstanding within the data stream of any one context at a given point in time.

Packets 8, 9 and 10 have been received by the destination, and queued for transfer to the host. They currently occupy space in the XTP buffer for receiving packets from the network. Buffer space is finite, and is partitioned among various contexts between the destination and other hosts on the inter-network. The buffer space currently allocated for this context's receiver buffer at the destination is large enough to hold 13 data packets. **(Note: Buffer space is actually allocated as a number of bytes[4], not packets, because the amount of data contained in each packet may vary. The scenario presented here has been simplified.)** Packets 8, 9 and 10, in addition to

packets 15, 16, 17 and 18 are in the buffer, leaving space for six more data packets.

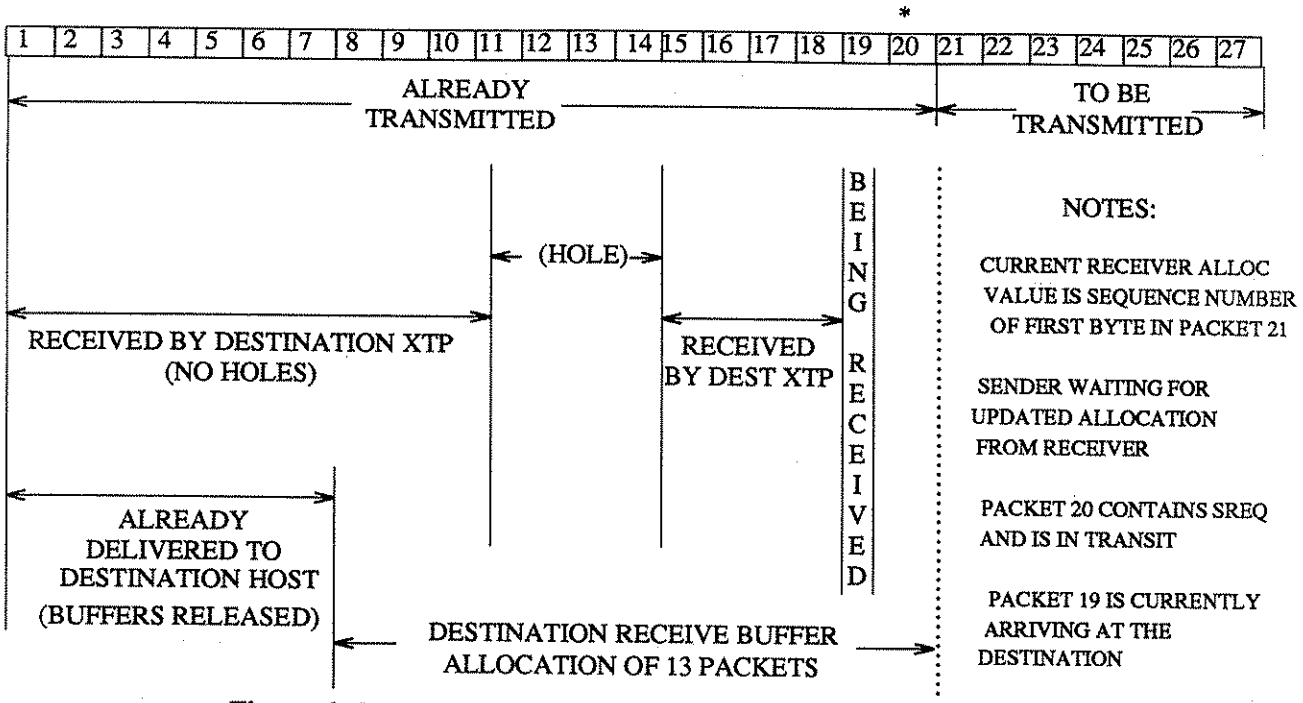


Figure 6. XTP Flow Control and Selective Retransmission

The four packets 11, 12, 13 and 14 were transmitted, but never arrived at the destination in an acceptable form. Together, they represent a gap in the stream of data received from the network for the context. These packets have been either discarded or delayed somewhere in the network. If the receiving XTP process continues to accept packets beyond packet 20 before the gap is filled, the receiver will not have enough buffer space for the gap when it arrives. The destination XTP process must assume that the gap will eventually arrive, and therefore sets aside space for the lost bytes from the allocation. Since the XTP receiver process is required to deliver the data

---

4. An advantage to this policy is that byte-oriented allocation is not affected by internet fragmentation.

stream in the same byte ordering that the source process generated it, packets 15, 16, 17 and 18 must be held in the buffer until the preceding gap is filled. Thus, 11 data packets are currently in the buffer space, or have space reserved. The remaining two packets of buffer space are currently free.

3.1 Flow Control

When the buffer space is full, the receiver will discard any additional non-gap-filling data packets, even if they are well formed — the receiver will not overrun its buffer allocation for the context. Unless the sender has detailed knowledge of the receiver's buffer space, and the existence and extent of gaps in the received data, it may continue transmitting new data packets that eventually get dropped by the receiver, needlessly overburdening the network.

Thus, a mechanism must exist for the receiving XTP process to inform the sending XTP process about the current state of its receiving buffers. This information is included in control packets sent from the receiver to the sender. Specifically, the receiver includes the parameters in Table 4.

Parameter	Type	Location	Description
ALLOC	32 bit sequence number	control segment	1 + sequence number of last byte receiver will accept.
DSEQ	32 bit sequence number	common trailer	1 + sequence number of last byte receiver delivered to destination client process.
RSEQ	32 bit sequence number	control segment	1 + sequence number of last byte receiver accepted.
ALLOC - DSEQ	Size of receiver's data buffer in *bytes*.		
RSEQ - DSEQ	Number of bytes received and waiting to be transferred to destination client process.		

TABLE 4. XTP Flow-Control Parameters

ALLOC constrains the sender from introducing more data than the receiver's buffers can accept. The sender refrains from sending bytes with sequence number ALLOC or higher. Thus, ALLOC is one greater than the highest byte sequence number that the receiver will accept. DSEQ is the sequence number of the next byte to be delivered to the destination application process, or client. Likewise, DSEQ can be thought of as one greater than the sequence number of the last

byte delivered to the destination client. All bytes with sequence number less than DSEQ have been successfully transferred to the destination client. DSEQ is always less than or equal to ALLOC. Subtracting DSEQ from ALLOC (modulo 2^{32}) yields the buffer size allocated in bytes to the context by the receiving XTP process. Note: A default value of ALLOC is used initially by the sender until a value is received from the receiver.

The sender holds data that have been transmitted in a buffer until it knows the data have been delivered to the destination client. As long as the data are buffered, they can be retransmitted if necessary. When the sender notes that DSEQ has been extended, it frees the buffers associated with the delivered data.

Note that DSEQ appears in the common trailer rather than in the control segment like ALLOC and RSEQ. Refer to Figures 16 and 18 for an exact layout of both the control segment and the common trailer included in control packets.

RSEQ is the sequence number of the first byte not yet received contiguously from the network. This can be the first byte in the first gap, or the first byte in the next data packet expected. As with ALLOC and DSEQ, an alternative interpretation exists for RSEQ. All bytes associated with sequence numbers less than RSEQ have been buffered by the receiving XTP process at the destination, but may not have been delivered to the destination client process yet. Thus, RSEQ is one greater than the largest *consecutively* received data byte sequence number. The sequence numbers of all bytes associated with gaps lie between RSEQ and ALLOC. The following relationship conceptually holds for DSEQ, RSEQ and ALLOC:

$$DSEQ \leq RSEQ \leq ALLOC$$

Collectively, these parameters provide the means for XTP to implement *flow* control whereby the receiver can restrict the sender from sending excessive data prematurely. Note that all sequence number parameters in XTP occupy 4 bytes — SEQ, RSEQ, DSEQ, ALLOC and the

sequence number pairs contained in the SPAN field of CNTL packets associated with gaps in the received data stream.

The exception to the above inequality occurs when the number of bytes to transmit exceeds 2^{32}. In this case, insufficient bit patterns exist using 32 bit sequence numbers to uniquely identify each byte to be transmitted. To arbitrarily bound the size of data transmissions to this or any other number would be unacceptable. To allow unbounded-sized transmissions, sequence numbers must be reusable. XTP, like other protocols, reuses sequence numbers when necessary using modulo arithmetic — byte 0 follows byte $2^{32} - 1$. Thus, in practice, ALLOC may wrap when extended, and actually decrease in value, as depicted in Figure 7.

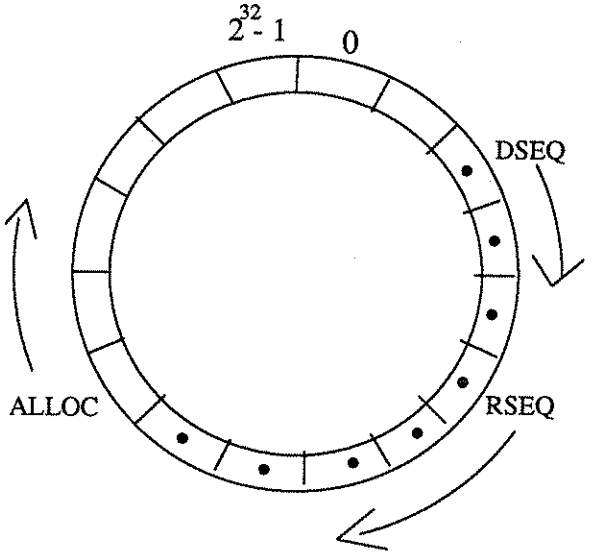


Figure 7. Flow Window Ring Structure

It should be observed that each byte still in the *bit pipe*, i.e., each byte currently in transit or still subject to retransmission, must be uniquely identifiable, so that retransmission is possible. In TCP/IP sequence numbers are limited to 16 bit numbers with only $2^{16} = 64K$ bytes possible in the bit pipe at any given point in time. On the other hand, XTP's 2^{32} bit patterns yield over 4 billion unique sequence numbers. Thus, XTP is more naturally suited to networks with both high

bandwidth and/or high end-to-end latency than TCP/IP.[5]

Once the sender has been informed of the receiver's allocation limit via the ALLOC parameter, it continues to transmit until the allocation has been reached, without the need for individual acknowledgements of each packet transmitted. Thus, XTP more efficiently utilizes the higher reliability of modern networks, such as fiber optic LANs. Once the allocation has been reached in this hypothetical example, the XTP sender process sets the SREQ parameter in the last data packet transmitted, and the receiver responds as earlier described with a control packet that acknowledges all data received, describes any gaps detected, and, if appropriate, advances the allocation.[6]

For example, in Figure 6 the receiver is currently receiving packet 19, and waiting to deliver the packets 8, 9 and 10 to the destination client. Suppose that both tasks complete before packet 20, marked with an asterisk, arrives. The values of the parameters DESQ, RSEQ and ALLOC will have changed as depicted in Table 5.

---

5. Van Jacobsen has proposed extending the TCP protocol to, among other things, include 29 bit sequence numbers to extend the size of the TCP flow control window.[6]

6. Note that other policies are possible for determining when to set the SREQ bit in XTP; in XTP, the SREQ policy is determined by the user application.

Parameter	Before	After
ALLOC	(sequence number of first byte in packet 21) or (1+sequence number of last byte in packet 20)	(sequence number of first byte in packet 24) or (1+sequence number of last byte in packet 23)
DSEQ	(sequence number of first byte in packet 8) or (1+sequence number of last byte in packet 7)	(sequence number of first byte in packet 11) or (1+sequence number of last byte in packet 10)
RSEQ	(sequence number of first byte in packet 11) or (1+sequence number of last byte in packet 10)	no change

TABLE 5. Change in Flow-Control Parameter Values

These changes reflect the freeing of buffer space associated with packets 8, 9 and 10 that allows the allocation to be extended by three data packets. RSEQ has not advanced because the sending XTP process is still unaware of the gap's existence and has not retransmitted the missing packets contained in the gap. When packet 20 is decoded, the receiver sees that SREQ has been set, and responds by sending back across the network a control packet with the new allocation and a description of the gap. Until the control packet arrives at the sender XTP process, the sender refrains from further data packet transmission. After decoding the control packet, the sender notes the new, extended allocation, and transmission may resume. Packets 11, 12, 13, 14, 21, 22 and 23 could be sent.

An alternative allocation policy exists in XTP based on the size and availability of the receiving client application's buffers. This mode is referred to as *reservation* mode. In reservation mode, the transmission is determined by the size of the receiving user's buffers reserved specifically for the context. In this mode, the sender must pause between message transmissions (the end of a message is indicated when the EOM bit is set in an outgoing XTP packet) until the receiving client has posted a new client buffer to receive the next message. This is necessary to separate adjacent messages into different client buffers, since each message may not entirely fill its buffer.

The mode is invoked by the RES flag in the common header flags field. The 4 byte ALLOC field is redefined in this mode to contain the size of the current receive buffer at the receiving client when the RES flag is set. ALLOC is located in the common header. Note that the field designated RESERVED in the XTP header (see Figure 16) has nothing to do with the reservation mode described here. The RESERVED field is reserved for further extensions to XTP and is undefined at present.

In reservation mode, the reservation buffer size may differ greatly from the normal allocation size, and may be greater. This mode is similar to the the allocation control mechanisms in the VMTP[7] and NETBLT[8] protocols.

3.2 Rate Control

Unfortunately, flow control is not sufficient to ensure efficient, error-free transmission between the sender and receiver, even on an extremely reliable network. Imagine a network containing both hardware and software implementations of the XTP protocol. Since the VLSI chip set will allow much of the protocol to be executed in parallel, a sending XTP process implemented in hardware may overwhelm a receiving XTP process implemented in software if it sends multiple, back-to-back packets.

One solution would be for the receiver to impose a one packet allocation scheme in which the sender would block after each packet — i.e., *stop-and-wait*. Each data packet would contain a SREQ, and each packet would be individually acknowledged by the receiver. In this scheme, excessive numbers of control packets would be generated (one per data packet), and the transfer of data would proceed slowly.

Even with all-hardware implementations, a router between two networks may be transferring multiple data streams between two networks where each data stream is attempting to use the maximum data flow rate possible. Although the hardware XTP receiver in the router may have no

trouble processing and queueing a burst of incoming data packets as it arrives, the router's output buffers may fill up due to the unpredictable backlog of packets queueing for output on the target network. Consider, for example, a node on an FDDI LAN connected to a node on an Ethernet LAN through a router. Clearly, the router occasionally needs a mechanism for lowering the packet arrival rate. The one packet allocation approach would be very inefficient, and all of the extra control packets would still pass through the router, taxing its capabilities further. In short, a better approach is needed.

The XTP solution uses *rate* control to restrict the size and time spacing of bursts of data from the sender. Within any small time period, the number of bytes that the sender transmits must not exceed the ability of the receiver (or intermediate routers) to decipher and queue the data — otherwise they will be overwhelmed and begin dropping packets, creating gaps in the received data stream. This problem is independent of the flow control/buffer size problem discussed previously. The receiver may have adequate buffer space available, but back-to-back packets may arrive faster than the XTP receiver process can analyze them. The XTP parameters used to implement rate control are shown in Table 6. Together, the two rate control parameters allow the receiver to tune the data transmission rate to an acceptable level.

Parameter	Location	Description
RATE	control segment	Maximum number of bytes receiver will accept in each one second time period.
BURST	control segment	Maximum number of bytes receiver will accept per burst of packets. The transmitter may not transmit more than BURST bytes between RTIMER timeouts.
RATE/BURST	Maximum number of packet bursts per second.	
BURST/RATE	Seconds per Packet Burst. The rate timer (RTIMER) is set to this value.	
RATE = -1	Rate control is disabled — i.e., sender transmissions are unconstrained.	

TABLE 6. XTP Rate-Control Parameters

In the first situation described, where a XTP receiver implemented in software is listening to a hardware-implemented sender, packet bursts must be time spaced to guarantee that the slow receiver has sufficient time between back-to-back packet bursts to complete protocol processing

before the arrival of the next burst. With the above parameters, inter-packet spacing can be achieved as follows. Set the BURST parameter equal to the MTU (maximum transmission unit) of the underlying network. Thus, each packet "burst" may not contain more than one packet's worth of data. If the receiver can handle *N* packets per second, set RATE equal to *MTU* * *N*. In this manner, the sender is constrained to spacing back-to-back packets accordingly. See Figure 8 which plots bytes transmitted versus time during a one second time period for a hypothetical XTP transmitter.

The RATE and BURST parameters are adjustable, and for each implementation of XTP, appropriate values could be determined experimentally. Their values would then be included in all out-going control packets from the receiver. Note that in this example, RATE > > BURST.

In Figure 8, the BURST and RATE parameters have been adjusted such that an inter-burst *separation* occurs. Each burst of data is depicted by a ramped triangle. The separations between adjacent bursts are shown by horizontal dotted line segments in which no progress is made towards the top of the graph. During each pause in the transmitter, the slower receiver is allowed to catch up.

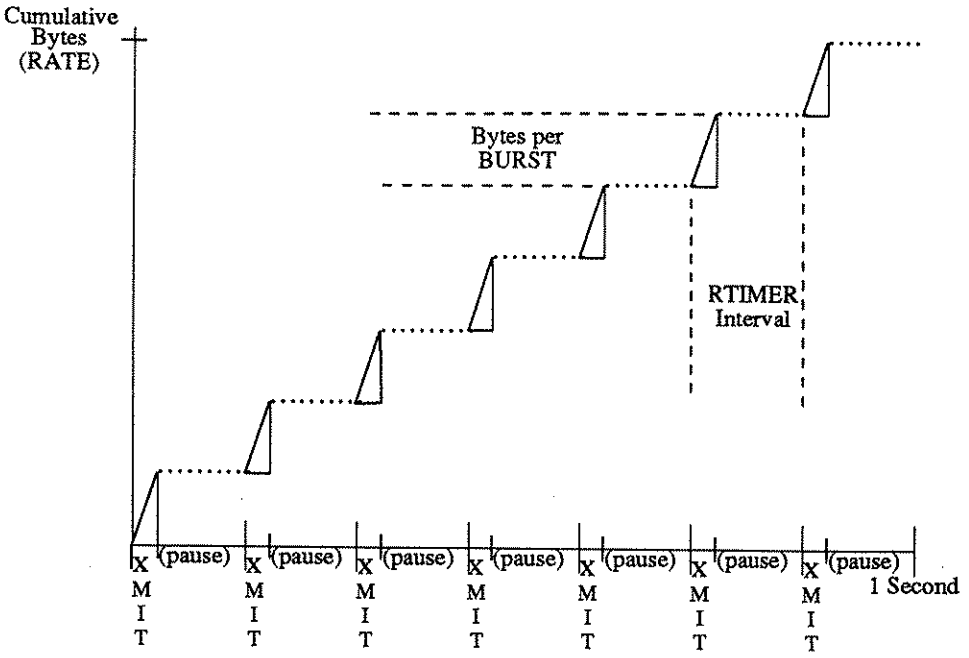


Figure 8. Rate Control of a Hypothetical XTP Transmitter

Unfortunately, the sender process does not know the appropriate RATE and BURST values to use with a particular receiver until the first burst of data has been completed; the proper value for ALLOC is also unknown initially. The appropriate values only become known when the first control packet arrives at the sender. Before this control packet is returned, the sender must use default values for the various *flow* and *rate* control parameters. These values may be different for outgoing data than for incoming data; for instance, on a network with one hardware implementation and 5 slower software XTP implementations, incoming packets to the hardware XTP receiver can be handled with no spacing between packets, but outgoing packets need to be spaced by the same node. Thus, different *flow* and *rate* control parameters may be used by the sender process.

If protocol processing speeds vary widely on a network, the default values for ALLOC and *rate* control parameters affect the number of dropped packets during the initial data burst. A conservative approach would be for the sender to set the default ALLOC to a small number of

bytes (say one average-sized data packet as defined by the *maximum transmission unit)* and to use the aforementioned approach to setting the default *rate* parameters such that packet spacing is sufficient for the slowest receiver on the network. After the initial burst, which also establishes the context connection, the sender would block, waiting for the returned control packet generated by the SREQ in the last data packet of the burst. This control packet would contain the more accurate *flow* and *rate* control parameters specifically applicable to the receiver. In this case, few packets would be lost at the cost of moderately more overhead in the initial burst.

In the router example discussed above, back-to-back packet delay is not needed, but limiting the number of bytes arriving in a given time period is; thus setting RATE > > BURST (=MTU) is not adequate for controlling flow rate. The router, implementing the XTP protocol in hardware, can absorb back-to-back packets as fast as they arrive, but must avoid exhausting the buffers between the two networks. To implement this, BURST could be set equal to RATE, and RATE would be set to the rate at which the router could relay frames for the context in terms of bytes per second. In this scenario, the RTIMER's interrupt rate would be once per second, and the number of bytes per second allowed would equal RATE. As more inter-network contexts become established, the router may need to restrict the burst rate for existing contexts with the RATE (=BURST) parameters. Later, as contexts become inactive or removed from the inter-network, the router may chose to increase the flow rate of the remaining contexts. RATE (=BURST) would be increased in outgoing control packets in this case. RATE and BURST allow the router to dynamically control the flow into the router so as to avoid overwhelming it with requests.

XTP's rate control feature may be disabled by setting RATE equal to -1 in outgoing CNTL packets.

3.3 Error Control

When errors do occur in transmission, XTP, like TCP and TP4, must detect the errors and initiate retransmission of the erroneous data. XTP uses two checksums over the XTP packet contents to verify the integrity of the data received through the network. These two checksums appear in Table 7. The XTP checksum algorithms were chosen for speed and VLSI compatibility; details of their operation are found in Appendix A of the XTP Protocol Definition version 3.4.[1]

Parameter	Location	Description
DCHECK Value	trailer (4 bytes)	4 byte checksum over data fields. Includes the control segment in control packets; the information segment in information packets.
HTCHECK Value	trailer (4 bytes)	4 byte checksum over header and trailer.
NODCHECK Flag	trailer *flags* field (1 BIT)	Flag used to signify that DCHECK checksum is not present in current packet.
NOCHECK Flag	header *flags* field (1 BIT)	Flag used to signify that checksum calculation is disabled in current packet.
XOR	Calculated using exclusive-OR operations only. Represents the vertical parity of data bytes.	
RXOR	Each intermediate result is left rotated before exclusive-ORing in the next word.	
XTP's checksum function is formed using left rotation and exclusive-OR operations over the 16-bit words covered. The 4 byte checksum is the concatenation of two 2-byte checksums XOR and RXOR. (XOR | RXOR).		

TABLE 7. XTP Checksum Parameters

It is preferable to place the checksums in the last few bytes of the XTP frame so that the checksum calculation can be concurrent with packet transmission or reception. If the checksums were placed in the front of the packet, the entire packet would have to be accessed to compute the checksum before packet transmission begins. Thus, two sweeps over the data would be necessary — one for the checksum, and one for copying the bytes to the network. This inefficient approach is inherent to TCP and TP4, whose checksums occur before the information segment, and avoided in XTP where the checksums follow the rest of the packet and are found in the common trailer.

The checksum DCHECK is optional in that it can be activated or deactivated by setting the NODCHECK flag in the XTP common trailer's flag field. When NODCHECK is set, no DCHECK is calculated by the sender XTP process, and the DCHECK field is undefined.

Checksum calculation is also not performed when the NOCHECK bit is set in the header *flags* field.

When either checksum indicates that the packet received contains erroneous information, the receiver assumes the packet is garbled and discards it. If the source were known, the receiver could immediately inform the source XTP sender process that the packet was garbled in transit — allowing the source to begin retransmission. Normally, this information is available by referencing the packet's KEY field, located in the common header, that uniquely identifies the originating client process at the node that transmitted the packet. But, the receiver cannot assume that the KEY field is correct, since the error could conceivably have occurred anywhere within the packet including the KEY field itself (if the HTCHECK checksum is invalid). Thus, the receiver always discards packets received with errors.

At the sender, transmission continues as if no error had occurred. The next packet is placed onto the network. If this new packet arrives correctly, the receiver examines the starting sequence number for the packet. Like the context identifier KEY, the starting sequence number is contained in the packet's header (in the SEQ field). The receiver expects the SEQ value of the incoming packet to equal the current RSEQ value for the context. Since a packet was dropped, the incoming SEQ is larger than RSEQ by the size of the dropped packet. The receiver accepts the data packet, noting that it arrived out of sequence, and that a gap exists in the data stream. Now the receiver can utilize the KEY information of the current packet to send back a CNTL packet to announce the gap. Having the receiver indicate when a gap has been detected is optional in XTP; if the receiver fails to send the CNTL packet, the sender will eventually include a SREQ and block, or timeout.

3.4 Gaps And Selective Retransmission

A receiver could describe a gap using a pair of sequence numbers that bound the gap. Instead, XTP describes the groups of bytes (called *spans)* which *were* received. This process is known as *selective acknowledgement.* Thus, in XTP, the location of gaps is inferred to be between the spanning byte groups selectively acknowledged. Each byte group is described with two sequence numbers that bound the bytes received. Associated with each byte group is a gap immediately preceding it in the ordered data stream. The first sequence number in the pair marks the byte where the group started (i.e., the first byte *in* the group). The second sequence number is one greater than the last byte in the group (i.e., the first byte *not contained in* the group.) Between each pair of received byte groups is a gap, or hole, in the received byte stream encompassing one or more bytes, as illustrated in Figure 9.

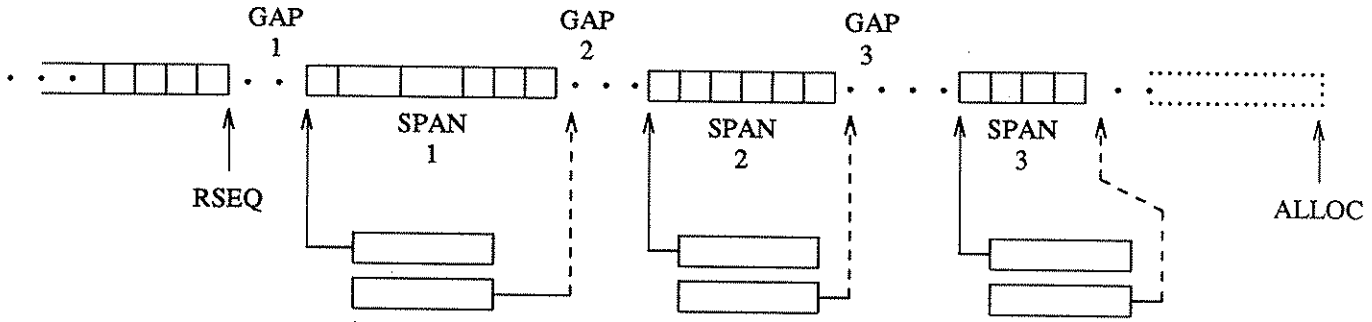


Figure 9. Gaps and Spanning Byte Groups

As mentioned earlier, XTP allows a receiver to track and notify up to 16 separate gaps for any given context. This capability is not required, however — receivers may chose to ignore all out-of-sequence data. In this case the receiver would never allow gaps to be created, and would force the sender to retransmit both lost data and correctly received out-of-sequence data. This latter method is referred to as *go-back-n* retransmission.

Since up to 16 byte groups may be described in any CNTL packet, the SPAN field must be variable in length. It is located in the *control* segment, and contains descriptors for the gaps.

Each byte group descriptor takes 8 bytes and contains the two 4 byte sequence numbers that bound the group. Preceding the sequence number pairs is NSPAN, occupying 4 bytes so that the following SPAN sequence number pairs are aligned onto 8 byte boundaries. NSPAN contains the number of byte groups described with the SPAN field. In Figure 9, NSPAN would equal 3.

Each gap spans a portion of the data stream. For 16 individual gaps to accumulate would presumably be a rare occurrance, and only possible when large volumes of data are transmitted with few SREQs. Consider a massive file transfer between mainframes with considerable buffer space. The entire file could be transferred with a single SREQ in the final data packet. Any lost data could be determined and communicated to the sender in a minimum number of CNTL packets (one) in most cases. This process, in which only the lost data are retransmitted, is known as *selective retransmission*. The XTP selective retransmission parameters are summarized in Table 8.

Parameter	Location	Description
NSPAN	control segment	Number of *spanning* byte groups described in the SPAN field. Legal values range from 0 to 16.
SPAN	control segment	Variable length field containing pairs of sequence numbers. Each sequence number pair describes a *spanning* byte group. The first sequence number in each pair is associated with the starting byte of the group. The second sequence number is one greater than the sequence number of the last byte in the group.
Spanning Group	A contiguous group of bytes received out of sequence. To the left of each *spanning* group is a *hole* or *gap*. The sequence numbers of all bytes in all *spanning* groups are between RSEQ and ALLOC.	
Gap	Portion of the data stream currently in transmission which has been lost or delayed. The receiver detects a *gap* when a packet arrives whose starting sequence number (SEQ) > RSEQ.	

TABLE 8. XTP Selective Retransmission Parameters

4. XTP Timing Considerations

In certain pathological cases, the XTP connection may be severed without one or both ends realizing what has occurred. Typically, *connection* timers are used to detect the possibility that such an event has occurred. In XTP, the CTIMER is used to monitor for such events. CTIMER expires when the connection has been inactive for 60 seconds. By the time a break is suspected, a number of attempts may have been made to prompt the other "end" to re-synchronize the protocol. In XTP, these prompts are in the form of CNTL packets (called *sync* packets). If, after a number of attempts have been made, the situation has not improved, the XTP process will inform its client application process of the situation, and if so directed, close the connection. Each *sync* packet is issued when a timer expires.

In other situations, communication may have been temporarily suspended or interrupted, and connection closure is not required. In these cases, XTP attempts to re-synchronize the sender with the receiver. Re-synchronization is attempted when the sender has issued a SREQ to the receiver and the WTIMER times out before the receiver's CNTL packet has been received by the sender, as earlier described. The XTP sender process will assume that the packet containing the SREQ was dropped, and transmits another packet containing an SREQ to the receiver — a *sync* packet. If the original SREQ containing data packet is still on the network, two SREQs could arrive at the receiver, both requesting positive acknowledgement of data received, and an updated ALLOC value from the receiver. The receiver complies by outputting two CNTL packets back to the sender. Note that at the receiver, the values for ALLOC, RSEQ and DSEQ may have changed between the arrival of the two status requests. Thus the CNTL packets may contain different information — one outdated and misleading, the other one current, so the sender must be able to distinguish the most current CNTL packet from old ones.

XTP associates each receiver-generated CNTL packet with the SREQ that requested it. When

the sender issues a *sync* packet, it increments a counter value (the SYNC counter for the context), and includes this value in the SYNC field of the outgoing *sync* packet. The SYNC field is located in the control segment, and occupies 4 bytes. When the receiver receives *sync* packets from the sender, it copies the SYNC value from the incoming CNTL packet into the ECHO field of the outgoing CNTL (called an *echo* packet). The sender differentiates between old *echo* packets and the current one by comparing the ECHO value against the current SYNC counter contents. Like SYNC, the ECHO field is 4 bytes in length and located in the control segment.

Note that an ECHO/SYNC match does not guarantee that the *echo* packet is the most current *echo* packet, but from the sender's point of view, this is the best assumption. Consider the undesirable case where a sender's second *sync* packet arrives at the receiver before the first *sync* packet, by taking a different route on the inter-network. The receiver will issue two *echo* packets, but in the wrong order. In the first *echo* packet, ECHO is set to 2, while in the second, more current *echo* packet, ECHO is set to 1. When the first *echo* packet arrives at the sender, SYNC equals ECHO, and the packet is accepted. Although this scenario is possible, it is improbable.

When an incoming ECHO matches the context's SYNC counter value, the sender examines the receiver's current status data. If no retransmissions are needed, and ALLOC has been extended, the sender resumes with data transmission. If the receiver has not extended ALLOC, but there are gaps to retransmit, the sender begins retransmitting the lost data. Otherwise, the sender must wait for the receiver to extend the ALLOC value before proceeding, and must block.

Each time the sender outputs a new *sync* packet, it resets WTIMER, and blocks. If the sender fails to synchronize, or fails to receive an updated and extended ALLOC in a reasonable amount of time, the XTP sender notifies the sending client process, and may terminate the connection. If the sender and receiver reestablish synchronization, the sender quits outputting *sync* packets, and resumes data transmissions.

Sync/echo packets are also used to update the current round trip time (RTT) estimate. The sender sets the TIME field in the *sync* packet to the current time at the sender. When the receiver prepares the corresponding *echo* packet, it also copies the TIME field of the *sync* packet into the TIME field of the *echo* packet. When the sender receives the *echo* packet, it estimates the current round trip time by subtracting the echoed TIME value from the current time. This RTT estimate is used by the sender in setting the duration of the WTIMER. WTIMER is set to twice the RTT. Since XTP acknowledgements are generated at the sender's request (using SREQ), the RTT estimate more accurately reflects the average round trip time than schemes relying on timeout-generated acknowledgements.

XTP bounds the time each packet is allowed to "live" in the network using the time-to-live (TTL) field in the common trailer. The time value is expressed in 10 millisecond "ticks". In outgoing packets, this field is initialized by the user to a given number of ticks (in TCP time-to-live is based on the current RTT estimate). At each hop, the TTL value for the packet is decremented — when the value becomes zero or negative, the packet has exceeded its time to live and is discarded. Note that bounding the time the packet can exist on the inter-network aids in removing packets which can not be delivered due to pathological situations such as host or router crashes.

Since the TTL field occupies 2 bytes, 64K different values are expressible in the field yielding a range in values from zero seconds to 655.36 seconds in 10 millisecond steps. For networks with greater propagation time than 655 seconds, (e.g., a very wide area network) the TTL mechanism must be disabled. XTP allows the TTL mechanism to be disabled by setting the initial TTL value to zero. If a packet arrives with a TTL value of zero, it is assumed that the policy is to bypass the TTL decrement-and-discard step, and the packet is relayed onto the next network with the TTL value still equal to zero.

When transmitting, the sender may use a timer to comply with the receiver's *rate* control requirements for bytes/second (RATE) and bytes/burst (BURST). This timer (RTIMER) must be accurate enough to support the rate control timing requirements for the given implementation. The duration of the RTIMER is set to BURST/RATE seconds. Each RTIMER timeout reestablishes the limit on the maximum number of bytes which can be output on the context during the next RTIMER time period.

XTP requires only one timer at the XTP receiver process, and it is only needed during connection closing as shown earlier in Figure 5. This timer (also called the WTIMER) is set whenever the receiver process issues a CNTL packet with the RCLOSE request set. The timer estimates the round trip time, and if necessary, generates a new RCLOSE request upon expiring.

Each multi-context route requires a special timer called a *Path* timer (PTIMER). In routers, the PTIMER duration may extend for days to allow datagram-type service over stable, infrequently used routes. In each end-node, the PTIMER duration is substantially shorter, and may be measured in minutes or hours.

The timing parameters discussed in this section are listed in Table 9.

TIMER	Duration	Description
WTIMER	2*RTT	Used by sender when re-synchronizing with receiver. A new *sync* packet is transmitted when WTIMER expires. WTIMER is restarted every time a SREQ is issued, and during closing. Used by receiver during closing.
CTIMER	60 seconds	Used by sender to detect dead connections.
RTIMER	implementation dependent (BURST/RATE)	Used by sender to perform rate control.
PTIMER	minutes or tens of minutes	Path timer (one per route in each node). Used by host in managing routes. Each route may support multiple contexts.
PTIMER	hours or days	Path timer (one per route in each router). Used by router to detect dead routes. Each route may support multiple contexts.
Parameter	**Location**	**Description**
RTT	sender's context record	Round Trip Time estimate for context. Estimate based on time elapsed between transmission of sender *sync* packet and reception of associated *echo* CNTL packet.
TTL	common trailer	(Time-to-live). Used to detect and discard packets which stay on the network too long. The sender sets TTL to a number of 10 millisecond ticks when the packet is transmitted. Intermediate routers decrement and monitor the value.
SYNC	control segment	Counter value used by sender to individually mark *sync* packets. The receiver copies the SYNC value into the ECHO field when responding to a *sync* packet. Allows sender to differentiate between CNTL packets.
ECHO	control segment	Field that receiver copies received SYNC into when responding to *sync* packets.
TIME	control segment	Echoed back by receiver from sender's *sync* packet. Used to estimate current round trip time. When the sender receives a TIME echo, it subtracts the echoed TIME from the current TIME to estimate the current round trip delay (RTT).

TABLE 9. XTP Timers and Timing Parameters

5. Addressing Mechanisms In XTP

The aforementioned KEY field is but one of the parameters XTP uses to perform addressing. Addressing occurs on a number of levels. First, consider XTP as a client process to an underlying datalink layer as depicted in Figure 1. At this level, the datalink layer needs a unique *service access point* (SAP) for XTP to separate incoming XTP packets from non-XTP packets. The XTP packet, or frame would be encapsulated inside a datalink layer's protocol data unit (PDU). As the datalink layer process decodes the PDU, it determines the destination to be the XTP server process.

XTP was designed to interface with a variety of datalink layers. In each case, XTP packets must be encapsulated within the PDUs of the underlying datalink layer. This encapsulation must

conform to the requirements of the various datalink layer protocols. For those protocols capable of multiplexing their services among multiple transport layers (say XTP and TCP simultaneously), the datalink layer uses a unique, standardized identifier to distinguish between TCP and XTP packets. In 1990, XTP is expected to be operational on Ethernet, IEEE 802.5, and FDDI; XTP is already operational on top of the User Datagram Protocol (UDP).

Within the XTP layer, each end of a XTP connection must be able to uniquely identify its peer. To complicate matters, XTP's inter-network and multicast capabilities impose additional addressing requirements.

One solution would be to include all relevant addressing data explicitly in each packet. With large internet addresses, this approach would cause substantial per-packet overhead. The XTP approach caches the addressing data contained in the first packet at both the sender and receiver, and uses the KEY as a lookup index into the cache to access the actual addresses as needed. As described earlier, this initial packet is a special *information* packet of type FIRST. The following packets contain only the KEY, resulting in smaller packets because the KEY is encoded in fewer bytes.

In TCP/IP, 14 bytes are used for addressing information in every packet — IP requires an IDENTIFICATION field of length 2 bytes that is used, like XTP's KEY field, to identify the connection uniquely, 4 bytes for the IP source address, 4 bytes for the IP destination address in the IP encapsulation, and 2 bytes each for the source and destination ports in the TCP segment. In XTP, 4 bytes are used to specify the context number. Thus, XTP encodes the addressing information with less overhead per packet.

Medium Access Control addresses (MAC values) uniquely identify nodes on the same local area network. Note that within each XTP process, however, a MAC value may not be unique — if multiple XTP clients are connected to the same remote host, the MAC address for each

connection's context record will be the same. The concatenation [MAC,KEY] of the remote host's medium access control address and any given context identifier uniquely determines the connection.

When LANs are interconnected, packets must travel through routers or gateways and [MAC,KEY] may no longer be unique. As with the KEY, XTP associates an identifier with each route inside a router (the ROUTE value). ROUTE values are included in each packet, and the triple <MAC,KEY,ROUTE> does uniquely identify each context. Inter-network routing and the ROUTE field are further discussed in section 5.1.

The KEY is generated by the node initiating the connection, and included in the FIRST packet transmitted to the receiver. Also included in this FIRST packet are addressing data used to identify the intended receiver. These addresses are contained in a list for comparison with the receiver's address filter. In multicast mode, more than one receiver is targeted for each packet. The appropriate receivers note the arrival of the FIRST packet, and save the context identifier (KEY), the source of the datalink frame containing the packet (MAC address) and the route identifier (ROUTE) in a database associated with the *context record.* Subsequent packets need not contain the destination network address since the triple <MAC,KEY,ROUTE> can be used to lookup the context.

As described in Table 10, the KEY field of the XTP packet common header is 32 bits in length, but the context identifier KEY's value is restricted to a value expressible in 31 bits. The extra bit is located in the most significant bit position, and reserved for determining the direction of the packet — i.e., which end of the connection generated the packet. Packets *sent* from the node which generated the KEY value have the bit set to *zero;* packets *received* at the node generating the KEY value have the bit set to *one.* When the high bit is set, the KEY is referred to as a *return* key. If the KEY in an incoming packet's header is a return key, the receiver can use

the key as a lookup to determine the context for an incoming packet since the receiver generated the original key.

In order to make context lookup faster, the receiver must be able to substitute a value of its own choosing for the newly forming context's KEY. But the new KEY will only be useful if the peer uses it when transmitting packets on this context. The substitute KEY is transmitted back to the context initiator in the XKEY field of the next CNTL packet. The receiving XTP context continues to *output* CNTL packets containing the original KEY (with the high bit set), whereas the sending XTP context will adopt the receiver's requested KEY (also with the high bit set) when transmitting packets. Note that in this case, once KEYs have been *exchanged*, all packets will be using *return* keys — with the high bit set. See Figure 10. Key exchanging is only possible when there is a unique receiver (i.e., keys may not be exchanged in *multicast* mode).

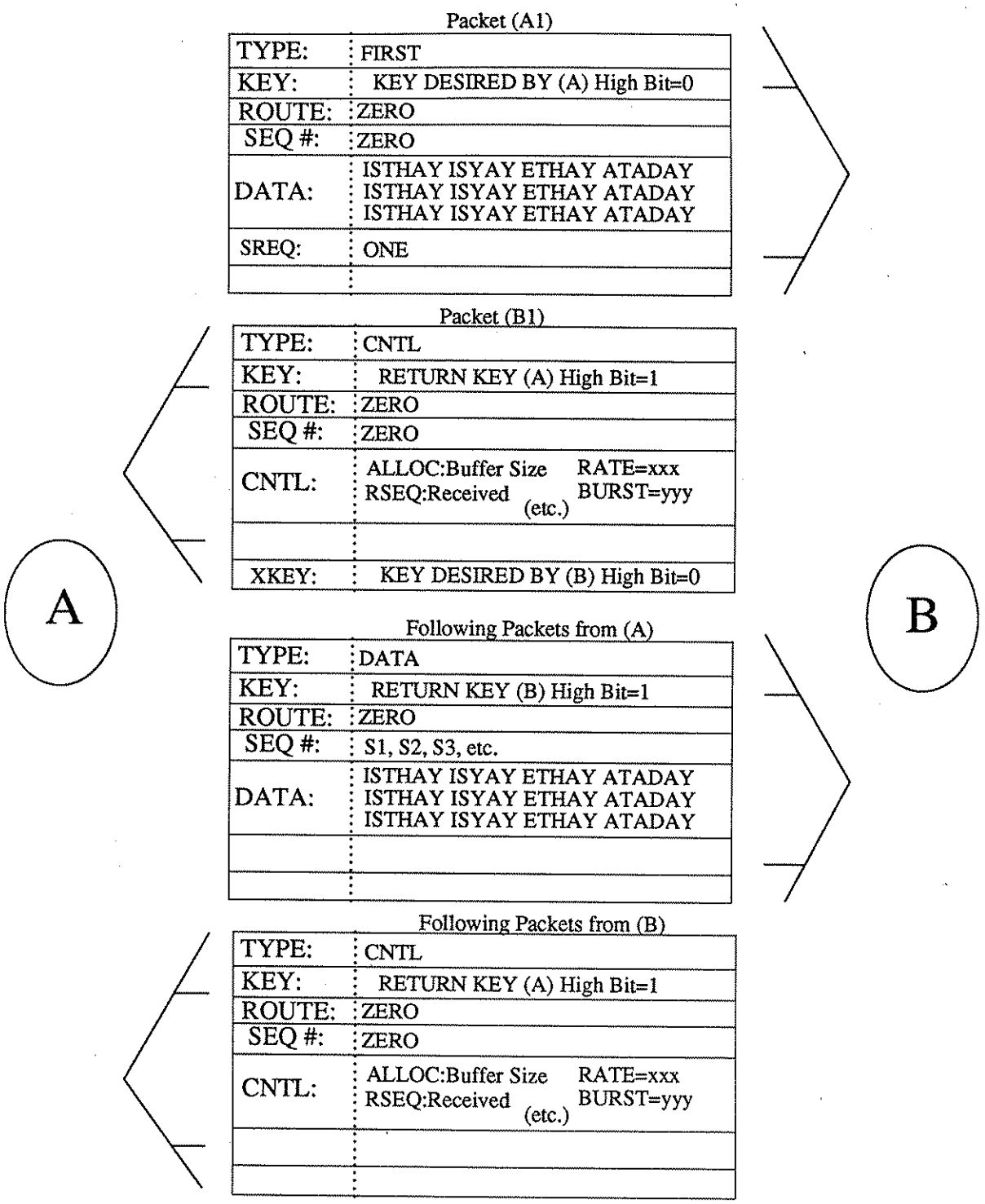


Figure 10. XTP Key Exchanging

In Figure 10, the sending context at node (A) issues a FIRST packet to set up a connection. This packet is labeled Packet (A1), and contains the KEY value that context (A) prefers be used

in returned packets. The XTP receiver associates this KEY (say K_a) with the context record at node (A) corresponding to the connection.

When the packet arrives at (B), the XTP receiver at (B) creates a new context, and saves A's desired KEY value (K_a) in the context record at (B) associated with the connection. All packets returned to (A) on the connection will use the *return* form of (K_a), denoted by (K'_a). (B) decides it would be advantagious to exchange KEYs. Packet (A1) contains a SREQ, so (B) responds with Packet (B1). Note the KEY field value (K'_a), and the XKEY field value (K_b).

When Packet (B1) arrives at node (A), the KEY value (K'_a) it is used to locate the context record for the connection. (A) saves the requested exchange KEY in the context record for out-going packets on the context.

Any additional packets sent in either direction contain the appropriate *return* KEY value for the packet's destination. That is, in the A→B direction packets carry (K'_b) and in the B→A direction they carry (K'_a) respectively in their KEY field.

Parameter	Location	Description
MAC	External to XTP	*Medium Access Control* layer address. The physical address of the network interface for the given host.
KEY Field	XTP header	Uniquely identifies the XTP context at the sender. 31 bit number generated by sender occupies 4 bytes or 32 bits. The highest bit reserved for determining direction of packet.
XKEY Field	XTP control segment	Exchange KEY returned from destination for sender to use in subsequent packets. Uniquely identifies the XTP context at the destination.
ROUTE Field	XTP header	Used by sender when forwarding through routers. Similar to KEY field.
XROUTE Field	XTP control segment	Exchange ROUTE value returned from router for sender to use in subsequent packets. Contains router-generated number used to assist router in determining origin and destination of inter-networked packet.
MULTI Flag	XTP header *flags* field	Indicates that XTP *multicasting* addressing is being used. In this mode multiple receivers simultaneously listen to the same sender. More efficient than setting up individual contexts for each receiver and then duplicating outgoing packets.
ADDRESS Segment	XTP information segment	Contained in the FIRST packet only. This field is variable in length since multiple addresses may be specified. It is further sub-divided into the following fields:
		LENGTH — number of bytes in address field
		FORMAT — network address syntax
		null — to 8-byte-align address data
		ID — undefined in XTP version 3.4
		Actual addresses — depends on FORMAT
DADDR Flag	XTP header *flags* field	Indicates that direct addressing mode is used. In this mode the KEY field is interpreted as a short address.

TABLE 10. XTP Addressing Parameters

The *address* segment included in the FIRST packet contains two main fields — a fixed length descriptor field indicating the addressing format used, and a variable length field containing the actual list of addresses. The *address descriptor* field is 16 bytes long, and contains four sub fields — LENGTH (2 bytes), FORMAT (2 bytes), NULL(2 bytes) and ID (8 bytes). The LENGTH field is the number of bytes in the variable length address segment, including the 2 bytes in the LENGTH field itself. The FORMAT field specifies the address formatting scheme used in the list of addresses. At present, compatible formats are supported for both Darpa Internet and ISO formats.[9] Formats for accommodating Xerox XNS[10] style addresses, U.S. Air Force Modular Simulator project (MODSIM) addresses and Source Route addresses are under study

and should be available in future versions of the XTP protocol.

Included in the Internet compatible format are IP source and destination host addresses (4 bytes each), and source and destination ports or socket numbers (2 bytes each). For 8 byte alignment purposes, the IP address format also contains 4 null bytes.

The ISO address is formed by concatenating the appropriate network layer service access point (NSAP) with the transport layer service access point (TSAP) yielding two 24 byte addresses for a total of 48 bytes — one address for the destination (DSAP), one for the source (SSAP). For each address, the NSAP is positioned in the first 20 bytes. The TSAP occupies the remaining 4 bytes. Since the ISO address length is divisible by 8, no additional null bytes are needed with this formatting scheme.[9] The address descriptor ID field is currently not defined in the XTP protocol definition revision 3.4.

Address *filtering* occurs at the receiver when determining whether to establish the connection requested by the sender of a FIRST packet. Beforehand, the receiving client describes to the XTP receiver process the set of network addresses to which it will connect. When a FIRST packet arrives, the receiver compares its address segment contents against the receiving client's address filter to determine whether to accept the packet or not.

In the event that the network topology is known, and network addresses do not require more than 4 bytes, XTP can use a direct addressing mode. In this mode, the KEY field contains the actual destination address rather than an index used to look up the context. This direct addressing mode is invoked by setting the DADDR flag in each packet. The DADDR flag is located in the common header.

5.1 XTP Inter-Network Routing

When connecting to a process on a remote network, a connection must be established through

one or more routers until the destination network is reached, and finally to the remote host on which the receiver client resides. Packets hop from one network to the next through routers. The router receives the packet on the first network, makes a routing decision, and outputs the packet onto the second network. The router must be capable of determining the appropriate node on the new network to which the packet should be transferred, based on the destination addressing information contained in the packet. As in the single network case, this addressing information can be cached. Figure 11 illustrates the address management occurring when a connection is established on first an adjacent network requiring a single hop (through router R), and second on a remote network requiring two hops (through routers R and H).

The ROUTE field serves a similar purpose to the KEY field. Refer to Table 10. It is located in the XTP common header and is utilized by the router to locate the proper addressing data in its cached address translation map. As with the KEY, ROUTE values can be exchanged between adjacent routers and/or the endpoint nodes. Like the XKEY field, the XROUTE field is located in the control segment of CNTL packets.

When a FIRST packet arrives at the router, the router saves the incoming ROUTE value in the data structure associated with the route upon which the packet is travelling. Packets generated at the router to be returned to the context initiator will use the *return* form of this ROUTE value. The Router has the option of generating its own ROUTE values for the next host or router in sequence to use on the given route. When relaying the FIRST packet towards the destination, the router merely substitutes its preferred ROUTE value in the header, overwriting the original ROUTE value chosen by the context initiator.

Packets arriving at the router from the destination-end of the connection will contain the *return* form of the router's desired ROUTE value. The destination-end of the connection may choose to exchange ROUTE values with the router. If so, it will set the XROUTE field to its

chosen ROUTE value when transmitting its first CNTL packet. The router will note the XROUTE value, and use its *return* form in future packets to the destination-end.

The router relays the CNTL packet towards the sender-end of the connection. In this CNTL packet, the original KEY value and ROUTE value received from the sender-end in the FIRST packet are substituted into the CNTL packet header, both in *return* form. If the router choses to exchange ROUTE values with the sender-end, it creates a second ROUTE number, associated with the address of the destination, and includes this ROUTE value in the XROUTE field of the CNTL packet sent back to the sender node on the first network.

Once the CNTL packet arrives at the sender node, the sender adopts the router's XROUTE value, and includes the *return* form of it in subsequent packet transmissions for the given connection, in the ROUTE field. Thus, the router may use different ROUTE values for packets traveling in different directions. This is illustrated in Figure 11. In the top diagram, node (A) is the sender, node (B) is the receiver, and node (R) is the router connecting the two networks LAN 1 and LAN 2.

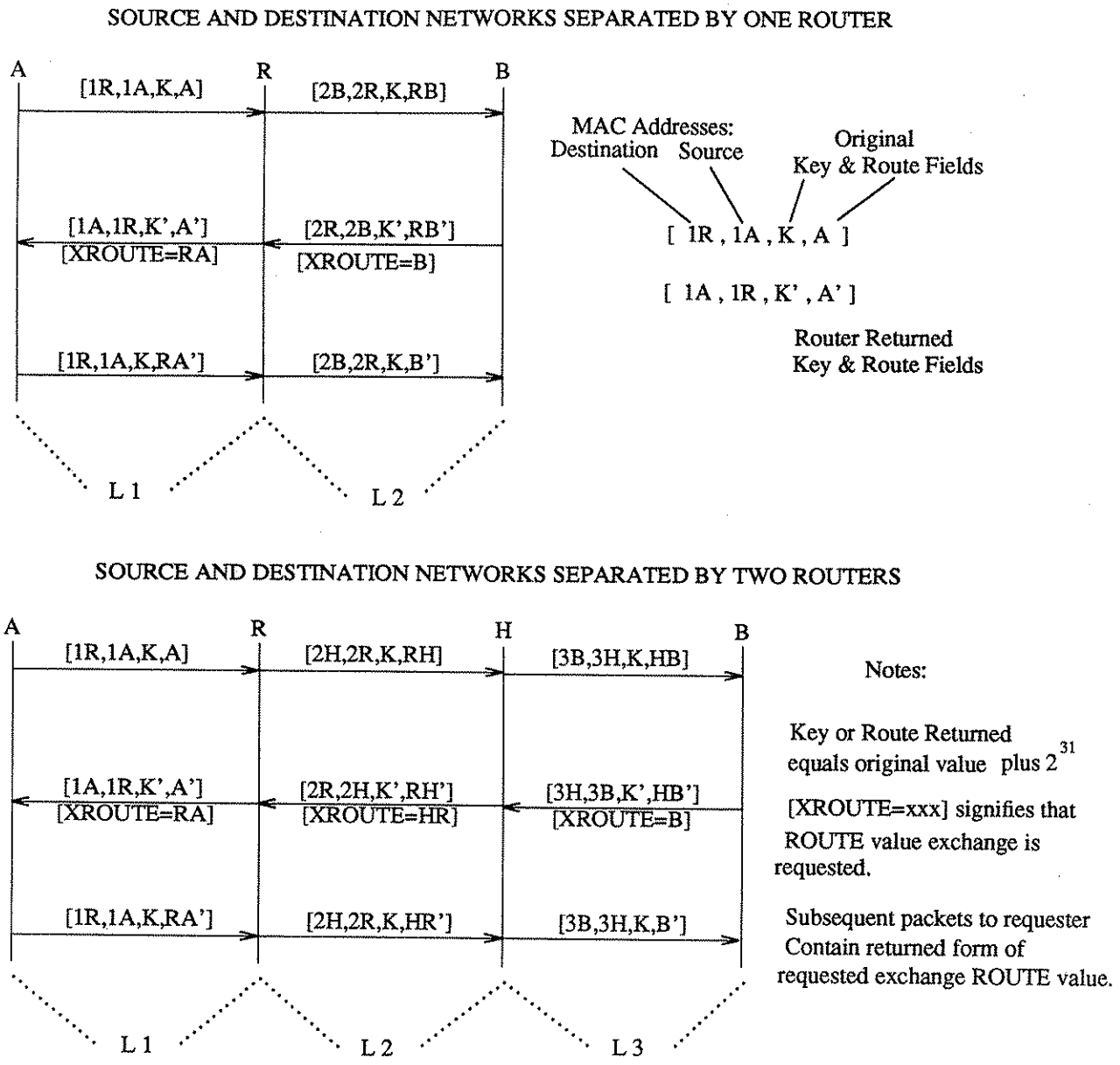


Figure 11. Address Substitution Mechanism in Routers

The bracketed notation [1R,1A,K,A] [XROUTE=xxx] describes the values of address parameters associated with a packet transmitted between adjacent hosts. As described in Figure 11, the first value (1R) is the MAC address of the destination node, the second value (1A) is the MAC address of the sender node, the third value (K) is the value of the KEY field contained in the XTP packet, and the last value (A) is the ROUTE value contained in the given packet. When

the XROUTE is not listed, the field is disabled and set to zero. The XROUTE designator appears below the arrow pointing from the packet's sender to the packet's destination.

In the first packet, transmitted from source (A) to the router (R), the route field is set to A. This is a FIRST packet, and thus contains an *address* segment inside its *information* segment. The router examines the address, and determines that the packet needs to hop from LAN 1 to LAN 2, and that the destination is at node (B) on LAN 2. When the router outputs the FIRST packet onto LAN 2, the KEY field is unchanged, but the router has modified the ROUTE field by setting it to (RB), which is associated with the MAC address of (A). When (B) receives the modified FIRST packet, it accepts the connection request. The return CNTL packet has the high bit set in both the KEY field (*K'*) and ROUTE field (*RB'*) indicating *return* forms for the values. Also, the XROUTE field has been set to B, signifying that (B) wishes to exchange ROUTE values with the router.

When the CNTL packet arrives at the router on LAN 2, the ROUTE value is used to retrieve the addressing data for node (A). Since this is the first transmission from (B) to (A), router (R) generates a new ROUTE value to exchange with (A). This ROUTE value (RA) is stored in the XROUTE field of the CNTL packet returning to (A). It will be used at the router to associate incoming packets from (A) with the MAC address of node (B) on LAN 2. Node (A) notes the XROUTE field value, and includes (*RA'*) as the ROUTE value in all subsequent packets output for the context.

Also depicted in Figure 11 is a scenario with three networks and two routers, (R) and (H). In this situation, packets must make two hops. Note that the routers overwrite the ROUTE field values incoming from packets generated by the other router. New ROUTE field values are generated at each hop.

Figure 12 demonstrates KEY and ROUTE exchanging between two nodes (A) and (B) separated by a common router (R) using the simple packet diagrams of Figure 11. Note that once all exchanges have completed, all KEY and ROUTE values used are in *return* format.

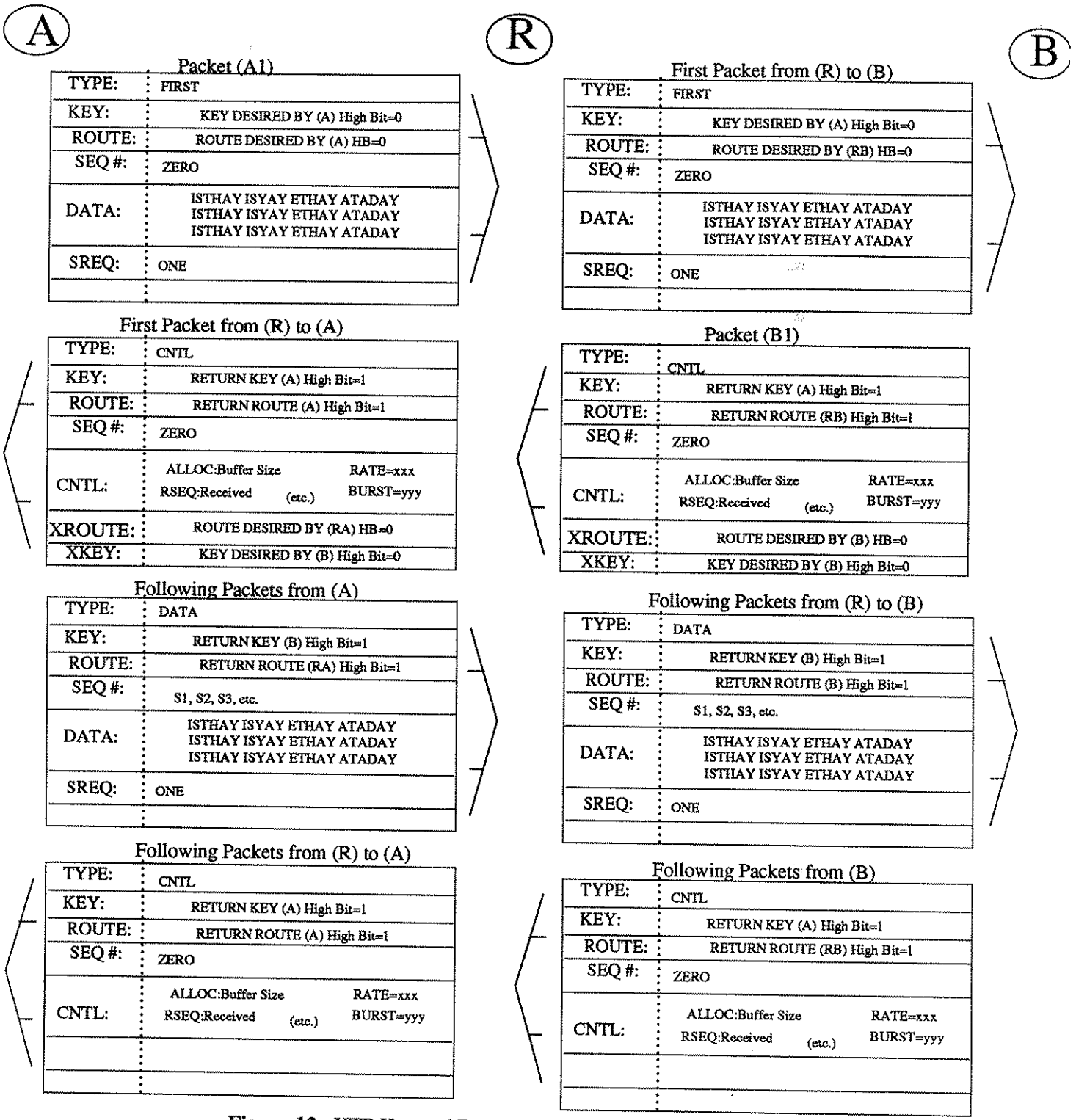


Figure 12. XTP Key and Route Exchanging

As discussed earlier, each individual route can exist for an extended period of time, (i.e.,

perhaps even for days inside routers.) By allowing more than one context to *share* a route, the cost of initializing and maintaining the route can be shared among contexts. Additionally, the *rate* control for the shared route can also be shared. Sharing routes allows the routers to combine redundant table entries in internal routing tables and minimize their space requirements. XTP supports route sharing, and inheritance between contexts.

In XTP, an existing route can be utilized by a newly-forming context by setting the ROUTE value in the header of the FIRST packet to the ROUTE number associated with the particular route. This number is available in the context record of any active context currently using the route.

One complication of route sharing is that the router can not detect when the route is no longer being used without being explicitly requested to *release* the route. In XTP, the special information packet type ROUTE is used by routers and nodes to tear down routes. When a node knows that it is finished using a given route, it issues a ROUTE packet to the router, which contains a RELEASE request embodied in the information segment. The router responds by issuing its own ROUTE packet acknowledging the request and releasing the route.

6. XTP Fragmentation Issues

XTP also supports fragmentation of data packets when necessary. The need arises when two connected networks have different *maximum transmission unit* sizes, as mentioned earlier. In this case, the routers perform the fragmentation transparently. The resulting set of smaller packets are referred to as *fragments*, although they are legitimate XTP frames themselves. Each fragment contains its own header, a portion of the original packet's data segment and its own trailer.

XTP CNTL packets are sufficiently small that they do not require fragmentation. The largest CNTL packet contains a 24 byte header, 16 byte trailer, 40 byte constant subset of the *control*

segment and 16 SPAN groups containing 8 bytes each, also located in the *control* segment. The maximum number of bytes in a CNTL packet is thus 208 bytes plus the media framing.

During fragmentation, the router must refrain from exactly duplicating the original data packet's header and trailer into the smaller fragments because certain option flags are *non-replicatable*. For example, the SREQ bit in the common trailer must not be replicated — if it were, each fragment would solicit its own CNTL packet status response from the receiver, when only one was desired. Partial exceptions are the *first* fragment's header and the *last* fragment's trailer. The first fragment's header is an *exact* duplicate of the original packet's header. All other fragments contain different SEQ numbers, and perhaps other differences from the original header. The last fragment's trailer would be an exact duplicate of the original trailer *except* that the HTCHECK header-trailer checksum is calculated over a different header from the original packet's HTCHECK.

Refer to Table 11 for a list of the *replicatable* and *non-replicatable* flags and option bits in the XTP header and trailer.

Replicatable	Explanation
All Header Option Flags *except* BTAG	These flags contain information which is unchanged during fragmentation, such as whether a direct-addressed priority multicast transmission is underway. They must be copied into all fragments created. The replicatable header options flags are: LITTLE, NOCHECK, DADDR, NOERR, MULTI, RES, SORT and DEADLINE.
Non-replicatable	**Explanation**
BTAG	This flag indicates that *beginning tagged data* is located in the first 8 bytes of the original data packet. Since the *tagged* data is positioned at the start of the data being fragmented, it is copied into the the first 8 bytes of the first packet fragment. The following fragments can not contain *beginning* tagged data. Thus, the original packet's BTAG flag is copied into the *first* packet fragment only. The BTAG field for additional packet fragments is set to zero.
All Trailer Flags	The trailer flags are also non-replicatable for various reasons. They are all copied into the trailer flags field of the last packet fragment only. The trailer flags field in the other packet fragments (i.e., the first and intermediate packets fragments) are suppressed by setting them all to zeroes. Each SREQ or DREQ generates a CNTL packet from the receiver. If they were copied into each packet fragment, the receiver would generate more CNTL packets than necessary. RCLOSE/WCLOSE and END are used in closing the context. Receiving any one of them prematurely would confuse the receiver and violate the closing rules of the protocol. ETAG, like BTAG indicates the presence of *tagged* data, in this case in the last 8 bytes of the original packet's data segment. This data must arrive in order, and thus must appear in the last packet fragment. EOM indicates that the data in the original packet completed a message transfer. If this bit were copied into more than one packet fragment, the receiver would assume more than one message had arrived erroneously. The EOM flag must not be set until the last packet fragment containing a portion of the message. NODCHECK indicates that the packet does not contain a checksum over the information segment. If in packet fragmentation this checksum is not recalculated for each packet fragment, the NODCHECK flag is not replicated.

TABLE 11. XTP Flag Replication During Fragmentation

A method is under development for combining packets at a router which have identical ROUTE fields. The combined packet is referred to as a SUPER packet, and contains a special experimental header referred to as a SUPER header. The individual XTP packets can be recovered if the SUPER packet must be fragmented.

7. XTP Multicast Mode

XTP defines a *multicast* mode of operation where one sender can broadcast the same data stream or datagram sequence to multiple receivers simultaneously (one-to-many). Figure 13 depicts such a case where the sender and all receivers are located on a token ring. The multicast sender is located at node (A), with multicast receivers at nodes (B), (D) and (E). To activate this mode, the MULTI flag in the common header is set, indicating a multicast transmission is in progress.

XTP's multicast mode is similar in operation to the single receiver mode in many respects. The transmitter issues a FIRST packet, and subsequent DATA packets. SREQ is used to solicit CNTL packets. *Error* control is supported using the *go-back-n* retransmission scheme; *selective retransmission* is not supported. Note that in multicast connections the allocated buffer space in each receiver may vary in size. Essentially, data transmission proceeds at the pace of the slowest receiver.

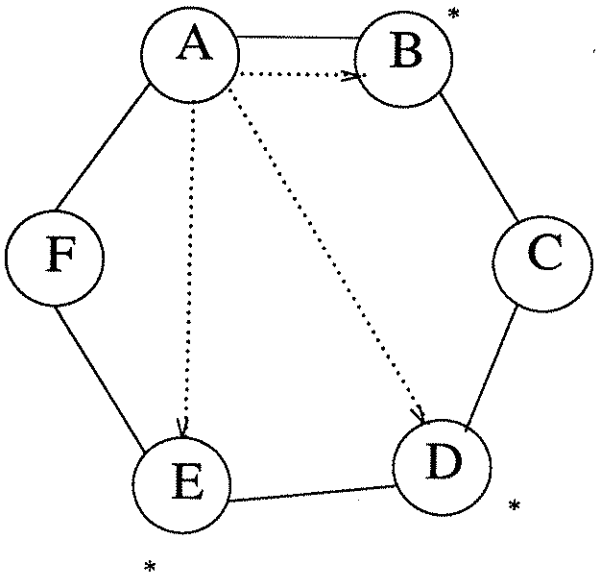


Figure 13. Multicast Transmission on a Token Ring

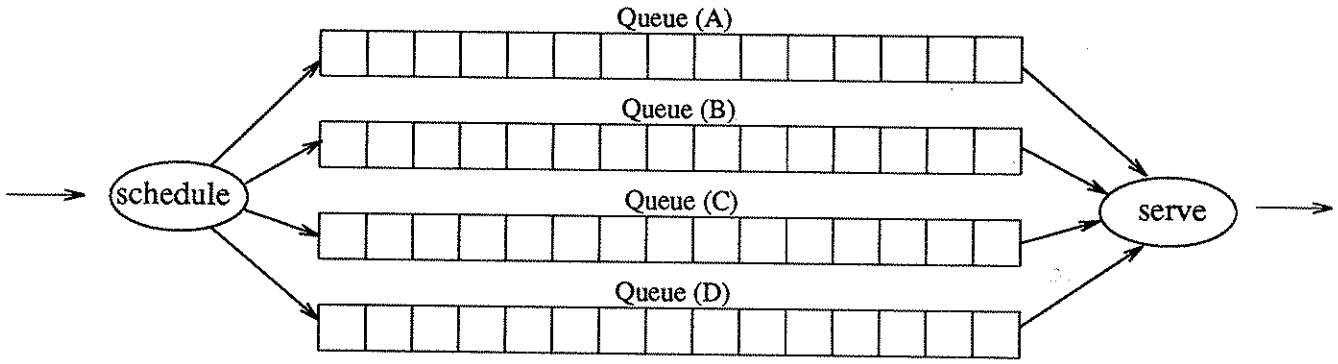
When a multicast receiver detects out-of-sequence data, it *multicasts* the CNTL packet, called a *reject* packet, so that all other receivers on the connection realize that an error has occurred.

If the multicast involves a large number of receivers, the sender will be inundated with *reject* packets as all receivers clammor to announce the error. To dampen this effect, XTP requires receivers to refrain from sending the multicast *reject* packet when aware that the sender has been properly notified. The receivers monitor the network for other *reject* packets during the time the packet is being prepared and waiting for transmission. If another *reject* packet arrives, destined for the sender on the same multicast context, the receiver compares its own RSEQ value to the one contained in the newly arrived packet. RSEQ is significant because this is the next byte the multicast receivers will accept — remember, no gaps are allowed in multicast mode. If the receiver's own RSEQ number is greater than or equal to the packet's RSEQ number, the receiver refrains from sending its own *reject* packet. In this case, the rollback requested in the existing *reject* packet covers the request at the current receiver also. If, on the other hand, the receiver's own RSEQ value is smaller than the packet's, the receiver outputs its own *reject* packet. The basic idea is to guarantee reliable reception at all receivers of the data stream, without complicating the sender's task.

XTP also allows the multicast mode to operate in a less reliable "no error" mode indicated by setting the NOERR bit flag in the common header. In this mode, receivers discard garbled packets, and inform their host of the occurrance, but no *reject* packet or retransmission scheme is used. This technique is appropriate for, say, broadcasting sensor data in a control system — the data are generated continuously, and a particular lost value is quickly replaced with a more current reading.

8. Prioritization Issues In XTP

XTP supports prioritization of packet processing at both the sender and receiver using *preemptive priority scheduling*. As packets arrive for processing, they queue for service when the server is currently unavailable. A preemptive scheduler determines the priority level of the arriving packet, and places it at the end of the appropriate queue. See Figure 14. In this scheme, each queue is associated with a specific priority level, and the server prefers to service packets from the highest priority queue whenever possible. Thus, if the server is currently processing a low priority packet as a higher priority packet arrives for service, the server is *preempted* from processing the lower priority packet and begins processing the higher priority packet. Only after all higher priority packets have been completed or blocked will the server return to the low priority packet. The granularity of pre-emption (i.e., whether on a *byte, frame,* or *message* basis) is currently under study.



PRIORITY RANKING:
(A) > (B) > (C) > (D)

Figure 14. Preemptive Priority Scheduling Among 4 Queues

In Figure 14, packets with the highest priority are placed in queue (A), packets with the next to highest priority are placed in queue (B), and so on. If queue (A) is non-empty, the server will choose its next packet to process from queue (A), regardless of how long other packets have been waiting. Thus, the multiple queues essentially *re-sort* the arriving packets.

In XTP, two preemptive schedulers exist — one for incoming packets, and one for outgoing packets. For both the reader and sender prioritization schemes, XTP supports 2^{32} different priorities, or 2^{32} different queues. (This will most likely be implemented by using position in a single queue.) Each context is associated with a particular priority level. Multiple contexts can be at the same priority level simultaneously.

For outgoing packets, the packet waits for access to the transmitter in the appropriate output queue. The priority level is encoded into a 4 byte integer and placed into the SORT field before transmission. The SORT flag in the common header is set to one to indicate that the packet contains a SORT value. When the packet arrives at the remote receiver, the SORT field is examined, and the packet is placed in the input queue corresponding to the packet's priority.

In XTP, the priority level is inversely proportional to the value of the integer encoding — i.e., larger SORT field values have lower priority. This scheme is static, in that the priority level remains constant as the packet travels through the network. XTP also supports a dynamic preemptive scheduling scheme based on *deadline* times and synchronized system clocks with 100 microsecond resolution. In this mode, the original SORT field value represents a future clock time (the *deadline)* whose priority is proportional to the immediacy of the deadline. As the system clock time advances towards the deadline, the packet's priority level increases. As with the static SORT mode, lower SORT values also correspond to higher priority levels, and are used to determine the queue into which the packet should be placed.

The two scheduling schemes just described are not allowed to co-exist on any given XTP network. Each network may utilize one or the other, *but not both* simultaneously. Alternatively, priority operation may be disabled altogether.

Table 12 further describes the three parameters which control priority scheduling in XTP packets.

Parameter	Location	Description
SORT	header option flags	Flag used to indicate preemptive priority scheduling is active. When SORT is set to one, the SORT field value (see below) is interpreted as a priority level. The *type* of scheduling is determined by the value of the DEADLINE flag.
DEADLINE	header option flags	Flag used to indicate *which type* of preemptive priority scheduling is currently being used on the XTP network. Possible types are dynamic *deadline* scheduling, and, *static* scheduling. When DEADLINE is set to *off*, the SORT field is interpreted as a static priority. The packet's priority remains at the same priority level until it arrives at its destination. *Lower* SORT field values correspond to *higher* priority levels. When DEADLINE is set to *on*, the SORT field value is interpreted as a future clock time at which the packet's *deadline* will occur. The clock's resolution is 100 microseconds. At each hop, the SORT field value is compared to the synchronized clock time to determine the packet's current priority. If the deadline passes and the packet is undelivered, the packet's priority drops to zero.
SORT	header (4 bytes)	This field contains the packet's priority level in both the SORT and DEADLINE prioritization methods. With 32 bits, the field provides over 4 billion distinct priority levels. With DEADLINE scheduling, the maximum time-until-deadline expressible is approximately 12 hours.
SORT and DEADLINE scheduling are mutually incompatible, and therefore can not be used on the same XTP network simultaneously.		

TABLE 12. XTP Prioritization Control Parameters

To enable deadline priority, both the DEADLINE flag and the SORT flag must be set to on. The SORT flag enables processing of the SORT field, while the DEADLINE flag determines the scheduling discipline used. In deadline scheduling, it is possible for the deadline to arrive before the packet has reached its destination. This would be detected if the SORT field became older than the current time. In this case, the packet is not discarded but instead becomes low priority. XTP will attempt to deliver packets with expired deadlines only after all other packets have been processed.

9. Detailed Format Descriptions for XTP Packets

Refer to Figures 15 and 16 for details on the layout of the XTP packet. In Figure 15, the format of *control* packets is shown, including the *header*, *control* and *trailer* segments. In Figure 16, the format of *information* packets is depicted. In Figure 17, the *command word* field in the common header is illustrated to show the bit location of each of the header's option flags. Likewise, Figure 18 details the bit locations of each flag in the common trailer's *flags* field. Note that the trailer's *flags* field and *align* field together occupy 2 bytes. The *flags* field contains 10 bit flags. The remaining 6 bits are the *align* field.

The *information* segment may include address descriptors, addresses, beginning tagged data, ordinary user data and ending tagged data. In the *flags* field are found additional option flags for the XTP protocol.

As seen in Figures 15 and 16, the fields group together naturally into 8 byte blocks for both XTP packet types. In XTP, all packets are multiples of 8 bytes, and the trailer must be aligned on an 8 byte boundary. Thus, some variable length fields occasionally have null bytes appended on to the end of the field to enforce the 8 byte alignment policy.

An interesting feature of the XTP packet format concerns the order in which bytes are arranged in a word for various computers. This ordering affects the sequence in which the bytes are placed onto the network. Bytes within a word can either be arranged from highest to lowest address, or from lowest to highest address. Different equipment manufacturers support different byte orderings. Since no standard exists, XTP provided a natural way to support both orderings transparently.

These two orderings are referred to as *big-endian*, and *little-endian*. In *big-endian*, the most significant byte is transmitted first. In *little-endian* the least significant byte is transmitted first. Thus *big-endian* transmits from most significant byte to least significant byte, and *little-endian*

transmits vice versa.

The problem is to encode in each packet an indication of which byte ordering was used by the sender to prepare the packet, and in such a way that a receiver adhering to either byte ordering scheme can determine the correct order of the bytes. This was solved in XTP using two bit flags. The position of the two flags were chosen so that they map into each other even if the byte ordering is guessed incorrectly. The two flags are both set to the same value by the sender. These flags are called the LITTLE bits, and are found in the highest and lowest byte of the common header *command word* (refer to Figure 17). When the LITTLE bits are set to one, the sender issued the packet using *little-endian* byte ordering. If the LITTLE bits equal zero, the packet is in *big-endian* format. If necessary, the XTP receiver process remaps each sequence of 4 bytes into the ordering preferred by its host.

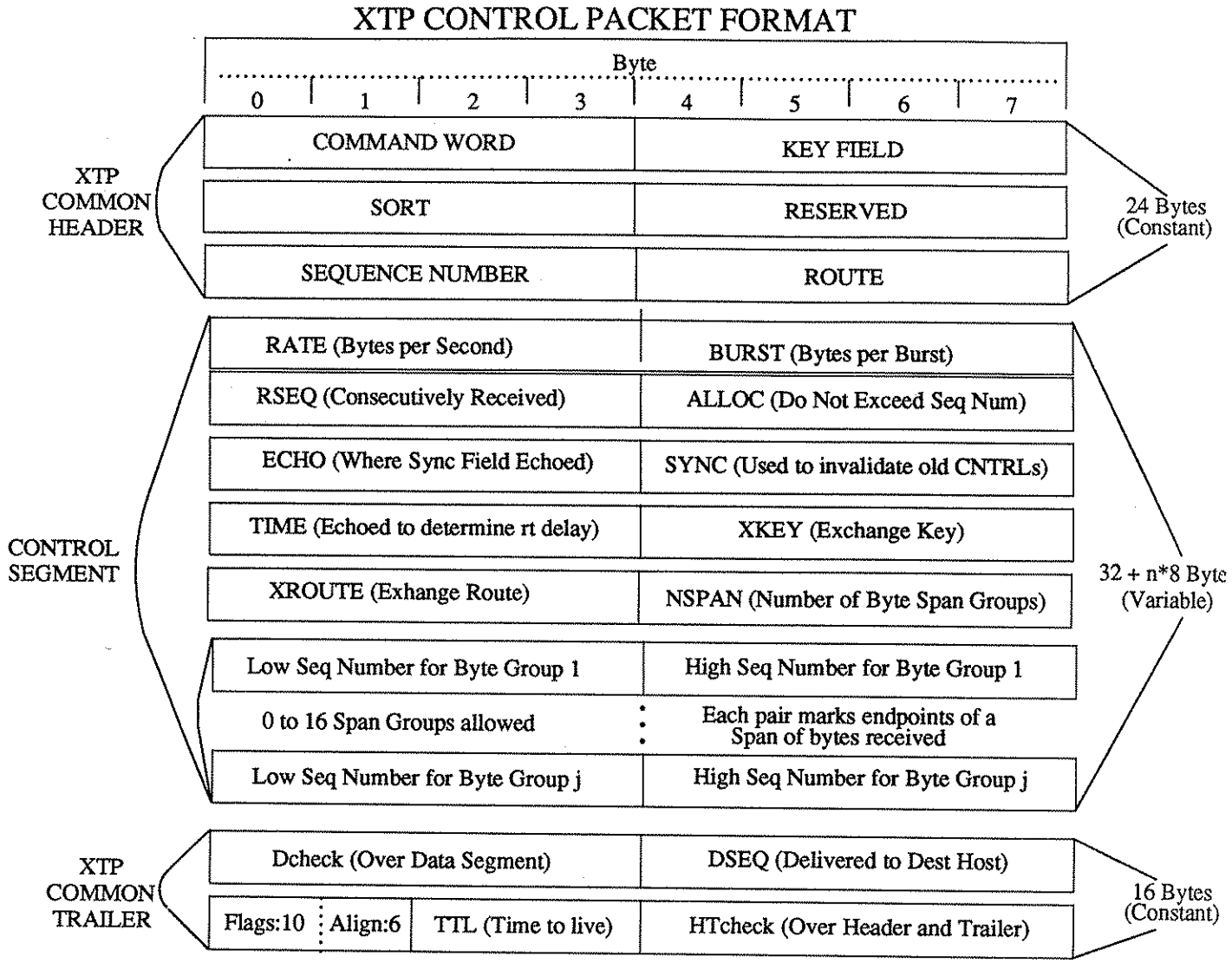


Figure 15. XTP Control Packet Format

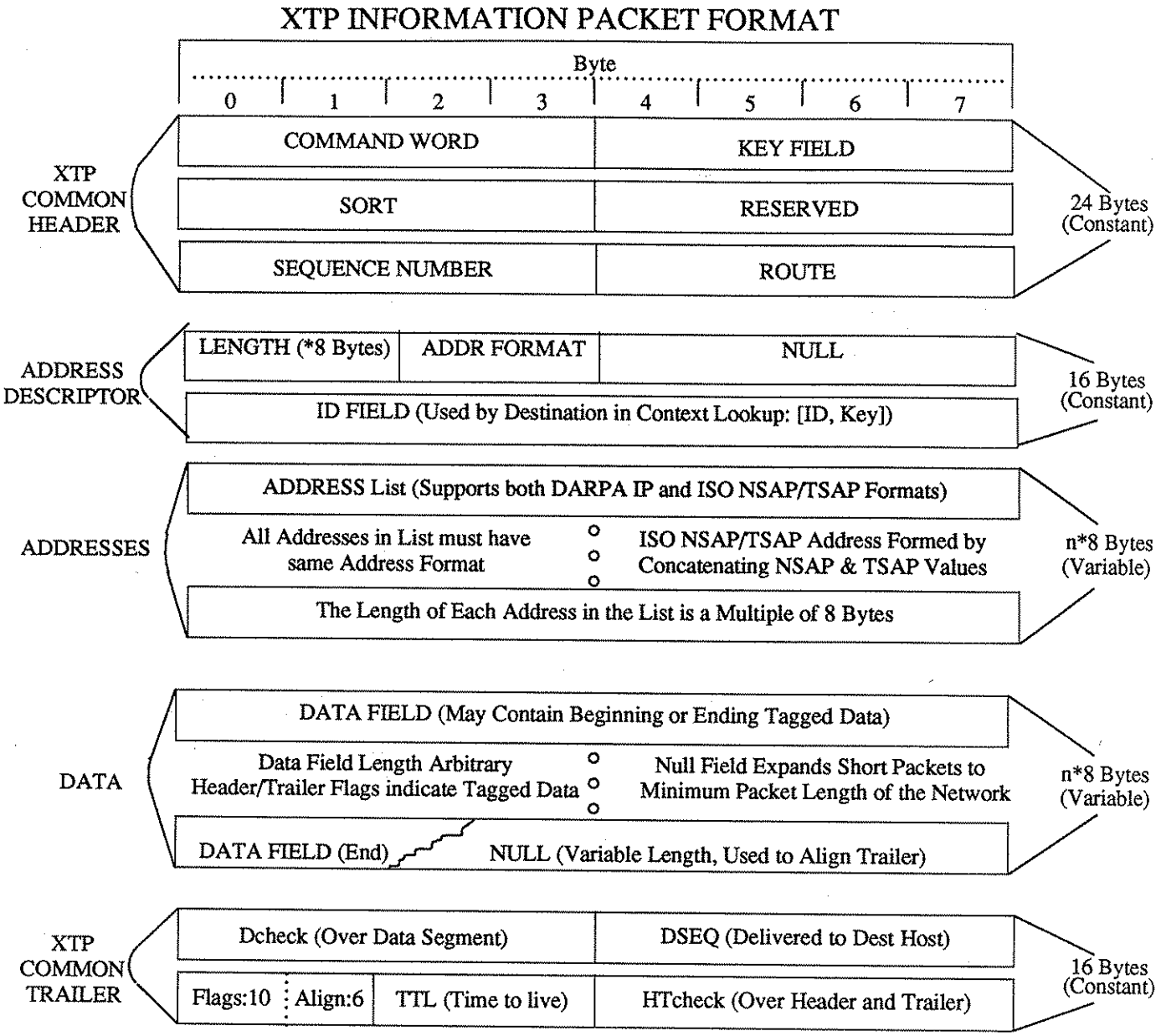


Figure 16. XTP Information Packet Format

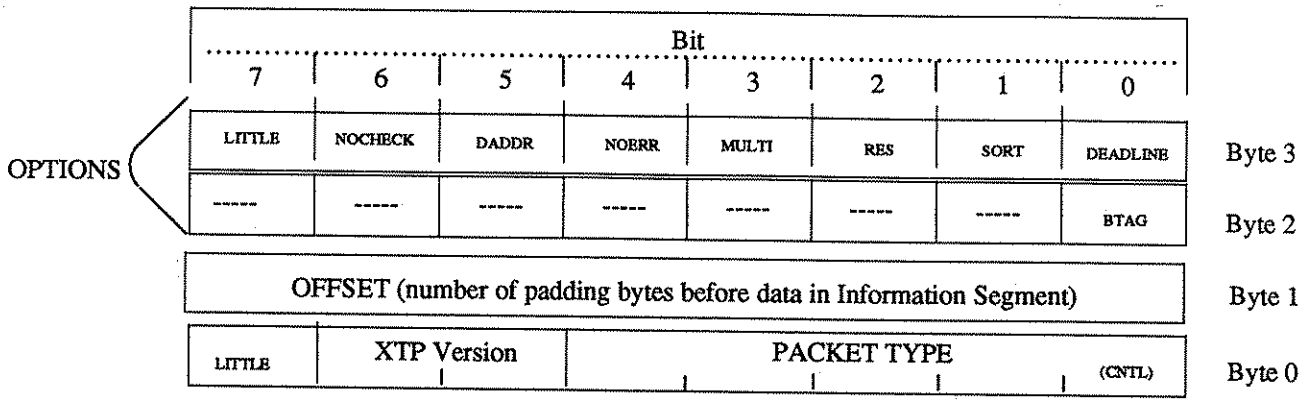


Figure 17. The Command Word — The First Four Bytes of an XTP Packet

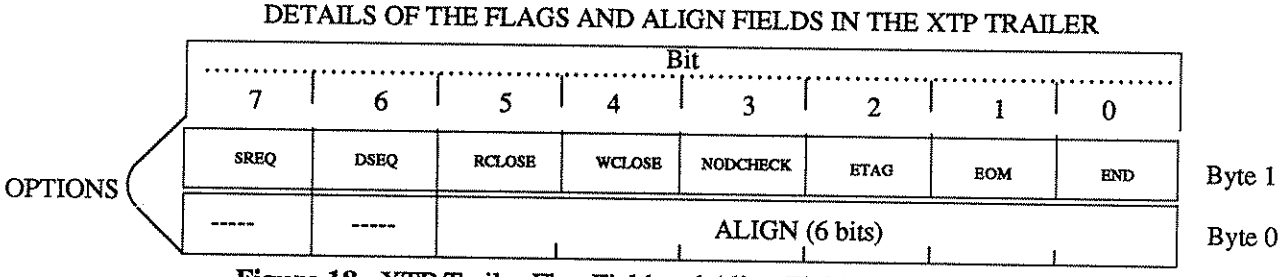


Figure 18. XTP Trailer Flag Field and Align Field Format

REFERENCES

1. "XTP Protocol Definition 3.4", Protocol Engines, Incorporated, 1900 State Street, Suite D, Santa Barbara, California 93101, 1989.

2. Chesson, Greg, "The Protocol Engine Project", UNIX Review, Vol. 5, No. 9, September 1987.

3. Chesson, Greg, "Protocol Engine Design", USENIX Conference Proceedings, Phoenix, Arizona, June 1987.

4. Comer, Douglas, **Internetworking with TCP/IP**, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

5. Stallings, William, **Handbook of Computer Communications Standards, Volume 1: The Open Systems Interconnection (OSI) Model and OSI-Related Standards,** Macmillan Inc., 1987.

6. Jacobsen, Van and Braden, R.T., "TCP Extensions for Long-Delay Paths", Request for Comment 1072 (RFC 1072), 1988.

7. Cheriton, David, "VMTP: Versatile Message Transaction Protocol *Protocol Specification*", Preliminary Version 0.6, Stanford University, 1988.

8. Clark, David, and Lambert, Mark, "NETBLT: A Bulk Data Transfer Protocol", Request for Comment 998 (RFC 998), 1987.

9. ISO 8348 International Organization for Standardization, **Addendum 2: Covering Network Addresses**

10. Hutchison, David, **Local Area Network Architectures**, Addison-Wesley, Wokingham, England, 1988.