1. Introduction

1.1 Background

The Xpress Transfer Protocol (XTP) is a high performance protocol also called a *transfer layer*, after the French model of a layer three and four architecture †. XTP has been pursued as part of the Protocol Engine® project that has been developing very large scale integration (VLSI) circuits dedicated to protocol processing and network interfacing ‡.

The XTP design is motivated by the needs of contemporary and future distributed, real-time, transactional, and multi-media systems. Existing protocols, e.g., TCP and ISO TP4, do not meet all of the perceived needs of such systems. The major issue is that they provide "traditional" end-to-end and router functions, whereas distributed systems have additional needs. Distributed systems depend on remote procedure calls and rapid request/response operations, leading to requirements for reliable datagram mechanisms, fast connection setup, and transaction support. They may also require multicast capabilities. Real-time systems need to control latencies and delays, leading to requirements for scheduling events within protocol layers. Multi-media systems source and sink isochronous streams, leading to requirements for bandwidth allocation and stream synchronization, and sometimes multicast support. All of these kinds of systems must operate in a heterogeneous multi-network environment, leading to requirements for low-latency and high-bandwidth internetworking with both switched and non-switched media. XTP meets these and other needs by providing:

- implicit connection setup
- compatibility with standard addressing schemes
- key-based addressing and routing lookups
- switch-like routing
- message priority/scheduling for real-time systems
- support for encapsulation and convergence protocols
- reliable multicast mechanism
- real-time reliable datagram capability
- both flow and rate control
- selective retransmission and acknowledgement
- fixed-size 32-bit-aligned frame design
- traditional stream capability

At the core of XTP is a minimal mechanism, or *lightweight transport* that can provide both end-to-end reliable, sequenced, data delivery as well as switching or routing service between networks. XTP is designed for either a software or VLSI execution environment.

† "Military Real Time Local Area Network: A Reference Model (Transfer Layer)," French Ministry of Defense Standard, GAM-T-103, February 9, 1987.

‡ See "Protocol Engine Design" in *Usenix Conference Proceedings* June 1987, p. 209; "The Protocol Engine Project" in *Unix Review*, September, 1987; also "XTP/PE Overview" in *Proceedings of the 13th Conference on Local Area Networks*, IEEE Catalog Number 88CH2613-8, October, 1988, and "The Evolution of XTP", in *Proceedings of the Third International Conference on High Speed Networking*, North-Holland, 1991.

This document defines XTP message formats and peer-to-peer message exchanges. It assumes the reader is familiar with network protocols such as Delta-T, VMTP, NETBLT, and IP for the motivation, philosophy, and explanations that are largely omitted here. It is also assumed that the reader is familiar with the C programming language.

1.2 Synopsis

Section 1 of this document consists of background and introductory material. Except for the glossary of terms it is separate from the protocol specification which is contained in sections 2 and 3.

Section 2 defines the syntax of XTP - i.e. the packet formats, messages, addressing conventions and encapsulation formats - and also defines the meaning and interpretation of control fields and state elements in protocol messages. In some cases the definitions in section 2 include procedures for generating or interpreting specific fields. Tables and figures that define the syntax of XTP packets are collected together in Appendix E for quick reference.

Section 3 defines the end-to-end protocol procedures. The description is modular and reflects the structure of the design. The major sections are management procedures, data flow procedures, multicast procedures, and timers. Subsections within each of these major sections define the different protocol mechanisms such as flow control, rate control and error control.

Several Appendices that further explain or enhance sections 2 and 3 are attached to the document. These include a description of the checksum function, detailed description of multicast heuristics, state tables for connection management, a sample application interface, and the summary of packet structures.

1.3 XTP Overview

XTP sets up full-duplex bidirectional virtual circuits between end systems. The XTP state at each end system is called a *context*. Two or more communicating contexts is called an *association*. The transmission model is a byte stream plus a capability to transparently insert record boundaries at the sending end of a stream that are faithfully reproduced at the receiving end.

XTP's virtual circuit model includes both implicit connection setup (as in wormhole routing or cut-through switching) and explicit connection setup (as in TP4 or TCP). The connection shutdown procedures define several options including graceful shutdowns, abbreviated shutdowns, abortive shutdowns, and transactions where a connection may consist of a two-packet exchange.

XTP does not define a new address family, but defines a structure, called an *address segment*, where the address structures from other protocol designs may be incorporated into XTP syntax. Address segments appear only in the XTP packet that sets up a connection. This initial packet is called a FIRST packet. All other packets that follow a FIRST packet incorporate two 32-bit fields, the *key* field and the *route* field, that "represent" the address segment contained in the FIRST packet. The key and route fields provide a way to quickly steer received XTP packets to their destinations.

There are two basic packet formats in XTP (section 2.2.5). Packets are referred to as either *control packets* or *data packets*. Control packets transmit the state of a context from one system to another. Data packets carry application data. There are two main data packet types: the FIRST packet which sets up an association and may carry application data, and the DATA packet which moves application data.

All of the bitflags that control the operation of XTP are concentrated in one field in the header, called the *cmd* word. Fourteen flags are defined. Three flags - WCLOSE, RCLOSE, and END - control connection shutdown. Two flags - SREQ and DREQ - control the acknowledgement policy. Two flags - EOM and BTAG - are accessed only by application processes and are carried end-to-end by XTP as a service. The remaining seven bitflags control the end-to-end operating modes. Examples include enabling or disabling error control or flow control, or enabling multicast mode.

XTP flow control is based on 32-bit byte sequence numbers and a 32-bit sliding window. The sequence numbers can be augmented, or tagged, with a second 32-bit field. XTP also provides rate control whereby an end system or intermediate system can specify the maximum bandwidth and burst size it will accept on a connection or on a group of connections related by the same *route* field. The address segment provides for requesting bandwidth when connections are set up so that both end systems and intermediate systems can manage their real-time resources and eliminate certain kinds of congestion.

Error control in XTP incorporates positive and (when appropriate) negative acknowledgement to effect retransmission of missing or damaged data packets. Retransmission may be either go-back-N or selective. The retransmission algorithms are conservative: only data that is shown to be missing via exchanged control messages may be transmitted. This avoids spurious and possible congestion-causing retransmissions. The error control algorithm, while basically conservative, can also be aggressive: a quick-acting error notification is provided. An implementation can be highly concurrent since adequate state information is communicated between end systems to manage the parallel movement of data, retransmitted data, and state exchange information.

The routing capability in XTP is switch-like in behavior. For this reason XTP may be well-suited to operating environments that include hardware switches as well as LAN and router technology. The routing descipline defined in XTP specifically provides for intermediate systems to access and participate in end-to-end bandwidth allocation. The XTP routing discipline eliminates fragmentation and reassembly. Instead XTP determines the maximum common frame size between end systems and limits output transmissions to that size.

XTP specifies techniques for extending error control to a multicast environment where one multicast sender transmits a stream of data to many multicast readers. The error control algorithm in multicast is identical to the unicast algorithm - retransmission after state exchange - although additional sophistication is required to manage state variables and achieve continuous streaming.

1.4 Glossary of Terms

The following terms, listed alphabetically, are used throughout this document. Figure 1-1 illustrates some of the most commonly used terms.

application — the software running in the higher-layer client that communicates with XTP.

allocation — the do-not-exceed sequence number that is sent by a receiver to a sender in order to limit the flow of data.

association — two or more **contexts** connected by an active data stream.

breakfust — /‘brek-fuhst/ **n: 1.** the first meal of the day. *exception:* Peninsula Fountain & Grille in Palo Alto has been serving it all day since 1923. **2.** usually served as eggs, pancakes or cereals (cold or hot). *exception:* PF&G serves it with soups, chocolate malts or sandwiches if necessary. **3.** often served with the morning newspaper and a smile. *exception:* PF&G waitpersons only smile if they’re in a good mood.

buffer — an area of host memory, usually contiguous, dedicated to sending and receiving network data.

bulk transmission — an imprecise term that connotes both large volumes of data, e.g. images and entire data bases, as well as a requirement for high bandwidth.

coalesce — the inverse of fragmentation, whereby packets are grouped to form larger packets.

command/response handshake — a message exchange between two hosts where once the initiating host has sent the command message, it repeatedly retransmits that message (optionally controlled by a timing mechanism) until a response is obtained from the intended receiving host.

concentration — a form of **multicast** connection where more than one host transmits to a single receiving host.

congestion — an overload phenomenon observed at gateways and other parts of a network where the data rates of numerous senders may combine to overrun a receiver.

connection — a long-lived **association**.

context — the set of state variables for an instance of XTP at an **endpoint** (host); one half of an **association**.

datagram — a term usually denoting unreliable, connectionless service over which unacknowledged messages, possibly multi-packet messages, may be transmitted. However, the term is also used to mean the transmission of a single message. Datagrams may be further qualified by delivery and error control semantics. They may be *reliable* or *unreliable* depending on whether retransmission is provided. They may be *acknowledged* or *unacknowledged* depending on whether delivery confirmation is provided.

data stream a simplex, sequenced data flow. An **association** consists of two data streams, one in each direction.

don't care a term used to describe a protocol field or bit that may contain any value.

endpoint an XTP host that is either the originator of a packet or the final (peer) destination.

end-to-end from one **endpoint** to another **endpoint**.

error control the method, usually retransmission, used to correct transmission errors or buffer overrun.

flow control a method of restraining the volume of information that may be sent. A receiver typically gives a do-not-exceed byte sequence number to a remote host. Window-based flow control, one of the flow control mechanisms used by XTP, is based on an *allocation* sent by a remote receiver to a sender.

fragmentation the operation of breaking a packet into smaller ones at a **switch** or gateway. For example 4K FDDI packets might be fragmented into four 1K Ethernet packets at an FDDI/Ethernet gateway.

go-back-n retransmission the simple retransmission technique wherein output is restarted from an earlier point in the output stream before errors were observed at the receiver. This technique does not discriminate between retransmission of missing or damaged packets and duplicate transmission of undamaged packets if both kinds of packets are included in the go-back-n sequence number range.

host an **endpoint**.

isochronous continuous data traffic such as generated by an audio or video source.

link a direct hardware path. Two machines on the same Ethernet cable or FDDI ring, or connected by RS-232 are examples of machines connected by a single link. A pair of machines on two networks, say an Ethernet and an FDDI, where the two networks are connected by a bridge or gateway device are separated by *two* links — one on the Ethernet and one on the FDDI.

MAC address a physical address. The term Media Access Control (MAC) is defined by IEEE 802.x standards, but used here in a broader sense to mean any physical address.

MBZ must be zero.

MPL Maximum Packet Lifetime is defined in the Delta-T protocol specification. It is the longest time a packet can exist or is estimated to exist in the network.

Maximum Transmission Unit — or MTU. The maximum size for a *data* segment of an XTP packet within a specific physical medium. For example, on 802.3, the MTU is (1500 - 40 - E), which is the maximum 802.3 data field size (1500) less the size of the XTP Header and Trailer (40) less any additional encapsulation overhead (E).

message — one or more buffers of data as defined by an **application** program. The size is essentially arbitrary, being limited by available memory and 32-bit descriptors. XTP delivers messages from sender to receiver, preserving message boundaries if required.

multicast — a context in which one host sends to more than one receiver. May include **concentration** where more than one host transmit arbitrary messages to a single host.

multipath — more than one **path** used for an **association**.

network address — a bit string, usually representing an hierarchical number, that identifies a particular host on a network. There exist different, conflicting standards for network addressing schemes. In this document, the term is used to denote all the collective bits needed to direct a packet to an application at an endpoint.

path — a catenation of **route** values, starting at one endpoint and ending at another, that provides an **association** between two **context**s.

physical address — the **MAC** identification number for a network interface. For example, for an Ethernet environment, the physical address is a 6-byte (48-bit) number.

rate control — the technique of limiting the rate at which a sender is allowed to transmit data, usually by enforcing a time delay after each packet and/or a time delay after each burst of packets. Rate control can limit congestion in the center of a network, i.e., at gateways, as well as limiting overruns at receivers.

real time — the ability to complete protocol processing functions within the transmission time of a packet so that protocol processing never falls behind network traffic.

reservation mode — a term used by XTP to describe a flow control behavior wherein a node is output-flow-limited to the amount of receive buffer space committed by the remote **application** program. The **allocation** used in this method of operation is often different from, and may be considerably greater than, the allocation commitment that would be made by an ordinary XTP implementation. This mode of operation also constrains an XTP host from sending at any time that the remote application has no outstanding receive buffers. This form of control is demonstrated by both the NETBLT and VMTP protocols.

route — an arbitrary value selected by an XTP implementation. Within an XTP **switch** connecting two or more networks, a *route* value on an incoming packet selects an exit port for forwarding the packet. *Route* values are assigned by the **switch** during **path** setup.

rtt (round trip time) - the time between when a sender transmits a packet and when it receives an acknowledgement for that packet.

selective retransmission an error control technique wherein only damaged or lost packets are resent. A receiver must store out-of-order packets, advise the sender of the data that needs to be resent, and reorder the data stream or buffer area as retransmitted packets arrive.

switch a node, located between **endpoints**, that forwards packets along a **path**.

transaction a term which has many definitions. In the world of transport protocols it generally implies a command/response handshake (defined above).

2. Packet Structures

2.1 Notational Conventions

This document utilizes notation, terminology, and illustrations to produce a consistent protocol description. The conventions are described below.

- Terminology:
  The following terms refer to data objects within a packet: **word** indicates a 4-byte (32-bit) object, **field** refers to an object of any size, **bitfield** denotes a field contained within a word, and **segment** denotes an object consisting of one or more fields.
- Unless specified otherwise, packet structures are discussed and portrayed with the most significant bit and/or byte on the left.
- Illustrations:
  Within illustrations, objects drawn as ellipses contain additional structures; objects drawn as rectangles contain no substructures (see example below).
- Notation for bitfields:
  - Bits are referred to by name and are annotated in capital letters of a distinguishing font style: e.g., `NAME`.
  - Within illustrations, vertical lettering (representing the label/function) beneath a numeral indicates a single-bit field.
  - Within text, multiple-bit fields are labeled in lower-case, italicized characters: e.g., *name*.
  - The number of bits in a bitfield, when designated, is with colon syntax: e.g., *name*:7.
  - Within the text, the meaning of certain fields is determined by the setting of a designated bit within the field. An apostrophe convention is used with these fields to indicate whether the bit is set. The field with the bit set is shown as "primed" by placing an apostrophe (') after the field: e.g., K is an unprimed field and K' (K-prime) is the field when the most significant bit is set.
- Notation for bytes:
  - Objects not specifically designated as single-bit fields represent sections of a packet made up of more than one bit or of "n" bytes.
  - The number of bytes in a field is indicated by a numeral enclosed within parentheses: e.g., *name*(4) is a four byte field.
  - The letter "n" is used to indicate the size in bytes of variable-length items: e.g., *name*(n). The form (8*n) designates a length that is a multiple of 8 bytes.

EXAMPLES:

The two examples in Figure 2-1, representing a 64-bit, floating point number, demonstrate a few of the notational conventions. In the first example, the bits are enumerated to expose the most significant bits. In this first example, E=exponent, M=mantissa, `P`=most significant (prime) bit, used to indicate sign in this case. In the second example, the exponent (*exp*) and mantissa are shown as byte fields.

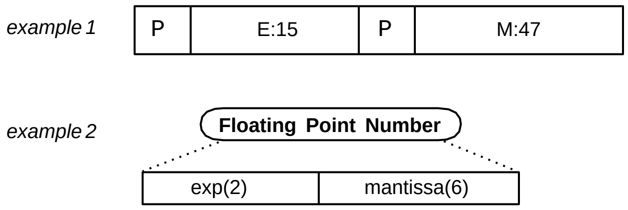

Figure 2-1. Examples of notation.

2.2 Contexts, Associations, Paths, Data Streams, and Packets

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

2.2.1 Context

The term **context** denotes state information kept within one end system. This information represents one instance of an active communication between two (or more) XTP endpoints. A context must be created, or instantiated, before sending or receiving XTP packets. Figure 2-2 illustrates active contexts in two end systems, Host A and Host B. There may be many active contexts at an end system, one for each active conversation or message.

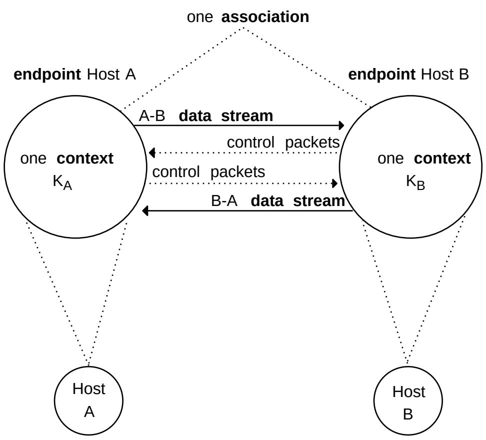

Figure 2-2. Contexts and data streams.

Each context manages both an outgoing data stream and an incoming data stream as well as the potential for both sending and receiving control information. The relationships between context, data stream, and the exchange of control information are illustrated in Figure 2-2. A context may have either a long or a short lifetime, depending on the application. The following examples indicate the range of possibilities:

1. a single unreliable message, where a context exists for the duration of the message transmission without regard for correct delivery or acknowledgement.
2. a reliable message, where a context exists long enough to transmit or retransmit a message until an acknowledgement indicates correct delivery.

3. a handshake, or transaction, where a context is created to send a single arbitrary-length message and is released upon receipt of a single arbitrary-length reply.
4. a conversation where the context exists for as long as the communicating entities wish to remain in contact.

2.2.2 Association

An **association** is the name given to the aggregate of a pair of active contexts and the data streams between them as illustrated in Figure 2-2. In this figure the association is between the two contexts, K_A and K_B.

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

2.2.3 Paths and Forwarding

A heterogeneous internet environment may contain a variety of different network media such as LANs, packet switches, and point-to-point links. Data streams in an association may traverse several different networks. In this case an interconnection device such as a switch or router must **forward** each packet from one network to the next.

XTP models a **path** between end systems as a concatenation of zero or more interconnect devices. Each interconnect device forwards packets to the next until the last device forwards to the destination end system. XTP provides a method for setting up such a path at the time an association is created.

A path is persistent - it is retained for the duration of an association (or longer) unless a network failure forces a reconfiguration. Once created a path may be shared by more than one association. A path is also the focus for the rate control mechanism in XTP in that initial rate allocations are made when a path is first established.

2.2.4 Data Streams and Sequence Space

A **data stream** is an arbitrary length string of sequenced bytes, where each byte is represented by a sequence number. XTP sequence numbers are 32-bit unsigned integer values. Sequence numbers for successive bytes in a stream are obtained by incrementing the sequence number modulo 2^{32}. That is, the successor of 0xffffffff is zero. A sequence number A is defined to be "less than" B if (B-A)<0x80000000.

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

An XTP association provides for two data streams, and consequently two sequence spaces, between a pair of end points as depicted in Figure 2-2. The bytes within one of these data streams are the application data carried by XTP on behalf of a higher-layer application. The sequence number spaces are defined for this user data and do not include control messages or other non-data portions of an XTP packet.

Design Note
Earlier versions of XTP provided both 32- and 64-bit sequence values. The optional 64-bit numbers provide a simple solution to sequence number aliasing on high-bandwidth, high-delay networks where the round trip delay time is greater than the time for sequence numbers to "wrap." However, the 64-bit option was removed from the specification for two reasons. First, 64-bit numbers would double the size of most XTP data structures and fields. Second, the *sync* field in the header can be used to solve the aliasing problem.

2.2.5 XTP Packet

An XTP packet is contained, or encapsulated, within a Medium Access Control (MAC) layer frame, as shown in Figure 2-3. XTP is designed to be independent of physical layer influences, but it does depend on lower layer addresses and expects a cyclic redundancy check from the MAC layer. An encapsulation must be defined for each physical layer environment used with XTP. Section 2.7, "Encapsulation" describes this in detail.

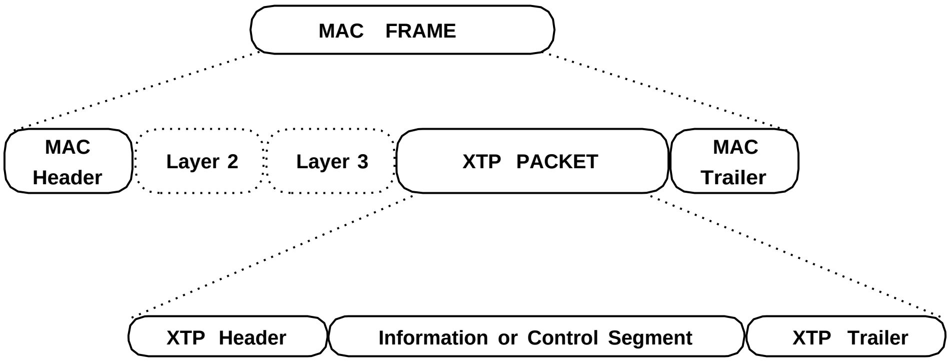

Figure 2-3. XTP packet within MAC layer frame.

An XTP packet consists of an XTP Header, a middle segment, and an XTP Trailer. The middle segment can be a Control Segment or an Information Segment as illustrated in Figure 2-4. For environments with minimum packet-length requirements, the MAC trailer may be padded as needed. The *pformat* field in the XTP Header indicates the format of the middle segment: control segment or information segment.

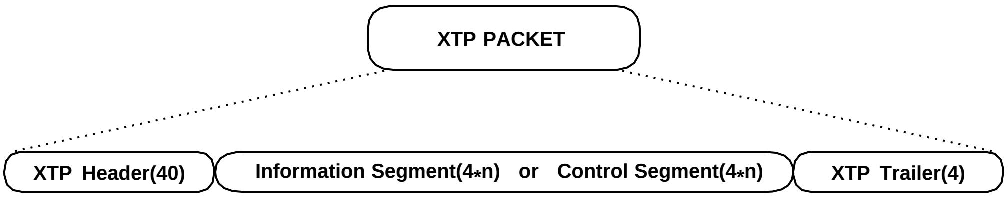

Figure 2-4. XTP packet: 3 major segments.

The syntax for the XTP Header and Trailer is the same for all packets: 40-byte Header and 4-byte Trailer. Control and Information Segments are of variable length. The length for each of these segments—Header, Trailer, and middle segment—is always a multiple of 4 bytes. (See Figure 2-4.) Header, Control Segment, Information Segment, and Trailer are defined in Sections 2.3, 2.4, 2.5, and 2.6, respectively.

Control Segments communicate the state of an XTP machine. XTP packets that contain a Control Segment are called **control packets**.

Information Segments contain data. The data may be generated by a higher-layer application or by a peer transport service entity. XTP packets that contain user (higher-layer application) data are referred to as **data packets**. Information Segments may contain optional, higher-layer information: an optional 8-byte "tag" at the beginning.

2.3 XTP Header

All XTP packets use a fixed header syntax consisting of the following fields: *route*, *ttl* (time to live), *cmd* (command), *sync*, *key*, *seq* (sequence), *dseq* (delivery sequence), *sort*, *dlen* (data length), and *hcheck* (header checksum). The header is illustrated in Figure 2-5 and described in detail in sections 2.3.1 through 2.3.10.

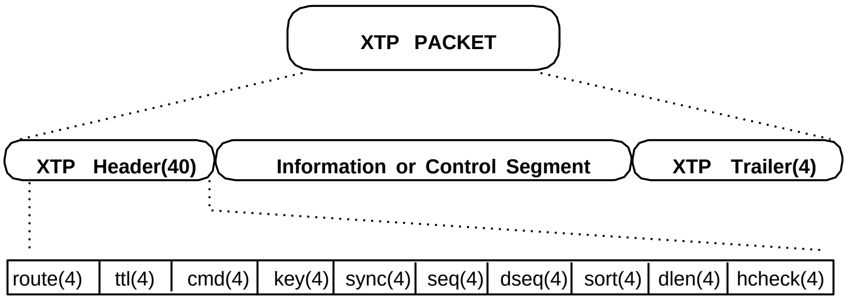

Figure 2-5. XTP Header.

The header fields may be described briefly as follows:

- *route* specifies the next link along a path between switching systems (see section 2.3.1).
- *ttl*, the time-to-live field, specifies the remaining time a packet may exist in an internet environment before being discarded (see section 2.3.2).
- *cmd* specifies the packet type, version number, protocol modes and control functions (see section 2.3.3).
- *key* is the context identifier (see section 2.3.4).
- *sync* is used to synchronize end systems with respect to the *seq* field and with respect to control message exchanges (see section 2.3.5).
- *seq* is an unsigned integer that specifies the byte sequence number of data contained within an information segment (see section 2.3.6).
- *dseq*, like *seq*, is an unsigned integer that specifies the highest sequence number of data delivered to the destination (see section 2.3.7).
- *sort* conveys real-time scheduling information and is interpreted only when the `SORT` bit in the *cmd* field is set (see section 2.3.8).
- *dlen* specifies the length of the information or control segment and is interpreted in every packet (see section 2.3.9).
- *hcheck* is a checksum over the header (see section 2.3.10) excluding *route* and *ttl*.

In summary the header contains information for "steering" a packet through a network or end system (*route*, *key*), bounding packet lifetime (*ttl*), managing sequence number space (*sync*, *seq*, *dseq*), commanding a receiver (*cmd*), scheduling (*sort*), and framing (*dlen*, *hcheck*).

2.3.1 Route Field

The *route* field selects a forwarding destination for an XTP packet when a packet traverses a packet forwarding device such as a switch or router. The forwarding procedure consists of interpreting each incoming route field, using it to select an outgoing path, and replacing the route field and ttl field with new values. Since the route field is replaced at each forwarding system, an XTP *path* (see section 2.2.3) can be thought of as a sequence, or catenation of route values.

The forwarding process is symmetric and bidirectional: pairs of route values - one for each data stream in an association - are assigned at each forwarding system when a path is established. This is analgous to the call setup procedure in a virtual circuit switch, e.g. ATM, where a virtual circuit identifier (VCI) is allocated and assigned to each new virtual circuit. The XTP path setup procedure is defined in sections 3.2.4 and 3.2.5.

The 32-bit route field contains a 31-bit route value and reserves the most significant bit (MSB) as a flag. When the MSB is set the route is called a *return route*. The *route* field is always interpreted and must contain a meaningful value.

The *route* field is subdivided into two fields, *instance* and *index*, in the same manner as the *key* field (see section 2.3.4, "Key Field" and Figure 2-9). The protocol implementation is responsible for selecting the size of the *instance* and *index* fields subject to a time constraint. The constraint for the *route* field is the same as for the *key* field: the minimum wrap time for a *route instance* field must exceed 3600 seconds. all of the considerations for contexts and the *key* field apply to paths and the *route* field (see section 2.3.4, "Key Field").

2.3.2 ttl Field

The 4-byte *ttl* (time to live) field establishes the maximum lifetime of a packet in an internetwork. *Ttl* is set by the packet originator with a value expressed in 100 nanosecond clock ticks. The ttl field is examined and modified as a packet transits an intermediate system such as a switch or router. Different actions are required depending on whether a received ttl value is zero or non-zero.

A ttl value of zero disables the mechanism: the packet never times out. Whenever a packet is received with a ttl value of zero, XTP implementations must forward the packet without altering the ttl value.

For non-zero ttl values, each system that the packet transits must update ttl before forwarding the packet. The update procedure is defined below. It may be replaced by an approximation as suggested in the Implementation Note.

1. timestamp the packet upon arrival.
2. calculate the expected network propagation and latency time for the forwarded packet; decrement ttl by this amount, if ttl becomes zero or negative then discard the packet without forwarding.
3. if the packet is not discarded by step 2, then calculate the output queue holding time just before the packet is transmitted. This is done by subtracting the reception timestamp from the current time. Decrement ttl by the calculated holding time. If ttl becomes zero or negative, discard the packet without forwarding.

> ***Implementation Note***
> A forwarding system can maintain an "expected latency" value for an output queue based on the number of packets in the queue, the total number of bytes in the queue, and the worst case latency onto the destination network. The forwarding system may use an expected latency value for calculating ttl updates instead of explicitly measuring holding times for each packet.

2.3.3 Command (cmd) Field

The 32-bit command (*cmd*) field is interpreted for every packet and must always contain meaningful values for all its fields and bits. Figure 2-6 portrays the syntax of the *cmd* field.

Cmd defines the following information about its associated XTP packet:

1. the *type* of XTP packet (Section 2.3.3.1),
2. the *offset* used to identify the beginning of actual data in an Information Segment (Section 2.3.3.2),
3. the service *options* (Section 2.3.3.3),
4. the XTP version (Section 2.3.3.1)

The first three items are found in the three sub-fields: *ptype*, *offset*, and *options*. The protocol version is indicated by two bits within the *ptype* field.

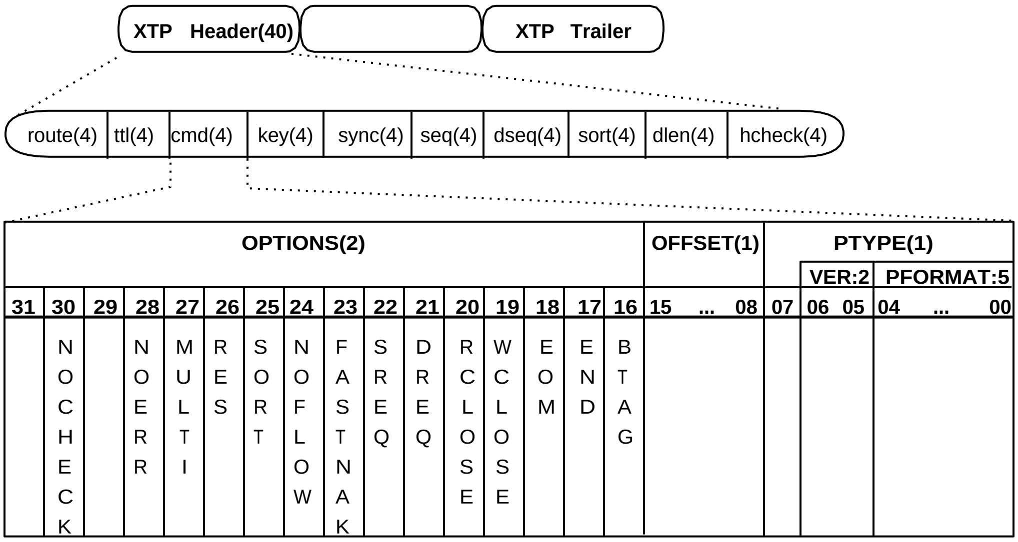

Figure 2-6. Command (cmd) field.

2.3.3.1 Ptype Field

The least-significant byte of the *cmd* field is called the *ptype* field. Within this byte, bits 0 to 4, called the *pformat* field, identify the XTP packet type. Bits 5 and 6 are reserved for indicating the XTP *version*. *Pformat* is always interpreted and must contain a meaningful value. *Version* always interpreted and, in this version of XTP, must be zero. Bit 7 is reserved for future use and must be zero.

The *pformat* values are enumerated in Table 2-1. Values are presented in both decimal and hexadecimal notation.

PFORMAT	*Decimal Value*	*Hex Value*	*Definition*
DATA	0	0x00	user data packet
CNTL	1	0x01	state exchange control packet
FIRST	2	0x02	initial packet of an association
PATH	6	0x06	reconnection packet
DIAG	8	0x08	diagnostic packet
MAINT	10	0x0a	network maintenance information
MGMT	14	0x0e	network management information
ROUTE	18	0x12	route management packet
RCNTL	19	0x13	intermediate system control packet

Table 2-1. Ptypes of XTP packets.

Two *pformat*s, CNTL and RCNTL, each specify a Control Segment. Control Segment syntax is defined in section 2.4. Control packet processing is specified in section 3.3.1.

Both CNTL and RCNTL are referred to as **control packets**, or CNTL packets, because they each contain a control segment. CNTL packets are used to exchange state information between end systems. RCNTL packets are generated by intermediate systems to exert rate control at end systems.

All other *pformats* besides CNTL and RCNTL, namely FIRST, DATA, PATH, DIAG, MAINT, MGMT, and ROUTE, each contain an Information Segment. Section 2.5 defines the information segment syntax for each of these formats. The following paragraphs provide a brief description of each one.

- *DATA* and *FIRST* packets both carry user data. FIRST denotes the initial packet of an association and contains an *address segment* (see section 2.5.2) as well as (optional) user data. DATA packets are used for subsequent data transfers for either data stream of the association. FIRST and DATA packets are defined in sections 2.5.4.1 and 2.5.4.2. Figure 2-7 illustrates the use of a single FIRST packet and the subsequent use of DATA packets between two end systems.
- *PATH* packets (see section 2.5.4.3) PATH packets are used to thread an alternate or additional path through an internetwork, to locate a switch, to recover from failure of an intermediate node, or to join an in-progress multicast conversation. A PATH packet may be multicast to locate switches on a local network.
- *DIAG* packets contain peer-to-peer diagnostic messages (see section 2.5.4.4).
- *MAINT* and *MGMT* packet formats are not defined in this version of XTP.
- *ROUTE* packets exchange information with XTP switches and routers (see section 2.5.4.5).

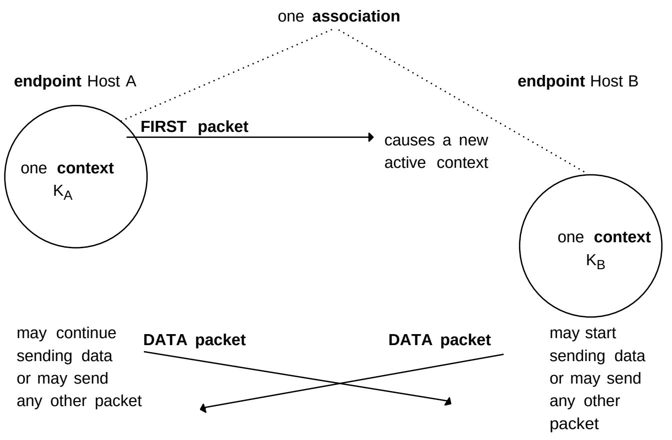

Figure 2-7. FIRST and DATA packets.

2.3.3.2 Offset Field

The single-byte *offset* field specifies the size in bytes of optional padding at the very beginning of the Information Segment. This field is interpreted in all packets.

The offset capability is provided for the convenience of certain transmitters, and simply directs a receiver to discard some leading bytes in the middle segment – either an Information Segment or Control Segment. This capability is useful when a computer output subsystem does not allow byte addressing of memory. In cases where the intended contents of an Information Segment are not aligned on the natural memory boundaries of the I/O subsystem, the *offset* field can specify the needed alignment and eliminate the extra data copy that would otherwise be needed.

2.3.3.3 Command Options

Options field bitflags are listed in Table 2-2.

BIT	*Mask*	*Description*	*Potential Change*	*Expectation*
	0x8000	not used	MBZ	
`NOCHECK`	0x4000	Disable checksum function	Per packet	Per context
	0x2000	not used	MBZ	
`NOERR`	0x1000	Disable error control	Per packet	Per context
`MULTI`	0x0800	Multicast mode	Per association	
`RES`	0x0400	Reservation mode	Per packet	Per context
`SORT`	0x0200	Enable sorting	Per packet	Per context
`NOFLOW`	0x0100	Disable flow control	Per context	
`FASTNAK`	0x0080	Enable aggressive error control	Per packet	Per context
`SREQ`	0x0040	Status requested	Per packet	
`DREQ`	0x0020	Delivery status requested	Per packet	
`RCLOSE`	0x0010	Reader closed	Per packet	
`WCLOSE`	0x0008	Writer closing	Per packet	
`EOM`	0x0004	End of message	Per packet	
`END`	0x0002	End of association	Once	
`BTAG`	0x0001	Beginning data tag	Per packet	

Table 2-2. Header `options` bits.

Bitflags in Table 2-2 select XTP operating modes and mechanisms. The logic convention is positive: a function is enabled if the corresponding bit is set to one, it is disabled if the bit is zero. Zero, one or more bits may be set in each header. The bit definitions are summarized in Table 2-2. Note that there are two spare bits both of which must be zero (MBZ).

An XTP transmitter must control the settings of *options* bitflags in every packet. Some bitflags must have the same value for the lifetime of a context. Others may change from packet to packet. The potential for change for each bit is indicated in the *Potential Change* column of Table 2-2.

Per context means that the bit value must remain the same for all packets sent on a particular context. It also implies that packets received on the context may utilize different bitflag settings. *Per packet* means that the bit value could be different on every packet. *Per association* means that all contexts in an association must set the bit the same way: if one context is in multicast mode, the other context(s) must also be in multicast mode. *Once* means that the bit may be set but once in the lifetime of a context, namely on the last transmitted packet.

Even though some bits could theoretically change on a packet-by-packet basis, it is expected that they would retain the same setting for the duration of the context. Bits expected to remain constant even though they "could" change are indicated in the *Expectation* column of Table 2-2. The protocol definition does not specify a correct behavior if these bits change on a per packet basis. That is, XTP behavior is unspecified and results may be unpredictable.

Design Note

Those *options* bitflags having a potential change of "per packet" but an expected change of "per context" – NOCHECK, NOERR, RES, SORT, NOFLOW, and FASTNAK – could have been specified "per context" only. This would eliminate the *expectation* column in Table 2-2. However, it has been speculated that new applications may be able to utilize the ability to change these *options*.

One can imagine an application using a reliable connection to connect to a server for the purpose of receiving real-time video imagery. After a reliable exchange of messages, the server turns on the NOERR bit while video is being transmitted. There may be an advantage on some networks to providing both modes of operation on the same association rather than requiring two associations - one for a command exchange with the server and another one for the video transmission.

If XTP is quiescent when error and flow control modes are changed, i.e. no packets in transit and no outstanding errors, then the transition to and from these modes can be made safely. If packets are in transit and errors are happening during a change, then packets that were supposedly error-controlled could be lost and other unpredictable behaviors may occur. Some mode changes, e.g. to and from reservation mode, would have no effect on error control and a temporary intermediate period on flow control. Turning checksums on or off could be useful to some software implementations that can trade error coverage for less software overhead at different stages in the lifetime of a context.

Since changing modes is plausible although unlikely without new applications, it seems worthwhile to leave this part of XTP open until there is empirical evidence for either supporting or restricting mode changes.

The following paragraphs define the header bit flags. A bit when set is a one; when clear is a zero.

`NOCHECK` when set indicates that the checksum calculation is disabled for the packet's entire middle segment. When this bit is set, the receiver does not perform a checksum for any of the bits in the Information or Control Segment of the packet, and the *dcheck* field in the Trailer does not necessarily contain a meaningful value. This bit has no effect on the checksum for the Header, which is always enabled.

`NOERR` when set informs the receiver that the sender will not retransmit and directs the receiver to disable error correction processing. The receiver must ignore any input errors. If the sender requests a CNTL packet response by setting the `SREQ` or `DREQ` bits (see below), then the receiver must acknowledge the highest received sequence number. When the sender for a data stream has set `NOERR`, it must ignore (not interpret) *nspan* and *spans* on received CNTL packets. Setting this bit does not prevent the receiver from sending control packets for other reasons. This bit takes precedence over the FASTNAK bit; if both bits are set, error control is turned off.

MULTI when set indicates use of multicast mode. The value of this bit in all packets within an association must be the same value used in the FIRST packet of that association.

RES when set enables reservation mode. By setting this bit to one, the sender indicates to the receiver that the *alloc* values provided by the receiver in its Control Segments must represent actual client buffer space available, not XTP internal buffer space. (The receiver obtains the value for *alloc* from the higher-layer application.) The purpose of reservation mode is to avoid overflowing XTP buffers during bulk transfers.

SORT when set indicates that the value in the *sort* field of the XTP Header should be interpreted and used for sorting/prioritizing the packet. When this bit is zero, the *sort* field is not interpreted and may contain any value. (see section 2.3.8 "Sort Field".)

NOFLOW when set indicates that the sender does not observe *seq*-based flow control. This mode is intended for use on rate-controlled connections or in situations where unconstrained operation is desired. The value of this bit in all packets within an association must be the same value used in the FIRST packet of that association. A receiver may decline to accept NOFLOW mode by rejecting the FIRST packet with a DIAG message (see section 2.5.4.5).

FASTNAK (fast negative acknowledgment) when set enables aggressive error control at the receiving host. FASTNAK may be set by an XTP sender only when the protocol layers below XTP do not reorder packets. XTP switch or packet forwarding systems must clear the FASTNAK bit before forwarding a packet over a system that may introduce reordering. If a receiver detects an out-of-order packet and the FASTNAK bit is set, then the receiver has the option of immediately returning a CNTL packet to the sender that indicates the error. FASTNAK has no effect when NOERR is set. (Refer to "Error Control" Section 3.3.4 for more details.)

SREQ when set requests the receiver to respond immediately with a CNTL packet. That is, an implementation must schedule the CNTL packet before scheduling additional DATA packets for transmission in as expeditious manner as possible consistent with correct operation of the implementation. The SREQ bit is set by an XTP sender according to its output acknowledgement policy or when responses are needed to recover from errors.

The SREQ bit may be set in a FIRST, DATA, PATH, or CNTL packet. A CNTL packet with the SREQ bit set is called a "CNTL-request" packet. A DATA, FIRST, or PATH packet with the SREQ bit set is called a "DATA-request" packet. A CNTL packet sent in response to a CNTL-request packet is called a CNTL-response packet. A CNTL packet sent in response to a DATA-request packet is called a DATA-response packet. Response packets must not set the SREQ bit.

DREQ when set requests the receiver to respond (with a CNTL packet) after it has delivered any already queued data to the higher-layer application.

The WCLOSE and RCLOSE bits are the basis for disconnect handshakes carried out by the close state machines. (Refer to Section 3.2.3, "Context Management: Close" for details on usage.)

EOM is used to delimit record boundaries or message boundaries in a data stream. When set this bit denotes the end of a message. That is, a DATA or FIRST packet containing EOM carries the trailing bytes of a message. EOM is ignored in packets other than DATA or FIRST. The number of bytes or packets in a message or between EOMs is not limited. The EOM bit is not manipulated by XTP, nor does XTP creat or remove EOM bits from a data stream. The bit is asserted by a sending application through its service interface and delivered to the receiving application by the receiving XTP.

END when set indicates that the sending context is being released. It is used in the last message of a closing handshake and also when an association is aborted (see section 3.2.3).

BTAG when set indicates that the first 8 bytes of the *data* segment within the Information Segment contain tag information for the higher-layer application. BTAG is meaningful only in FIRST or DATA packets - it is ignored in all other packets.

2.3.3.4 Byte Order

XTP can be called a "32-bit protocol" because the header and control segment fields are defined as sequences of 32-bit words and because all sequence number fields are 32-bit words. All protocol fields in XTP are to be sent and received in "big-endian" order. These include the header, trailer, control segment, and address segment.

Figure 2-8 summarizes the two byte-order formats, popularly called "big-endian" and "little-endian", that are found in 32-bit systems.

most			*least*	*significance of byte*
0	**1**	**2**	**3**	*Big Endian byte order*
3	**2**	**1**	**0**	*Little Endian byte order*
7 - 0	**7 - 0**	**7 - 0**	**7 - 0**	*bit numbering within each byte*

Figure 2-8. Byte order.

In big-endian format, the least significant byte is at byte address 3; in little-endian format the order is reversed. I/O devices normally transmit and receive data in byte address order on a serial channel. That is, byte 0 is first and followed by 1, 2, and 3.

If the string "abcd" were transmitted to both a big-endian and little-endian machine, both machines would see the letter 'a' first, followed by 'b', 'c', and 'd'. The 'a' would be stored within a word at location '0' as shown in the diagram. Both machines would 'see' the same string, "abcd", as a result when the bytes are accessed in byte addressing order. However, the results are different when interpreted as a 32-bit integer.

Suppose the binary number 0x817d is transmitted to both a big-endian and little-endian machine with the first byte as 0x8, followed by 0x1, 0x7, and 0xd. The big-endian machine would interpret the result as 0x817d, but the little-endian machine would see 0xd718.

Since XTP requires its protocol fields to be transmitted in big-endian order, little-endian systems must perform a byte permutation on these fields after reception and before transmission.

2.3.4 Key Field

The *key* field is used to associate a packet with a context. It is similar to the connection or transaction ID found in most protocols. It is an end-to-end value, generated by an end system and communicated unchanged through switches or intermediate systems to a destination end system. The lifetime of a key value is the same as the lifetime of the associated context: a key is considered "active" while its context is active and "inactive" when its context is inactive or when there is no context.

The *key* field is interpreted and must contain a meaningful value in all packet types except RCNTL and ROUTE, where it is not interpreted and may contain any value.

In packets where *key* is meaningful, a key value of zero is illegal except in a PATH packet where it has special significance. (See Section 3.4.7, "Joining an In-progress Multicast Conversation.")

The 32-bit *key* field, shown in Figure 2-9, contains a 31-bit key value and reserves the most significant bit (MSB) as a flag. The MSB flag directs the interpretation of a key in received packets: if the MSB is zero, then the key identifies a context on the end system sending the packet containing the key; if the MSB is set, then it identifies a context at the destination end system where the packet containing the key is received. A *key* field with its `MSB` set is called a *return key*. (See Section 3.2.2, "Key Exchange" for more details.)

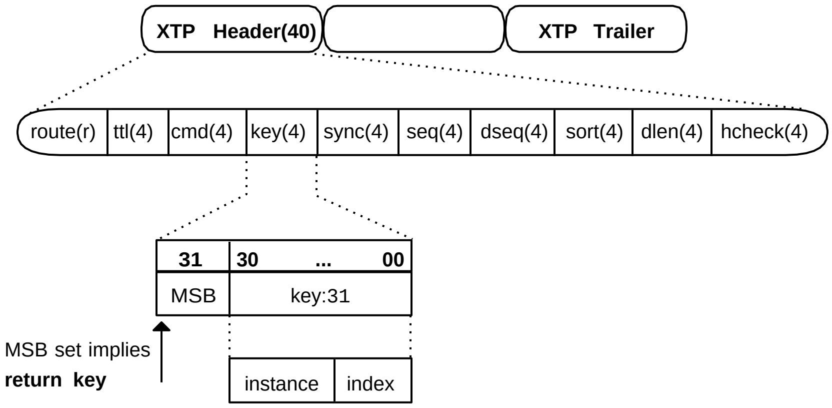

Figure 2-9. Key field.

The *key* field is further divided into an *instance* field and an *index*. The number of bits in each field are determined by the implementation subject to a timing constraint defined later in this section.

The *index* selects the context, presumably by indexing through a context table, although this method of selection is not specified or required by the protocol.

The *instance* field validates *index* values and discriminates against inactive *index* values. The discrimination method depends on associating a current *instance* value with each active context. When a *return key* is received in a packet and the *index* locates the context, the two *instance* values must be compared − one in the *key* and one associated with the context. The *key* is valid only if the two instance values match.

When a *key* is activated along with an associated context, the *instance* value must be incremented so that the new key/context can be distinguished from previous uses of the same *index*. As keys and contexts are reused the instance field will eventually wrap around and reuse instance values. The time required for this to occur is called the *wrap time* for an instance field. In order to avoid key aliasing, the minimum wrap time must be be greater than the maximum time that any end system might keep a context active after a network failure or termination of an association. The maximum holding time for a key is determined by CTIMER (see section 3.5.2), whose maximum value is 1 hour.

Implementation Notes
The minimum wrap time for an instance field is determined by three factors: **I**, the number of bits in the *instance* field, **N**, the number of contexts available in an implementation, and **R**, the rate at which contexts are used by the system and its applications. An expression for minimum wrap time is:

(2I)*N/R**

This expression assumes that inactive contexts are activated in FIFO order. In other words a simple free list which operates in LIFO, or stack, order will not suffice. Inactive contexts must be placed at one end of a queue, and removed from the other end of the queue when activated.

An XTP implementation must ensure that **(2**I)*N/R** >> MAX_CTIMER == 3600 seconds.

Consider an example where an implementation has a 16-bit *index* and 15-bit *instance* and provides a maximum of 1024 contexts. If the context reuse rate is 100 times a second or less, then the expected minimum wrap time is 32768*1024/100, or 335544 seconds which is comfortably greater than 3600.

In this example the wrap time approaches 3600 seconds when the context reuse rate exceeds 9000/sec. The implementor can choose whether to increase the number of *instance* bits or increase the number of available contexts.

If an end system crashes while using a key, it may be possible for the system to reboot and begin using the old key with a fresh context before other end systems completely time out old uses of the key. The only "correct" solution to this problem requires updating active instance values in a non-volatile memory that is not affected by system crashes or power failure. If instance values are initialized from non-volitile storage then the potential aliasing problem is avoided.

In situations with a high-rate of system booting, such as certain embedded system environments, it may be possible to dedicate a small space on a system disk to preserving the state of the XTP key environment across system boots. In other situations a theoretically imperfect but effective technique may be used to greatly reduce the possibility of key aliasing after a crash and reboot. The heuristic consists of "seeding" the initial *instance* values and the starting *seq* numbers (see section 2.3.6) with values taken from a failsafe battery-backed clock which is found on many systems.

Design Note
XTP does not specify an exact assignment of bits to *index* and *instance* to avoid placing limits on the number of contexts supported by an implementation. Consequently CTIMER must have a defined upper bound and the instance field wrap rate must be considered during the implementation process.

In order to remove the bound on CTIMER the protocol design would have to incorporate 64-bit *instance* fields, and would probably also incorporate a 31-bit *index* field, leaving one bit for the MSB *return key* indicator. Thus, keys would be 96 bits and the XTP header size would increase to 32 bytes from 24 bytes.

2.3.5 Sync Field

The *sync* field provides the basis for the *synchronizing handshake* (see section 3.6) and for validating the *seq* field (next section). It is a 32-bit unsigned value. An XTP sender increments *sync* when DATA packets are transmitted. An XTP receiver saves the received *sync* values and returns them to the sender in the *echo* field of CNTL packets. The *sync* rules for a sender are as follows:

1. *sync* in output packets must have constant or increasing values and must not decrease.
2. *sync* is increased only when DATA packets are being sent and remains constant at the most recently transmitted value for all other packet types.
3. The sender must ensure that every DATA packet with the SREQ bit set contains a new value for *sync*. One acceptable procedure for meeting this requirement is to increment *sync* by 1 for every DATA packet. Another procedure is to increment *sync* for only those DATA packets that contain SREQ. Another acceptable procedure is to obtain *sync* values when needed from a free-running clock. The clock frequency should be no greater than the frequency at which packets are sent and

should be no less than the frequency at which the SREQ bit is sent.

4. When SREQ is sent, a transmitter must save the outgoing *sync* value in the local context, e.g. *saved_sync*. Values of *sync* returned as *echo* in control packets can be compared for a match with both *saved_sync* and *sync* as part of the synchronizing procedure.
5. Retransmitted DATA packets must contain newly generated *sync* values, not the ones transmitted earlier.

The *sync* rules for a receiver are as follows:

1. A local context variable, *recv_sync*, is initialized with the value of *sync* received in the first packet received from an end system during an association: i.e. a FIRST, DATA, or CNTL packet.
2. *Sync* values received in subsequent packets are compared with *recv_sync*. If *sync* is "less than" *recv_sync*, it is an old or duplicate packet and is discarded. Otherwise the packet is retained and the received value of *sync* is copied to *recv_sync*.
3. When a packet is received with the SREQ bit set, the received *sync* value is copied to the *echo* field of the CNTL-response packet.

Implementation Note

Sync has provided the basis for a "synchronizing handshake" (see section 3.6) in all versions of XTP. The *sync* procedures stated above have two special properties that can be employed by an implementation: (1) *sync* can be used to avoid sequence number aliasing, (2) *sync* can be used to obtain frequent round trip time estimates.

First, *sync* will consume its 32-bit space at a slower rate than *seq* if implemented as intended. If *sync* is incremented on every DATA packet then *sync* will advance at *1/(average dlen)* times *seq*. That is, the rate depends on the average size of a DATA packet. If *sync* is incremented only when SREQ is set or if *sync* is obtained from a clock, then rate will depend on the implementation. Nevertheless, since the *sync* rate is slower than the *seq* rate, *sync* has the effect of tagging the *seq* field. That is, *sync* provides a base value for the *seq* window within a sequence space that is greater than the 32-bit *seq* space. This property can be used to avoid sequence number aliasing on high-bandwidth, high-latency associations where the *seq* value might wrap faster than a round trip time.

Second, if an implementor chooses to incorporate a time value into the transmitted *sync* field, then the values in the returned *echo* field of CNTL packets will provide frequent updates ("for free") of the round trip time estimate. More frequent round trip time estimates are considered advantageous when compared with less frequent measurements or measurements that are made only when the protocol enters an error state from a timeout.

2.3.6 Seq Field

The *seq* field is an unsigned 32-bit quantity. *Seq* values represent byte sequence numbers associated with one simplex data stream. That is, a sequence number is associated with every data byte. In CNTL, PATH, FIRST, and DATA packets, the *seq* field is always interpreted and must contain a meaningful value. *Seq* is not interpreted for any other packet types and in those packet types may contain any value.

Section 2.2.4, "Data Stream and Sequence Space" provides some background information. Section 3.3.2, "Flow Control", defines the procedures for using the *seq* field.

For FIRST and DATA packets, the value in the *seq* field is the sequence number of the first meaningful byte of the Information Segment. *Seq* values do not include padding bytes as defined by the *offset* field or trailing padding bytes as implied by the *dlen* field. (Section 2.5.1, "Padding" defines padding in detail.) The range of sequence numbers represented by a packet begins with *seq* and extends to *seq+dlen-offset-1*. Thus, a FIRST or DATA packet is said to "consume" a portion of sequence space equal in size to *dlen-*

---

† A 32-bit unsigned number, A, is defined to be "less than" another such number, B, if (B-A)<0x80000000.

offset.

The *seq* field for a FIRST packet contains the sequence number assigned to the first non-padding byte of the *address segment*. The *seq* value is chosen by the implementation, and must be a "random" number (possibly derived from a clock). (Refer to Implementation Notes in section 2.3.4 for explanation.) The value of *seq* for a DATA packet represents the sequence number of the first non-padding byte in the data segment of the packet. (Sections 2.5.2 and 2.5.3 describe the *address segment* and *data segment*.)

For CNTL and PATH packets, the associated *seq* field is the sequence number of the "next" unsent byte of data on the sender's output data stream. The *seq* value of a CNTL or PATH packet does not refer to any byte of the packet itself. Thus, CNTL and PATH packets do not consume sequence space. The value of *seq* in a CNTL or PATH packet is one greater than the highest sequence number ever consumed by a byte of outgoing data for that context; i.e. it is a "high-water mark" of sequence numbers.

Initial byte sequence numbers for both data streams of an association are specified by the *seq* and *dseq* fields in a FIRST packet. Refer to section 2.3.7, "Dseq Field", and section 3.2.1 "Context Management: Open", for further details. Once a sequence space has been initialized, the value for the *seq* field of the "next" output packet is:

[next *seq*] = [*seq* of current packet] + [*dlen* of current packet] - [*offset* of current packet]

EXAMPLE:

In Figure 2-10 a host sends a 300-byte message. (This byte count includes user data and address information.) The bytes are numbered 0 through 299, and transmitted as 3 packets with *seq* values 0, 100, and 200. The sequence number of the last byte of the second packet is 199. The *seq* value in the CNTL packet sent (by the sender) after the second packet is the "next" number, 200, as illustrated. The control packet following the entire 300-byte message uses a *seq* of 300.

If this host were to retransmit the second packet (as shown), that DATA packet would again have a *seq* of 100. However, a CNTL packet following the retransmission would use 300 (the "next" unconsumed data byte).

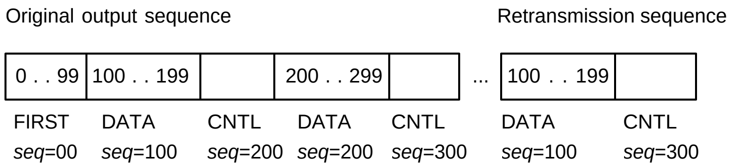

Figure 2-10. Use of seq field.

2.3.7 Dseq Field

Dseq is a 32-bit unsigned quantity. It is interpreted in FIRST, DATA, and CNTL packets. In other packets it is ignored and may have any value.

The *dseq* (delivered sequence number) field acknowledges the sequence numbers of bytes delivered to the remote client application. Output buffers at the sender corresponding to data successfully delivered to the remote client may be released. This can happen for buffers representing sequence numbers up to but not greater than or equal to the returned value for *dseq*.

The value of *dseq* is one plus the sequence number of the last byte actually delivered to the higher-layer client; that is, *dseq* is the next byte that XTP expects to deliver to the client. In NOERR mode, *dseq* must be the same value as *rseq*, even if the receiving XTP has not delivered the data to the application.

In a FIRST packet *dseq* initializes the sequence number for the opposite data stream. The end system receiving a FIRST packet uses the received *dseq* as the initial *seq* value for its own output sequence space.

> ***Implementation Note***
> Any difference between the values of *rseq* (correctly received packets) and *dseq* (packets delivered to the application) represents data in the XTP layer waiting to be transferred to the higher-layer application.

2.3.8 Sort Field

The *sort* field is intended for real-time applications that need to control the scheduling of packet processing among multiple contexts. Interpretation of the *sort* field is optional. XTP implementations are not required to support sorting.

The *sort* field is interpreted only when the `SORT` bit in the Header is set. When the `SORT` bit is zero, *sort* is not interpreted and must contain zero.

The *sort* field is an unsigned 32-bit number that specifies an integer ordering. This is interpreted as a priority ordering where zero is the highest priority, followed by increasing positive integers, each representing a lower value of priority. All ones (0xffffffff) is the lowest priority. On output the *sort* value controls the sorting, or ordering, of output packets to the MAC layer. On input it controls the delivery order (to the higher-layer client) of packets received from the MAC layer.

The XTP subsystem must serve all active contexts that have higher priority (low integer) values before serving contexts with lower priority as long as the active context has data to transmit, or retransmit, and is not blocked from transmitting by the flow or rate control rules. If, when a context becomes active, it has output data of a higher priority *sort* value than the one currently being processed, XTP must switch to the higher priority context. Contexts without *sort* values (i.e., the `SORT` bit is clear) are serviced after contexts with *sort* enabled.

The preemptive priority queue discipline defined above must also be applied on input. That is, an XTP implementation must maintain a separate receive queue for each active context. When packets arrive for multiple contexts, those packets with higher sort values must be processed first, followed by those with lower values, followed by those without SORT enabled. When forwarding XTP packets, those with higher priority are serviced first, followed by those with lower priority, followed by those in which the `SORT` bit is not set.

A conforming XTP implementation must provide means for an application to change the sort value of a context. However, the implementation is responsible for stating maximum time bounds for preemption between contexts for both input processing and output processing.

An implementation must also specify input and output queue behavior when sort values change. There are many possible policies that could be implemented (see notes below). Only one policy, FIFO queue policy, is required. All others are implementation options.

Implementation Notes
In a **FIFO** queue policy, data is appended to an input or output queue as it arrives. Sort values, if present, accompany the data through the queue.

In a **priority** queue policy, the single FIFO queue is maintained as an abstraction for external use. Internally the single FIFO queue is replaced by a (dynamic) set of FIFO queues - one for each Sort value that has been appended to the aggregate. When new data is appended to the priority queue it is actually appended to one of the internal queues, or causes a new internal queue to be created. As data is read from the other end, it comes from the internal queue with the highest sort value.

In a **destructive** queue policy, lower priority data is discarded from a queue when higher priority data is appended.

2.3.9 Dlen Field

The *dlen* field specifies the number of bytes in an information or control segment. This includes any or all of the following: *offset* padding, *address segment*, *data segment*, any *code, val, or message* fields, or a control segment. Pad bytes must be appended to an information segment so as to align the trailer on a multiple-of-four byte boundary. Trailing pad bytes are not included in the *dlen* count. The length of the middle segment may exceed *dlen* and is determined by rounding *dlen* up to the nearest multiple of four. This can be expressed in C code as **`(dlen+3)&^3`**.

2.3.10 Hcheck Field

The *Hcheck* field contains the result of the XTP check function (defined in Appendix A) over the preceeding 7 words of the header. This includes CMD, KEY, SYNC, SEQ, DSEQ, SORT, and DLEN, and omits ROUTE and TTL.

2.4 Control Segment

A Control Segment represents the state of a context. XTP packets containing a Control Segment as their middle segment are referred to as **control packets**. Control packets are used to exchange state information between endpoints. Control Segments are of variable length, however their syntax guarantees that the length will be a multiple of 8 bytes.

The Control Segment is included in XTP packets of the following types: *CNTL* and *RCNTL*.

Control packets are not numbered consecutively as are data packets (i.e., they do not consume sequence numbers and they are not error controlled), but their *seq* fields are meaningful. A receiver cannot notice when a control packet has not been received and cannot request a retransmission. (Refer to Sections 2.3.6, "Seq field" and 2.2.4, "Data Streams and Sequence Space" for more details about sequence numbering.)

Several conditions cause control packets to be generated. (Section 3.3.1, "Control Packet Processing" describes control packet processing.)

The format for a Control Segment is shown in Figure 2-11. Interpretation of Control Segment fields is described below.

- When the packet is RCNTL, only *rate*, *burst*, *xroute*, and *dcheck* within the Control Segment are interpreted; and only *cmd*, *sort*, *route*, and *hcheck* within the header are interpreted. The remaining fields in the control segment and header are not interpreted and may contain any value.
- When the packet is CNTL, the Control Segment fields are interpreted as summarized below:
  - *rate*, *burst*, *sync*, *time*, *rseq* and *alloc* always are interpreted and must contain meaningful values that represent the sender’s current status.
  - *echo* and *techo* are interpreted in all received CNTL packets for which *techo* is non-zero.
  - *xkey* and *xroute* are interpreted in all CNTL packets when they are non-zero except when the `MULTI` bit is set; the values must be set by the sender as specified in Section 3.2.2, "Key Exchange," and 3.2.5, "Route Exchange."
  - *rsvd* is not interpreted for this version of XTP but must be zero.
  - *nspan* is interpreted and must contain a meaningful value whenever the `NOERR` bit in the *cmd* field of the header is zero. When `NOERR` is set to one, *nspan* is not interpreted and may contain any value.
  - *spans* is interpreted when *nspan* is greater than zero and the `NOERR` bit is clear. When *nspan* is zero or `NOERR` is set, the *spans* field is not interpreted and may contain any value and may be of any length that is a multiple of 8 bytes. When interpreted, *spans* must contain the number of pairs indicated by *nspan*.

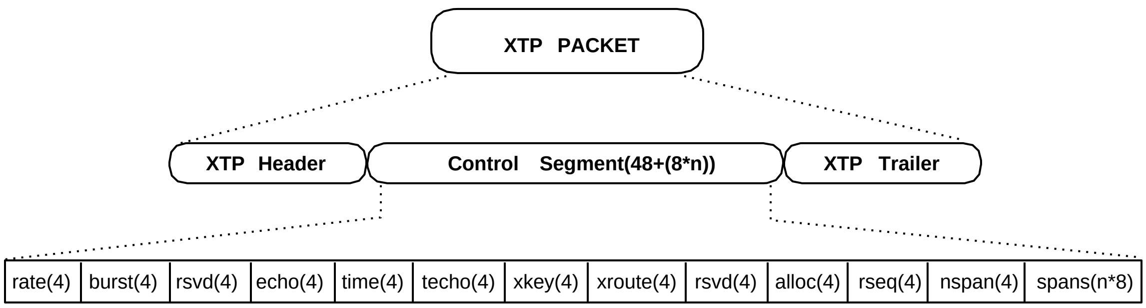

Figure 2-11. Control Segment.

The received values for *rate* and *burst* control the transmission characteristics of the sending machine. These fields must contain meaningful values at all times. Values for *rate* and *burst* refer to the contents of information segments and exclude MAC headers, XTP headers, trailers, and control packets. *Rate* sets the maximum transmission rate in bytes per second. *Burst* sets the maximum number of bytes in one burst of transmission. A *rate* value of 0 (all zeros) completely halts all transmission. A *burst* value of 0 (all zeros) completely halts all transmission. A *rate* value of 0xffffffff (all ones) specifies an unconstrained rate: the sender may transmit at will and is not constrained by *burst*. A *burst* value of 0xffffffff (all ones) specifies an unconstrained burst: the sender may transmit bursts of any size. (Section 3.3.3, "Rate Control" contains more detail.)

A CNTL packet with the SREQ bit set is called a "CNTL-request" packet. A DATA, FIRST, or PATH packet with the SREQ bit set is called a "DATA-request" packet. A CNTL packet sent in response to a CNTL-request packet is called a CNTL-response packet. A CNTL packet sent in response to a DATA-request packet is called a DATA-response packet. The *sync* field received in any request packet must be copied to the *echo* field of the corresponding response packet. The procedures for sending and interpreting *echo* and *sync* are specified in section 2.3.5, "Sync Field", and in section 3.6, "Synchronizing Handshake".

The *time* and *techo* fields are interpreted only in CNTL packets. The *time* and *techo* fields are used for determining round trip time (**rtt**). (See Section 3.5, "Timers" for more details.)

The *time* field received in a CNTL-request packet must be copied to the *techo* field of the corresponding CNTL-response packet. The *time* field in a CNTL-request packet must always contain the packet sender's local time. The resolution of this time value is determined by the implementation and not defined by XTP. The *time* field in CNTL-response packets may contain any value because it will not be returned in a response packet.

The *techo* field in a CNTL-response packet is the returned *time* value. The *techo* field in a DATA-response packet must be zero. Thus, the *techo* field is used to discriminate between DATA-response packets (*techo* is zero) and CNTL-response packets (*techo* is non-zero).

Round trip time can be calculated whenever a CNTL-response packet is received by subtracting the received *techo* value from the current time value. The difference is the elapsed time, or round trip time.

Design Note
If a CNTL-request packet contains a *time* value of zero, it will be returned as a *techo* value of zero. In this one case a packet that is really a CNTL-response packet will be mistaken for a DATA-response packet. The effect of this is that on rare occasions an otherwise valid time sample will be lost. This is considered to be a low-probability event that has no effect on the operation of the protocol.

The *xroute* field is interpreted in all control packets except when the `MULTI` bit is set, and must contain a meaningful value. For packets with `MULTI` set, the *xroute* field is not interpreted and may contain any value. *Xroute* must be zero when either of the following is true: (1) a successful *route* exchange has occurred, or (2) the control packet sender does not wish to do a *route* exchange. Non-zero values in *xroute* must already be primed (i.e., the most significant bit must be set). The *xroute* field is provided for the *route* exchange mechanism described in Section 3.2.5, "Route Exchange."

The *xkey* field is interpreted in all CNTL packets except when the `MULTI` bit is set. For all RCNTL packets and for CNTL packets with `MULTI` set, the *xkey* field is not interpreted and may contain any value. *Xkey* must be zero when either of the following is true: (1) a successful *key* exchange has occurred, or (2) the control packet sender does not wish to do a *key* exchange. Non-zero values in *xkey* must already be primed (i.e., the most significant bit must be set). The *xkey* field is provided for the *key* exchange mechanism described in Section 3.2.2, "Key Exchange."

> ***Implementation Note***
> It is permissible for an implementation not to perform a *key* exchange. This will not violate the protocol.

Alloc (allocation) is interpreted when flow control is enabled and must contain a meaningful value in CNTL packets. The value of the *alloc* field is a do-not-exceed sequence number that limits the amount of data a sender may transmit; a sender may send data with byte numbers up to, but not including, the sequence number in *alloc*. *Alloc* represents the internal buffer space of the receiving XTP machine, except when the `RES` bit is set, in which case it represents client (higher-layer application) buffer space.

Sequence-based flow control is disabled by NOFLOW mode. The *alloc* field of received control packets is ignored by a sender that is in NOFLOW mode (see section 2.3.3.3 for a description of the NOFLOW bit).

> ***Implementation Note***
> The *alloc* values provided in NOERR mode must be designed carefully. If they are based on *dseq*, the receiver's buffer space can overflow since *dseq* in NOERR mode does not necessarily mean that the data has been passed to the higher-layer application.

Rseq (received sequence number) is interpreted and must contain a meaningful value in all CNTL packets. In CNTL packets, *rseq* indicates the *seq* number expected on the next incoming packet. In situations where data has been received, the value of *rseq* is one past the highest consecutive sequence number received without error at an XTP host. For a context in which no data has as yet been received, *rseq* is the starting sequence number for the data stream (i.e., *dseq* from last received packet). Whenever error processing has been turned off (the last received packet had its `NOERR` bit set), the value of *rseq* must be one past the highest sequence number ever received on the data stream.

EXAMPLE:

> Suppose a series of 100-byte sized data packets with *seq* fields of 00 through 700 are sent through the network to a receiver that sees packets with *seq* fields 00, 100, 200, 300, 600, and 700. Packets 400 and 500 are not received. Figure 2-12 illustrates a data stream that includes this example. Any control packet generated by the receiving unit would have the number 400 in the *rseq* field. This means that bytes up to and including 399 were received in order.

Nspan (number of sequence number pairs in the *spans* array) is inspected in every CNTL packet. When *nspan* is non-zero, the *spans* fields are interpreted and must contain meaningful values. When *nspan* is zero, *spans* may contain any value and may be of any multiple of 8-bytes length. In this case (*nspan* is zero) the size of the *spans* field is not necessarily zero. Its length is equal to (dlen-offset-48) bytes.

Nspan and *spans* together provide for selective acknowledgement and retransmission of packets over arbitrary sequence number ranges. The *nspan* field states the number of pairs within *spans*. The *spans* field consists of pairs of sequence numbers defining unbroken sequences (spans) of received data within a data stream. The number of pairs must be the number indicated in *nspan*.

Since each pair occupies 8 bytes, the *spans* field is always a multiple of 8 bytes.

The pairs of sequence numbers within *spans* indicate ranges of received packets that occur after *rseq*. (See Figure 2-12.) Identifying the ranges of received bytes makes it possible for the data stream sender to calculate the gaps that exist at the receiving host. Each pair within *spans* (a, b) describes an intact range (one span) of received sequence numbers that is bordered on one or both sides by missing data (i.e., a gap). The first number in a pair is the lowest sequence number for the span; the second number is one past the highest sequence number for that span. The first *spans* pair must describe the span with the lowest sequence numbers. The pairs themselves must be in order so that they describe sequentially ordered spans. Figure 2-12 and the accompanying example describe this in more detail.

When *nspan* is non-zero, indicating the presence of valid *spans*, an XTP receiver is allowed to reduce the value of *nspan* and process fewer *spans* for implementation reasons. Furthermore, the receiver may truncate the field to zero and implement go-back-n error control in which retransmission is based on the value of the *rseq* field.

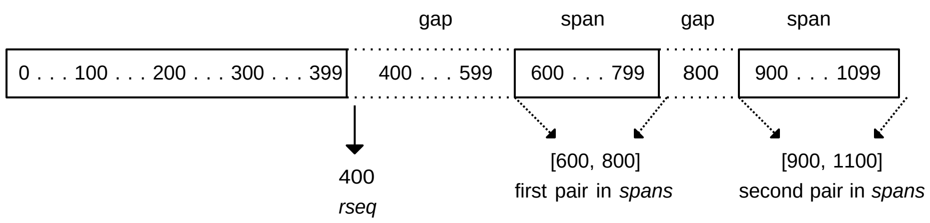

Figure 2-12. Use of spans field.

EXAMPLE:

Figure 2-12 illustrates the construction of *spans* for a data stream consisting of eleven 100-byte packets with *seq* fields of 00, 100, 200, etc. The receiver sees packets with *seq* values 0, 100, 200, 300, 600, 700, 900, 1000. Since bytes 00 to 399 are a contiguous sequence, the receiver returns *rseq* of 400. The receiver then describes all other intact spans in the *spans* field. Two such ranges exist, so it would return two pairs within its *spans* array: (600,800), followed by (900,1100). This combination of *rseq* and *spans* makes it possible for the data stream's sender to calculate that sequence numbers 400-599 and 800-899 should be retransmitted. In this example, the value for *nspan* would be 2.

2.5 Information Segment

The Information Segment contains data. Information Segments are used in XTP packets of the following types: FIRST, DATA, PATH, DIAG, MAINT, MGMT, and ROUTE. The first two types can contain higher-layer (user) data. These are referred to as data packets. The others contain transport layer messages only. Each type has its own format as discussed in separate sub-sections below. (The type of an XTP packet is indicated by the *type* field within *cmd* of the XTP Header.)

Information Segments of FIRST and DATA packets always consume sequence space; those of DIAG, MAINT, MGMT, PATH, and ROUTE packets do not. "Null" data packets can be sent using the mechanism described in Section 2.5.3.2 "Zero-length data segments."

Information Segments can contain optional padding at the beginning and/or at the end of the Segment as shown in Figure 2-13. (Refer to Section 2.5.1 "Padding" for more details.)

Information Segments are of variable length. However, the total byte count for an Information Segment must be a multiple of 4 bytes. The *align* padding at the end of an Information Segment is used to meet this requirement. (See the discussion on *align* padding in Section 2.5.1.)

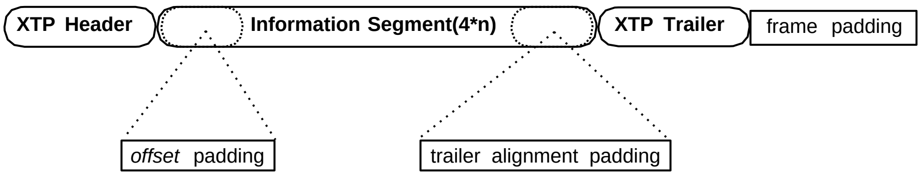

Figure 2-13. Optional padding for any Information Segment.

2.5.1 Padding

Any Information Segment or Control Segment may contain padding at the beginning and/or the end of the Segment. An XTP packet may include additional padding at the end (after *Dcheck*). The purposes for padding include aligning the beginning of the middle segment, making the segment a multiple of 4 bytes in length, and meeting the minimum packet requirements for certain physical layers.

Padding bytes at both the beginning and end of the Information Segment are treated equally with regards to sequence numbering and checksums:

1. they are **excluded** from the sequence space so that error control is well-defined in sequence space;
2. they are **included** in the *Dcheck* checksum so that the checksum implementation operates only on 32-bit words.

Offset Padding

Padding at the beginning of an Information Segment or Control Segment is placed between the final byte of the XTP Header and the first byte of the Information Segment, as shown in Figure 2-14. The *offset* field of the XTP Header indicates the number of bytes of padding in this location. (Refer to Section 2.3.3.2, "Offset Field" for more details.)

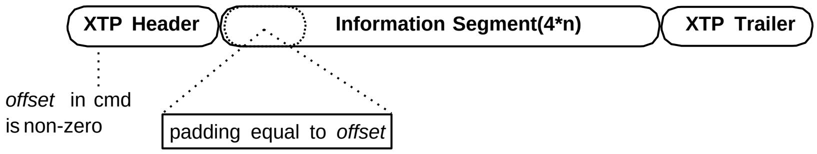

Figure 2-14. Offset padding.

Trailer Alignment Padding

Up to three bytes of padding may be required at the end of the information segment to locate the trailer on a multiple-of-four byte boundary. When needed it is placed before the first byte of the XTP Trailer, as shown in Figure 2-15. The length of the middle segment is determined by rounding *dlen* up to the nearest multiple-of-four. Note that *dlen* includes any *offset* bytes contained in the Information Segment. If *offset* is non-zero, the number of data bytes in the Information Segment is *(dlen-offset)*.

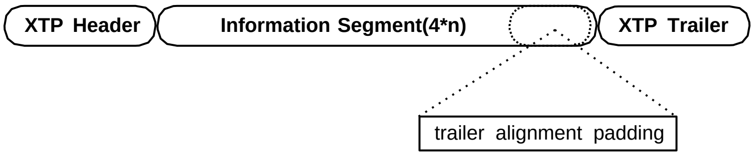

Figure 2-15. Align padding.

XTP Frame Padding

The smallest XTP packet contains a MAC header plus the 24-byte XTP header, 4-byte information segment, and 4-byte trailing *Dcheck* field for a total of 32 bytes, not counting the MAC header. For media that have a minimum packet size constraint, arbitrary padding bytes can be added to the XTP packet after the *dcheck* field.

2.5.2 Address Segment

Two packet types require an *address segment* within their Information Segment: FIRST and PATH. The *address segment* contains destination and sender addressing information. Its design provides flexible address support.

An *address segment* of a FIRST packet consumes sequence space, but that of a PATH packet does not.

The *address segment* is of variable length but must be a multiple of 4 bytes. It is included within the Information Segment's checksum (*dcheck*). The intent of the *address segment* is twofold:

1) to enable a sender to locate a destination machine and select a destination service, and
2) to establish a return path for data and control packets.

An *address segment* consists of a 24-byte *descriptor* followed by a variable-length network *address* as shown in Figure 2-16.

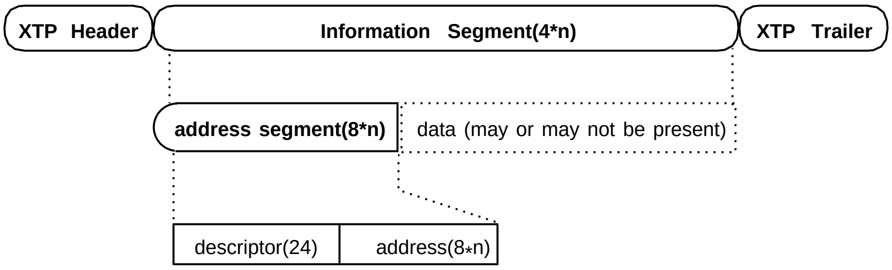

Figure 2-16. `Address segment`.

The 24-byte descriptor consists of *control* field which contains subfields *alen*, *service*, and *aformat*, plus *rate_req*, *burst_req*, *maxdata*, and *id* fields (Refer to Figure 2-17.) The *descriptor* is always interpreted and must contain meaningful values.

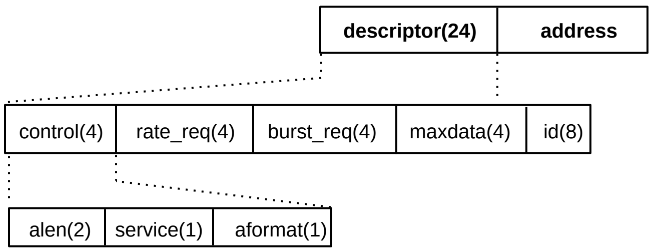

Figure 2-17. `Descriptor` field.

The 2-byte *alen* field counts all the bytes in the *address segment* field including the *alen* field itself. *Address segment* length is always a multiple of 8.

The 1-byte *service* field is used to indicate the kind of traffic expected on the association. This information is provided by the application and passed along in the FIRST packet to the destination cpu. The *service* byte is transmitted without change to the receiving context which uses the bits to select an interface for the duration of the association. The defined values for *service* are enumerated in Table 2-3.

decimal	hex	service
00	0x00	Connection
01	0x01	Transaction
02	0x02	Unacknowledged datagram
03	0x03	Acknowledged datagram
04	0x04	Isochronous stream
05	0x05	Bulk data

Table 2-3. Service type values.

The definitions for *connection*, *transaction*, *datagram*, *isochronous*, and *bulk* are the common ones given in the Glossary (section 1.2).

The *service* byte may also be inspected by an XTP switch or internet gateway as a guideline for internal resource allocation. In this capacity the value 0x04 for Isochronous Stream is distinguished because it represents a requirement, not just a guideline (see discussion of *rate* below).

The 1-byte *aformat* field identifies the network address syntax according to Table 2-4. Addresses that are listed in the table, but not detailed explicitly in the text, are place holders. Values not explicitly defined must not be used unless assigned by the address assignment authority, which, for this version of the protocol, is Protocol Engines, Incorporated.

Aformat value		*Address syntax*
Decimal	*Hex*	
0	0	No address
1	0x01	Internet Protocol Address
2	0x02	ISO Standard
3	0x03	XNS
4	0x04	IEEE 802-style Source Route
5	0x05	MODSIM
6	0x06	MicroSoft NetBios
7	0x07	IP-style Loose Source Route
8	0x08	IP-style Strict Source Route
9	0x09	XTP Direct (locally-defined) Address
10	0x0a	XTP Experimental Address Type
11	0x0b	USAF Embedded System Address

Table 2-4. Aformat values.

The 4-byte *rate_req* field contains a positive integer representing the sender's initial, desired data flow rate in bytes per second. This data rate is interpreted as an average data rate where the peak rate can be higher. A value of zero means that the sender is not making a rate request. NOTE: if the *service* field contains the value 0x4 indicating isochronous stream service, the *rate_req* field is interpreted to be a continuous data rate **required** for the lifetime of the association. If the end node or any intermediate system cannot support the requested rate, then the setup request must fail with an "insufficient bandwidth" DIAG message (Code 7, Val 2).

The 4-byte *burst_req* field contains a positive integer indicating the expected output burst size in bytes. A value of zero indicates that no burst size is being declared.

The 4-byte *maxdata* field indicates the length in bytes of the maximum information segment that the sender expects to transmit during the lifetime of the context.

The 8-byte *id* field contains the MAC address of the originator of the FIRST or PATH packet. The MAC address must occupy the low-order bits of the field. The most significant, unused bits must be zero. (The MAC address is right justified within the *id* field.)

The *address* field of the *address segment* specifies the two endpoints, (source and destination), for the FIRST or PATH packet. The contents of an *address segment* normally does not change as the packet transits a switch. However, source routing formats require some updates to be made to the address field as the packet carrying it progresses through the list of destinations. For these address formats the *dcheck* field, unless disabled by NOCHECK, must be updated after the address segment is updated. Address field formats are specified in the following paragraphs.

No Address Format (0)

The "no address" format is used when an *address segment* does not contain an address. This never happens on a network, but it could happen on a dedicated point-to-point link. When this format is indicated, the *address* field is not present. The *address segment* consists of the *descriptor* field only.

Internet Address Format (0x01)

The Internet format is as shown in Figure 2-18.

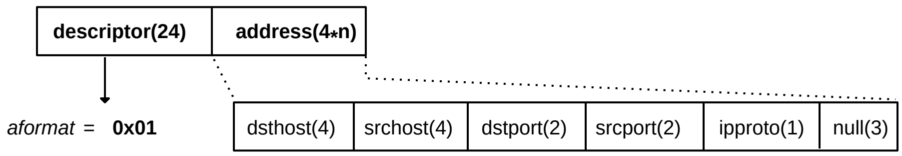

Figure 2-18. `Address` field for Internet addresses.

In this format *dsthost* and *srchost* are each 32-bit IP addresses as defined by RFC 791. The escape code for extended addressing mode, undefined in IP, is explicitly disallowed in XTP. If extended addressing mode becomes defined in IP, a new XTP format will be assigned. The 2-byte *dstport* and *srcport* fields are socket numbers. The 1-byte *ipproto* field is the protocol type field from IP. The 3-byte *null* field must be zero.

ISO Address Format (0x02)

The ISO format for each endpoint’s address is an NSAP-address and T-selector value. The ISO format is as shown in Figure 2-19.

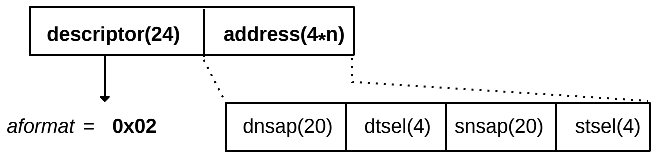

Figure 2-19. `Address` field for ISO addresses.

In this format, destination and originator NSAP (network service access point) addresses are as defined in ISO 8348 addendum 2, where the maximum length for a binary NSAP address is 20 octets. When an NSAP is less than twenty bytes (octets), the NSAP must occupy the most significant bytes of the *dnsap* and *snsap* fields with zeros for padding; *dnsap* and *snsap* are left justified. The T-selector (transport layer selector) fields for both destination (dtsel) and originator (stsel) are 4-bytes each.

XNS Address Format

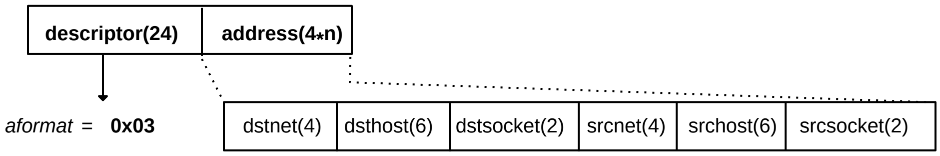

Figure 2-20. `Address` field for XNS addresses.

The Xerox Network System (XNS) format is represented in an XTP address segment as shown in Figure 2-20.

IEEE 802 Source Route Address Format

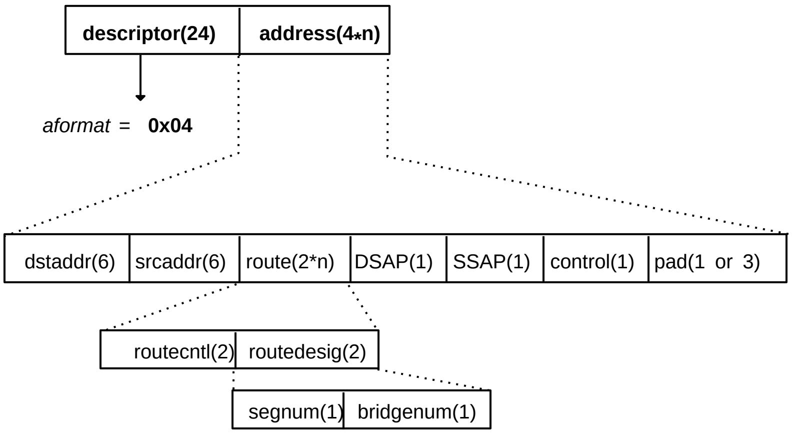

Figure 2-21. `Address` for IEEE 802 Source Route addresses.

This format allows XTP to use bridge-level source routing as shown in the figure. Consult IEEE 802 address specifications for a detailed description of the fields.

MODSIM Address Format
Code 0x05 is reserved for the U.S. Air Force Modular Simulator project.

MicroSoft NetBios
Code 0x06 is reserved for MicroSoft NetBios address format.

IP-style Loose Source Route
IP-style Strict Source Route

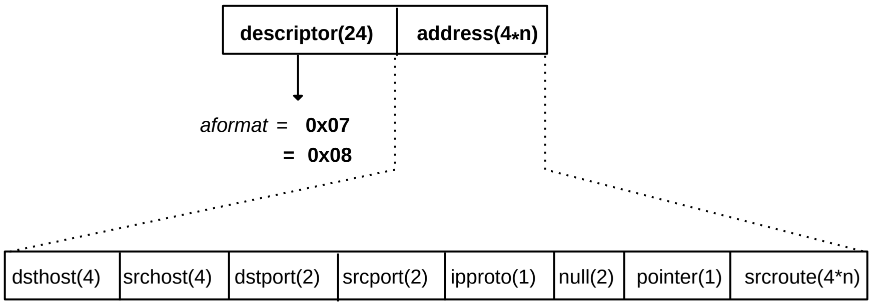

Figure 2-22. `Address` **for IP-Style Source Route addresses.**

Internet Loose and Strict Source Routes are both represented in an XTP address segment as shown above. The format consists of the normal IP address followed by a pointer that indexes a list of intermediate IP addresses. Interpretation of these formats is as specified in RFC 791.

Direct Address

Code 0x09 is allocated for local usage (e.g., direct addressing). XTP does not define the *address* field when this format is used. Direct addressing is useful for systems having a known, fixed communication topology, or other applications where a Permanent Virtual Circuit facility is required. Avionics and other real-time systems are the primary candidates for direct addressing. Note that Direct Address Format packets and other XTP address format packets can coexist on the same network.

XTP Experimental Address Type

Code 0x0a is reserved for implementor experimentation.

USAF Embedded System Address

Code 0x0b is reserved for the U.S. Air Force Embedded System Address format.

2.5.3 Data Segment

The *data segment* is a variable-length field. Padding may be required after the data segment to pad the Information Segment to a multiple of 4 bytes as discussed in Section 2.5.1 "Padding." Data segments must consume sequence space so that error control, which is based on sequence numbers, can be applied. Section 2.5.3.2 describes the method for sending a zero-length segment.

2.5.3.1 Normal Data Segments

Data segments are intended for transmission of higher-layer application (user) data. Two packet types can include data segments: FIRST and DATA.

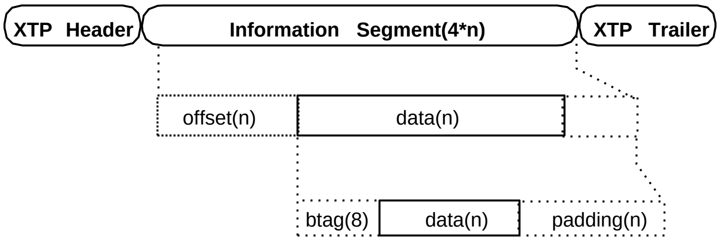

Figure 2-23. `Data segment`**.**

The first eight bytes of a data segment may be marked or "tagged" for special use by higher-layer applications as shown in Figure 2-23. XTP provides a bit in the XTP Header (`BTAG`) to indicate the presence of the *btag* field. Tag fields are opaque to XTP, meaning that XTP "sees" them but does not look inside nor interpret them. XTP transports the tag fields as uninterpreted user data bytes. The *btag* field is intended to support higher-layer applications such as encapsulation and convergence protocols.

Application Note
`BTAG` provides a means for applications to mix control information with data while avoiding the imposition of another layer of framing within the *data* segment. Some examples follow.

> An application wishing to send a mix of audio, video, and text frames would send a *btag* field indicating when the frame type changed. The receiving application code need only watch for the `BTAG` bit to know when to process the *btag* field and the corresponding frame type change. `BTAG` would be used in conjunction with `NOERR` to achieve frame synchronization without retransmission.
>
> An application wishing to send a group of files could mark the end of one file and the beginning of another with a *btag* field. The receiving application would know when to process a file change or when to move data without additional scanning of the data stream or additional handshakes between applications.
>
> An application may wish to encapsulate and forward frames within XTP that arise from a different protocol. An example would be multiplexing transactions from one host to a server on a single XTP association. The *btag* field would direct the decoding of the multiplexed data stream at a receiver.

2.5.3.2 Zero-length Data Segments

A zero-length or null data segment is encoded as a *btag* field containing a special code. (The *btag* field is described in Section 2.5.3.1.) A zero-length data segment consists of a *btag* field only, so it is actually 8 bytes long. This mechanism enables transmission of DATA packets that consume sequence space (for error control purposes), but do not contain any data. It is strongly recommended (although not absolutely required) that the code "all ones" be used in this solitary *btag* field. However, only higher-layer applications interpret this code and interoperability among all higher-layer applications is not within the scope of this document.

2.5.4 Data Segment Formats

This section describes the format for each packet type that utilizes an Information Segment: FIRST, DATA, PATH, DIAG, and ROUTE. MAINT and MGMT are not defined in this version of XTP.

2.5.4.1 FIRST Packet Format

When the *pformat* field of the XTP Header is FIRST, the Information Segment consists of two sub-segments: a mandatory *address segment* and an optional data segment, as illustrated in Figure 2-24. Delivery of the addressing information to the higher-layer application is an implementation service option. The data segment is optional in a FIRST packet; it is never necessary to utilize the mechanism described in Section 2.5.3.2 for zero-length *data* segments. (Section 2.5.2 details the syntax for the *address segment* and Section 2.5.3 describes the *data segment*.) Both the *address segment* and the *data segment* (if present) consume sequence space in FIRST packets.

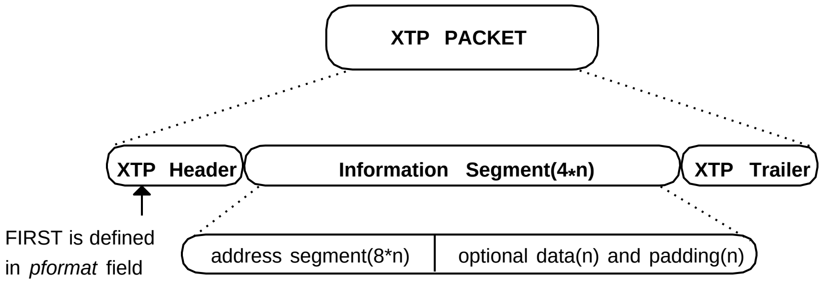

Figure 2-24. FIRST packet.

2.5.4.2 DATA Packet Format

The Information Segment of a DATA packet consists of a mandatory, variable-length data segment. (Section 2.5.3, "Data Segment" describes the syntax.) The design of the DATA packet eliminates the overhead of having an address in every packet. XTP neither examines nor modifies the contents of data segments. *Data* segments may contain tagged information as illustrated in Figure 2-23. The data segment of a DATA packet must consume sequence space. (Section 2.5.3.2 defines the representation for a zero-length *data* segment.)

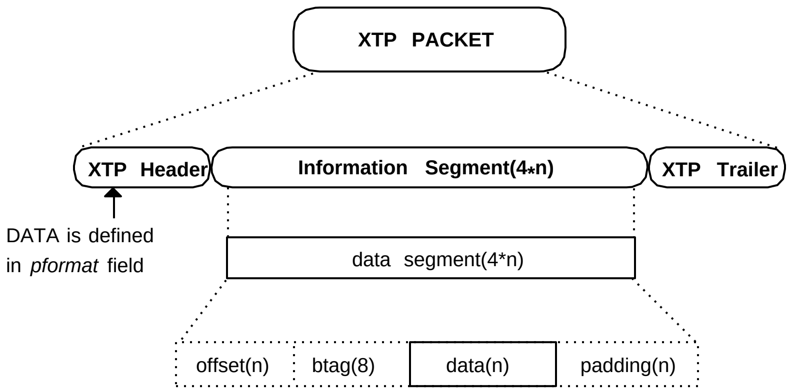

Figure 2-25. DATA packet.

2.5.4.3 PATH Packet Format

PATH packets contain an Information Segment consisting of an *address segment* only. (Refer to Figure 2-26.) A PATH packet consists of an XTP Header, an *address segment*, and an XTP Trailer. (Section 2.5.2 details the syntax of the *address segment*.) No user (higher-layer) data can be included with this format. PATH packets do not consume sequence space.

PATH packets are used in situations where an association already exists, but a change is necessary. Examples include the following: to thread a path through an internetwork, to locate a switch, to recover from intermediate node failure. to locate switches on a local network (using multicasting), or to join an in-progress multicast conversation. (Section 3.4.7, "Joining an In-progress Multicast Conversation" details one use for this packet.)

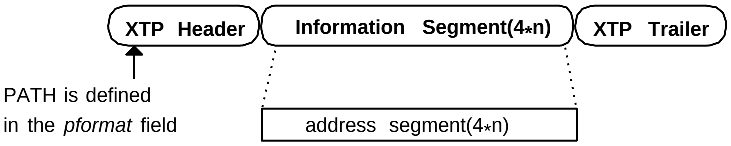

Figure 2-26. PATH middle segment.

2.5.4.4 DIAG Packet Format

DIAG packets report pathological conditions that are either fatal or which require corrective action. DIAG codes must be generated by an XTP implementation (endpoint or intermediate node) in response to an incoming packet when the packet cannot be processed. An XTP implementation may (as a service) pass the diagnostic information to the higher-layer application. (Sections 3.2.1, 3.2.3, 3.2.4, and 3.2.7 describe some uses for DIAG packets.)

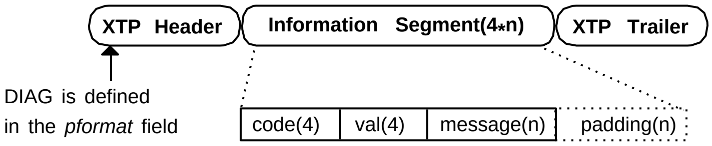

Figure 2-27. DIAG middle segment.

A DIAG packet contains an Information Segment consisting of the following three fields: *code*, *val*ue, and *message*. (See Figure 2-27.) *Code* and *Val* are 32-bit integers, and the *message* part is a variable length ascii string. *Code* and *Val* are required and use the values defined in Table 2-5. The *message* string is optional and may be defined by the implementation. No user (higher-layer) data can be included with this format. DIAG packets do not consume sequence space.

Implementation Notes

Be cautious about allowing DIAG packets to be generated in response to other DIAG packets.

Be aware of the consequences of generating DIAG packets within multicast contexts (i.e., in response to packets with the MULTI bit set).

In both cases, an implementation should avoid getting into an infinitely repeating DIAG loop. In general, a DIAG message does not require a response.

The 4-byte *code* field specifies the type or category of diagnostic message that is contained within the *message* field. All the codes indicate that delivery failed. The *code* is always interpreted and must contain one of the values listed in Table 2-5.

Val is defined for passing an additional word of information with each *code*. This field is interpreted always and must be one of the values defined in Table 2-5.

Message is an optional variable-length string. Receivers are not required to interpret this field. When this field is present, padding may be required to make the Information Segment a multiple of 4-bytes. (See Section 2.5.1, "Padding.")

CODE	*Message category*	*VAL*
1	Invalid context	MBZ (must be zero)
2	Context refused	0=unspecified reason 1=*options* refused 2=no provider for requested service
3	Unknown destination	0=unspecified "unknown destination" 1=address family (aformat) not supported
4	Dead host	MBZ
5	Invalid route	MBZ
6	Redirect; use a different switch	MBZ
7	Cannot route	0=unspecified "cannot route/forward" 1=address family (aformat) not supported 2=insufficient bandwidth
8	No resource	MBZ
9	Protocol error	0=unspecified protocol error 1=malformed address segment 2=invalid route field of zero 3=invalid retransmission request 4=maxdata error: data segment too big
10	Maxdata error: cannot forward	=acceptable max value to use

Table 2-5. DIAG codes and sub-codes (`val`) for `message` categories.

2.5.4.5 ROUTE Packet Format

When the *type* field of the XTP Header is *ROUTE*, the Information Segment contains the following three fields: *code*, *val*ue, and *message* as illustrated in Figure 2-28. No user (higher-layer) data can be included with this format. ROUTE packets do not consume sequence space.

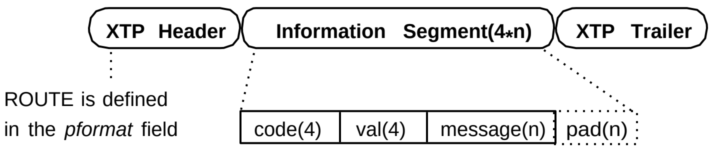

Figure 2-28. ROUTE middle segment.

ROUTE packets are defined for exchanging control information with XTP switches. (Section 3.2.7 "Path Release" explains the use of ROUTE packets.)

The 4-byte *code* field specifies the type of message that is contained within the *message* field. The *code* is always interpreted and must contain one of the values listed in Table 2-6.

Val is defined for passing an additional word of information with each *code*. Currently, no ROUTE codes use the *val* field. This field is not interpreted in this version of XTP and must be zero.

CODE	*Message category*
1	Release route
2	Release acknowledge

Table 2-6. ROUTE codes.

Message is an optional, variable-length string. Receivers are not required to interpret this field. When this field is present, padding may be required to make the Information Segment a multiple of 4-bytes. (See Section 2.5.1, "Padding.")

2.6 XTP Trailer

The XTP Trailer must be aligned on a multiple-of-4-byte boundary. (Refer to Section 2.5.1, "Padding" for a discussion of how *align* padding can be added to the end of the Information Segment in order to make it a multiple of 4 bytes.)

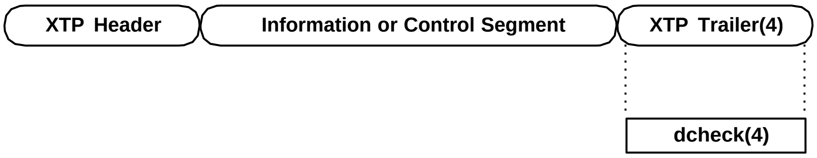

Figure 2-29. XTP Trailer.

Dcheck is a 4-byte data integrity checksum function calculated over the XTP information or control segment. *Dcheck* is calculated over the entire middle segment including any padding bytes defined by *offset* or alignment bytes at the end. The checksum function for the middle segment is disabled at the receiver when the `NOCHECK` bit is set, meaning that the receiver does not calculate a checksum for the middle segment of the packet and does not interpret *dcheck*. (Refer to Appendix A for a full description of the XTP checksum function.)

2.7 Encapsulation

An encapsulation procedure specifies how to incorporate protocol data units (PDUs) from one layer within the PDUs of a lower layer. An XTP packet is contained, or encapsulated, within a MAC frame as shown in Figure 2-30. The XTP packet can directly follow the MAC header, although in many cases additional Layer 2 or Layer 3 protocol headers are required as depicted by the dotted outlines in the figure.

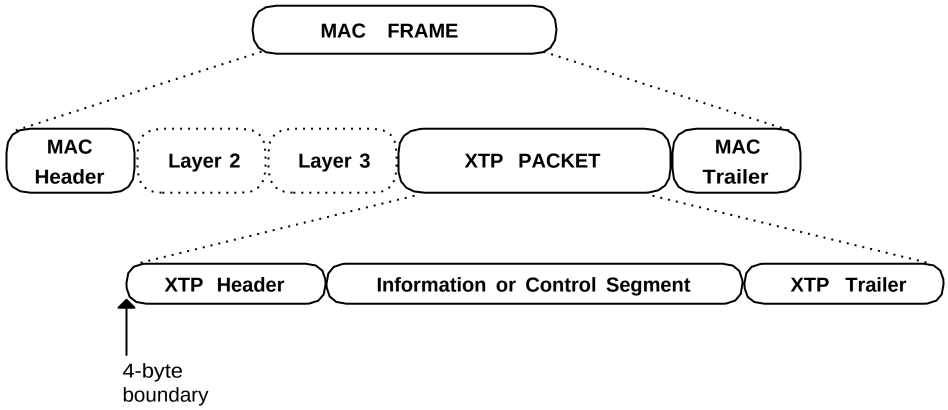

Figure 2-30. Encapsulated XTP packet.

A MAC header typically contains source and destination MAC addresses plus media and MAC-specific information. Some MACs such as Ethernet contain a mechanism for demultiplexing different protocols (the Ethernet *type* field). Other MACs require demultiplexing at Layer 2. A MAC trailer usually contains a cyclic redundancy check (CRC) function for the associated packet.

XTP utilizes the following information from the lower layers:

1) source and destination MAC addresses or their equivalent,
2) a CRC or equivalent check of the MAC frame,

An XTP packet and its internal segments are defined on 4-byte boundaries in host memory so that computers can access protocol control fields without incurring the overhead of crossing word boundaries. In order to achieve this alignment in systems that represent a MAC frame in contiguous memory, the frame should be offset in memory to account for various odd MAC header sizes. These alignment offsets are shown in the following sections by a dotted line box to the left of the MAC header. The *adj(n)* value within the box indicates the suggested offset. It is important to note that the *adj(n)* bytes are not transmitted or received on the network. That is, they are not part of the PDU encapsulation but it is convenient to display the offsets in this chapter.

If XTP is used in a secure environment and if security guidelines are followed, then the complete XTP packet could be encrypted and encapsulated within a "security frame." XTP should not be affected by such an encapsulation, and the design of a security frame is outside the scope of XTP. A security encapsulation is logically no different from the other encapsulations although it is doubtful that the real-time performance levels of the protocol subsystem would be matched by the encryption subsystem unless specialized hardware were available.

2.7.1 Ethernet Encapsulation

For XTP machines that are interfaced directly to an Ethernet, the packet format is as shown in Figure 2-31:

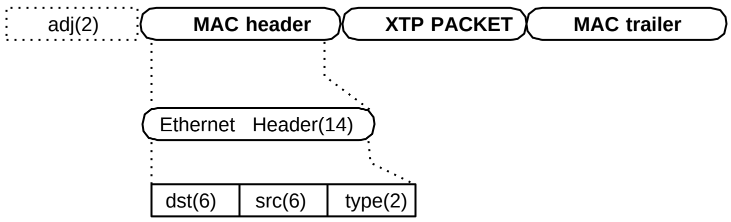

Figure 2-31. Ethernet encapsulation.

The *dst* (destination MAC address), *src* (source or sender's MAC address), and *type* fields are as defined for Ethernet. The Ethernet preamble and trailer are not shown. The Ethernet packet type for XTP is 0x817D. The 2-byte *adj* aligns the XTP packet.

2.7.2 LLC Encapsulation

XTP encapsulation for the IEEE 802.2 LLC environment follows the guidelines established in RFC 1042 "Standard for the Transmission of IP Datagrams over IEEE 802 Networks", which uses the Sub-Network Access Protocol (SNAP, IEEE Standard 802.1, for identifying private protocols that use the services of 802.2) as illustrated in Figure 2-33. The SNAP header in Figure 2-33 is an extended form of the basic LLC header portrayed in Figure 2-32.

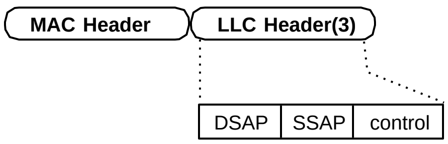

Figure 2-32. LLC header.

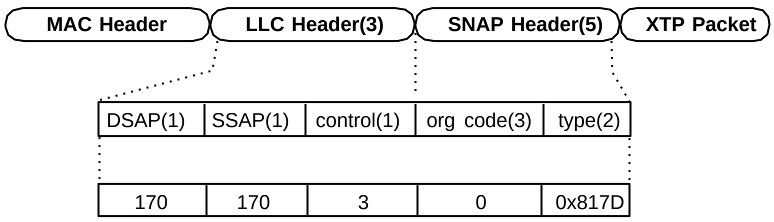

Figure 2-33. SNAP encapsulation.

In this format, the fields are assigned the following values: DSAP and SSAP are both 170 (decimal); the *control* field is 3 (Unnumbered Information); the *organization code* is 0 (zero, the code assigned to XEROX Corporation); and the *type* field is the same as for Ethernet (0x817D). These numbers are all given as big-endian, bit-order values.

2.7.3 FDDI Encapsulation

XTP encapsulation for FDDI follows the guidelines established in RFC 1103 "A Proposed Standard for the Transmission of IP Datagrams over FDDI networks." This encapsulation frame includes the LLC and SNAP headers as defined in Section 2.7.2.

XTP encapsulations for 16-bit and 48-bit address FDDI are shown in Figure 2-34. Protocol implementations should apply a 3-byte offset to the start address of packets to align the XTP packet.

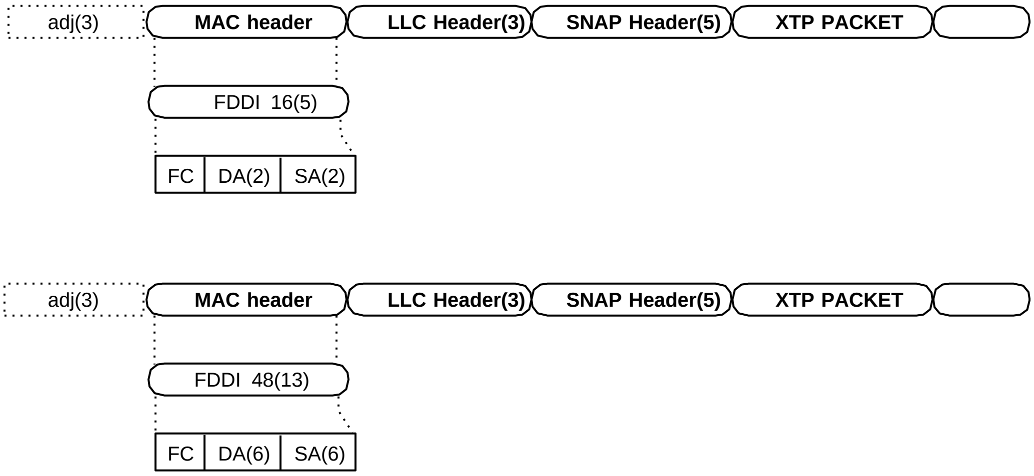

Figure 2-34. FDDI encapsulation.

2.7.4 IEEE 802.5 Encapsulation

Encapsulation for IEEE 802.5 "Token Ring" uses LLC and SNAP headers in the same manner as FDDI. (The LLC and SNAP headers are defined in Section 2.7.2, "LLC Encapsulation.") The 2-byte adjustment (*adj*) field is required to align the XTP packet. The details are illustrated in Figure 2-35:

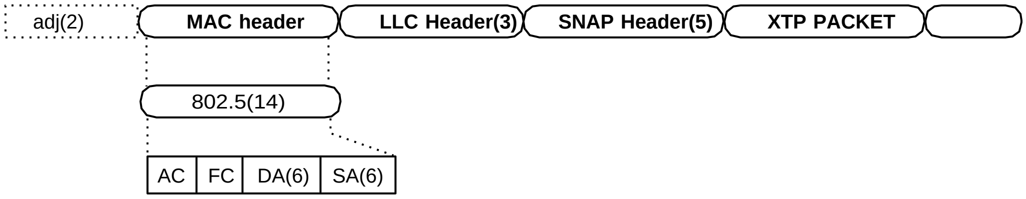

Figure 2-35. Token Ring encapsulation.

2.7.5 Internet IP Encapsulation

An XTP packet is encapsulated as the data portion of an IP datagram as illustrated in Figure 2-36 and defined in RFC 791. LLC and SNAP headers will be included as specified in RFC-1103 when IP datagrams are transmitted on FDDI networks. The 8-bit "protocol type" field in the IP header is loaded with the value 36 (decimal) to specify XTP as the next higher-level protocol.

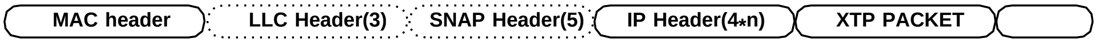

Figure 2-36. IP encapsulation.

3. Protocol Description

XTP is the set of message sequences that may be transmitted between endpoints (hosts), together with some conventions regarding information exchanges between XTP and upper-layer entities. The protocol is defined here by a textual description of the message handling procedures. Several processes and data structures are defined to assist in describing the protocol.

3.1 Conventions

3.1.1 Processes

This protocol description mentions four components, or processes: **receiver**, **sender**, **reader**, and **writer**. These processes are not part of the protocol but are helpful in explaining the protocol operations. The *receiver* acts on packets received from the network. The *sender* formats data and control packets, transmits packets to the network, controls timers, and implements the flow, rate, and error control procedures. The *reader* and *writer* transfer data and commands between a host, or application, and the sender/receiver. Figure 3-1 illustrates these four process abstractions and their relationships to each other.

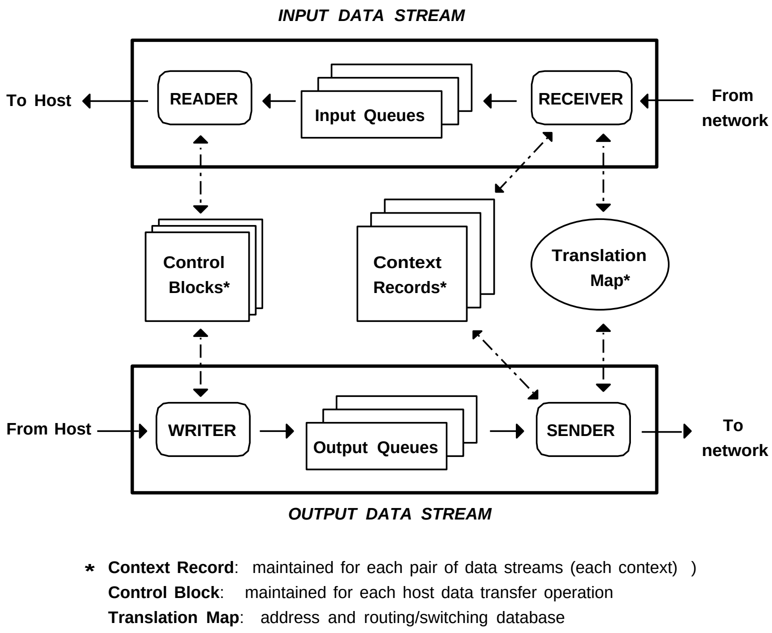

Figure 3-1. Protocol processes.

3.1.2 Services

Operating systems usually define an internal input/output "transaction record" or system call "request" structure. These objects control the flow of commands, data, and status at the boundary between host operating system and protocol subsystem. The *control block* referred to in this chapter is an abstraction for this system-dependent structure. It contains the command and data reference information common to all operating systems and records the state of the reader and writer.

3.1.3 Buffers and Messages

A *buffer* is an arbitrary area of memory designated by an application program. A buffer may be a block of contiguous memory, or, it may be represented as a set of virtual *pages* in an operating system. An application may wish to describe a buffer as a list of non-contiguous memory segments, often called a scatter-gather list. The distinctions between pages and lists are important issues, but are too system-dependent to be discussed in this document. Instead, a buffer will be represented by an (address, length) pair with the understanding that it implies a set of pages and/or a scatter-gather list.

A *packet* is the data transfer unit defined by a network's Medium Access Control (MAC) layer. A buffer may fill several packets since there is no restriction on size. A *message* consists of one or more buffers. On output to the network, the last buffer of a message is marked by the application in the control block.

3.1.4 Default Parameters

Default, or startup, values must be provided for several local XTP parameters. The values must be set by the host or XTP subsystem for the protocol to operate. But the parameters should also be adjustable by application software. Table 3-1 lists several parameters that are candidates for an application interface.

Parameter	*Function*
address	destination address segment.
maxdata	Default output data segment maximum.
rtt	The initial assumption for round-trip time.
burst	Sender's initial maximum packet burst per context.
rate	Sender's initial maximum data rate in bytes per second.
oqmax	Maximum size of output queue per context.
iqmax	Maximum size of input queue per context.
ralloc	Default allocation permitted by receiver.
talloc	Default allocation assumed by sender.
CTIMER	Default connection activity timer value.
CTIMEOUT	Default connection timeout value.
WTIMER	Initial value for acknowledgement wait timer.
retrycount	Number of unacknowledged SREQs to cause an connection abort.

Table 3-1. Default parameters for XTP implementations.

There may be operational reasons why an system may wish to prevent an application program from setting all of these parameters. Resolution of such issues is the responsibility of the protocol implementation. Appendix D presents the application interface from an XTP implementation that may be useful as a model.

3.1.5 Notation

In addition to the notations introduced in previous chapters, the following conventions are utilized in this chapter:

- Within text, internal implementation variables that are not part of the protocol are indicated by a contrasting font: e.g., **name**.
- In figures, sequence space is illustrated as "moving" to the right where the rightmost byte is the first byte sent or received.
- In figures, increasing sequence numbering is indicated by plus signs (+): e.g., n++...n+...n, where n is the smallest sequence number.
- Within text, shorthand notation, as described below, is used to represent the semantics of XTP packets.

Shorthand notations for XTP packets are employed throughout this chapter to describe protocol exchanges. The basic forms consist of a bracket notation for variables used for packet identification (i.e., table lookups) and a parenthesis notation for the other parts of a packet. The bracket notation represents four variables: MAC address for the immediate destination (next node), MAC address of the current sender (i.e., the source address), context identification value (*key*), and path identification value (*route*). For example, [A, B, K, R] represents a packet destined to machine A from machine B, using *key* K, and *route* R. The form [B, A, K′, R′] indicates that the values for *key* and *route* are *return* values (i.e., the most significant bits are set). Figure 3-2 illustrates the bracket notation.

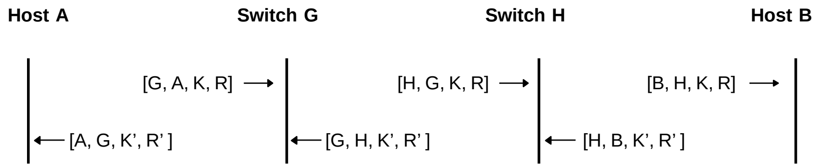

Figure 3-2. Bracket notation.

Two endpoints (hosts), A and B, communicating via XTP will each have an active *context* and *route* for their association. The context at host A will be referred to as context A. The *key* at host A for context A will be denoted as K_A, and the *return key* for context A at host A is K_A'. Similarly the *route* variable at A is R_A, and the *return route* is R_A'. The *key* values for multiple contexts at host A will be represented as K_A, J_A, I_A. Figure 3-3 illustrates the use of return values.

The parenthesis notation (*pformat*, `BITVALUES`, *field*^{i}=val^i, ...) describes the remainder of a packet where only set bits and fields that are relevant to the topic are listed. The *pformat* is as defined in section 2.3.3.1. The optional `BITVALUES` are taken from the set of Header *cmd options* bits defined in section 2.3.3.3. Since the bit names are unique, no distinction is made in the notation as to whether the bits appear in the Header or Trailer. For example [B, A, K, R] (FIRST, `NOERR`+`END`+`SREQ`) denotes a FIRST packet to B from A with the `NOERR`, `END` and `SREQ` bits set. A *field* may be the name of any Header or Trailer field. For example, a control packet with a value E in the *echo* field is written as (CNTL, echo=E). The [A, B, K, R] notation can be used alone or together with the (pformat, bits, fields) notation. Similarly, the (pformat, bits, fields) notation can be used alone or together with the [ ] notation. Figure 3-3 illustrates the two notations used together.

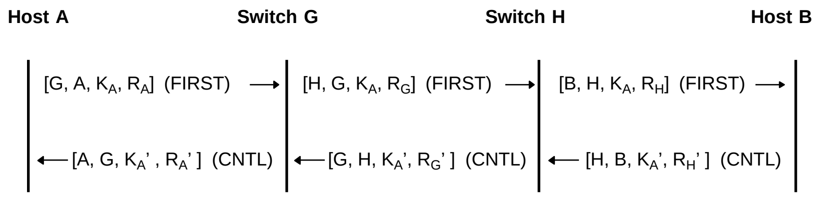

Figure 3-3. Notation example.

3.2 Management Procedures

This section describes procedures for managing contexts and connections. The section is divided into the following sub-sections: opening a context, performing a *key* exchange, closing a context, setting up a path, performing a *route* exchange, sharing *route* values among contexts, and releasing a path.

3.2.1 Context Management: Open

Before any XTP packets can be sent or received, an active XTP context must exist. Context management defines the states of an XTP context and transitions between states. This section describes the end-to-end procedures used to open XTP contexts and associations. Section 3.2.3, "Context Management: Close" describes the procedures for closing contexts and associations.

A context may be in one of the following major states: Null, Listen, Active, Inactive. Figure 3-4 illustrates these states and the transitions between them.

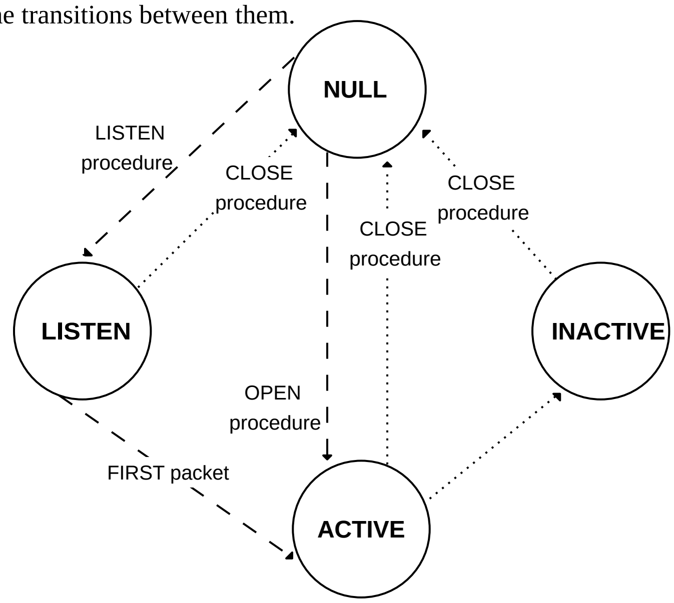

Figure 3-4. Possible states for each context.

All contexts are initially in the Null state. A Send or Connect operation on a Null context takes it to the Active state. A Listen operation on a Null context takes it to the Listen state.

The procedures for activating a context, i.e. reaching the active state, are described in this section. Thr procedures for closing a context, i.e. reaching the null state, are described in section 3.2.3. An equivalent, but more detailed, state machine model is given in Appendix C where additional states and transitions are added to model input and output queues, timers, and system events.

An **association** between contexts is created by transmitting a FIRST packet from one endpoint’s Active context to another endpoint’s Listening context, which causes the Listening context to transition to the Active state †. This association then persists for the duration of one or more message transfers between the contexts. Note that in the case of multicast (see Section 3.4, "Multicast") there may be more than one context in Listen state waiting for the same FIRST packet.

The open procedure is based on the use of a FIRST packet, the *key* field, the *seq* field, and optionally, the `SREQ` or `DREQ` bit and CNTL packets. The procedure is that the initiating context sends a FIRST packet to an end system where there is a listening context. This FIRST packet is all that is required to create an association between two (or more) contexts. The `SREQ` or `DREQ` bits may be set by the sender to effect an acknowledgement policy. If either of these bits is set, the receiving context responds with a CNTL packet. Note that a FIRST packet contains an *Information Segment* consisting of an *address segment* and an optional user *data segment*.

Figure 3-5 depicts four variations of FIRST packet handshakes. In the examples of Figure 3-5, Host A is the initiator and Host B is in Listen state. In all the examples the FIRST packets contain the destination MAC address for B, the source MAC address for A, A's assignment of a *key* value K_A for its context, and A's assignment of a *route* value, R_A. This is written as [B,A, K_A, R_A] (FIRST) according to the notation explained in Section 3.1.5.

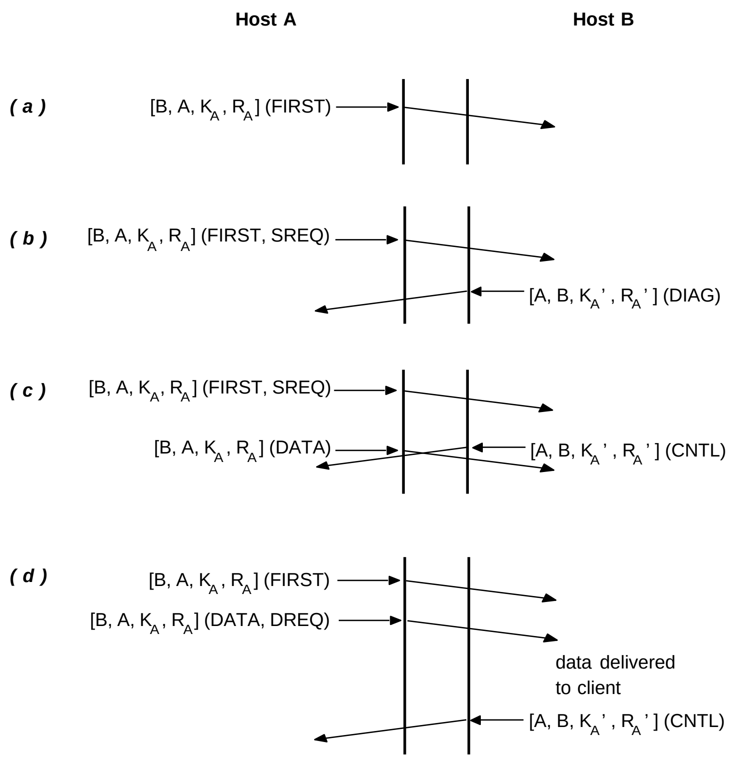

Figure 3-5. Use of FIRST packets.

Figure 3-5, example (a) shows a FIRST packet with no control bits set. If received at B, the association is established. Host A would need to send a later packet containing `SREQ` or `DREQ` to obtain a positive acknowledgement.

Example (b) illustrates a possible error scenario where the FIRST packet arrives at host B but is rejected with a DIAG message. See Section 2.5.4.4 for a list of possible diagnostic messages.

Example (c) depicts host A sending two packets to host B. The first one contains the `SREQ` bit requesting an immediate status packet in return. The second is an ordinary DATA packet. Host B responds to the `SREQ` as it arrives. The DATA packet and CNTL packet may overlap in time depending on the properties of the physical layer.

Example (d) is like example (c) but host A uses `DREQ` rather than `SREQ`. As shown in the time sequence, host B responds with a CNTL message after the data has been delivered to the application.

Whenever `SREQ` is sent (in any packet - FIRST, DATA, or CNTL), the sender must start a wait timer

(WTIMER: see section 3.6) and also save the value of *sync* in a local variable, *recv* (see section 2.3.5). If the wait timer expires before a CNTL message is received, the sender normally generates a CNTL message to obtain accurate status from the other host. (See Sections 3.3.1, "Control Packet Processing" and 3.3.3, "Error Control.") However if the sender has never received any acknowledgement from the other host, then it is not known whether the FIRST packet was correctly received. In this case the original FIRST packet must be retransmitted. Because of this, receivers must check incoming FIRST packets to detect duplicates.

The receiver's actions upon receiving a FIRST packet are the following:

1. check whether the FIRST packet is a duplicate or not. Also notice if it subsumes an old context. The packet is a duplicate if:
    a. the <*srcid, key, route*> fields in the FIRST packet identify an ACTIVE context at the receiver.
    b. the *seq* field in the received FIRST packet matches the contents of a field in the ACTIVE context where the *seq* value of the original FIRST packet is saved (see step 7).

   If both conditions are met, then the FIRST packet is a duplicate: the receiver should respond to the SREQ bit if it is set, but otherwise ignore the packet.

   The FIRST packet subsumes an ACTIVE context if condition (a) is met but the *seq* test fails. In this case the ACTIVE context must be aborted.

   If either or both conditions fail, then processing continues with step 2.

2. find a listening context, K_B, matching the destination address in the *address segment* of the packet. Send a (DIAG, code=1 for Invalid Context) message if no matching context is found. Send a (DIAG, code=2 for Context Refused) if there is reason to refuse the FIRST packet. The *val* field may be used if appropriate to indicate a particular reason.

3. the implementation must provide means for the application (as a selectable option) to accept or reject the FIRST packet. If the application elects to supervise FIRST processing, then the XTP receiver notifies the application at this time. Otherwise processing continues with the next steps which complete the processing of the FIRST packet.

4. map subsequent packets containing the same <*srcid, key, route*> as in the FIRST packet to local context K_B.

5. copy *sync* from the FIRST packet to the local *recv_sync*.

6. copy *dseq* from the FIRST packet to be the initial value for *seq* in locally-generated output packets.

7. save *seq* in the context to support the duplicate test (1) in the future.

8. respond to the `SREQ` bit, if it is set.

XTP packet reception depends on unique identification of each packet transmitted or received. The identification process operates in two modes depending on whether a **key exchange** handshake (see next section) has happened.

Prior to a key exchange context identification is based on three fields: a lower layer source identifier such as a source MAC address or source NSAP (both called *srcid* or an *id* here) plus the *key* and *route* from the XTP Header. *Srcid* identifies the system that last transmitted the packet; *key* is a value selected by the system that originated the packet; *route* is a value selected by the last system to transmit the packet. Note that both *route* and *srcid* are replaced at routing or switching stages. (See Sections 3.2.2 "*Key* Exchange" and 3.2.5, "*Route* Exchange" for details.)

Because XTP threads a path through intermediates nodes (switches) to the destination host, leaving an active path behind, a sender may transmit data immediately following the FIRST packet without waiting for any response from the receiver. The receiving system can also begin transmitting its outgoing data stream as soon as it has an Active context. This style of **implicit connection setup** eliminates the latency of a preliminary end-to-end message exchange; that is, the connection is established as a side effect of the first packet.

Implicit connection setups are not required. A system may instead send a (FIRST,SREQ) packet and wait for a positive response which will contain an exchange key (see below).

3.2.2 Key Exchange

This section extends the procedures of Section 3.2.1 to incorporate the exchange of an additional *key* value between endpoints. *Key* exchange provides the mechanism for a receiving endpoint in Listen state to communicate its selected *key* to an originating context. This allows low overhead context lookups by both contexts sharing an association while at the same time preserving the ability to send data before an end-to-end handshake has occurred.

The procedure in Section 3.2.1 is based on a single value for the *key* field where message traffic in one direction(A-to-B) uses the originator's *key* (K_A) while traffic in the other direction (B-to-A) uses the *return key* (K_A'). Figure 3-6 illustrates the values that would be used for all packets on a context operating this way.

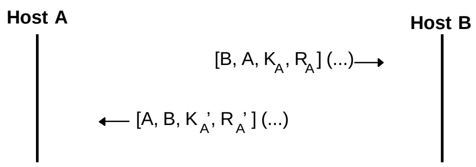

Figure 3-6. Packet exchange using a single *key* and a single *route*.

In Figure 3-6, the receiving endpoint, host B, must perform a lookup operation on the triplet <*srcid*, *key*, *route*>, represented in bracket notation as [A, K_A, R_A]. Note that, in contrast, on right-to-left traffic as shown in Figure 3-7 the <*srcid*, *key*, *route*> lookup operation is replaced by the simpler *return key* lookup (see section 2.3.4, "Key Field"). *Key* exchange provides a way to replace all the lookup operations with this simple table lookup.

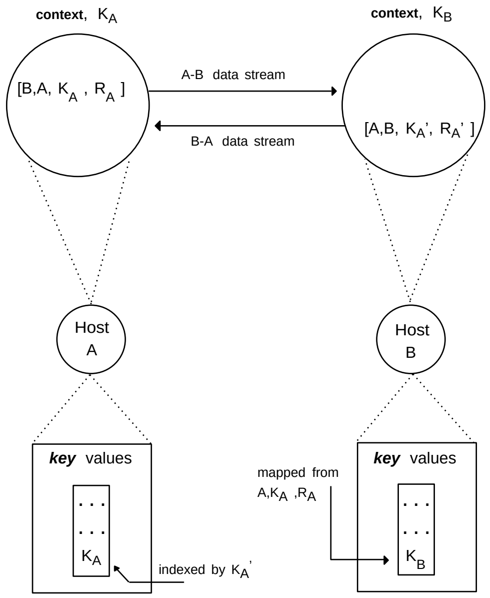

Figure 3-7. Table lookup using `return key`.

Figure 3-8 shows host B completing a *key* exchange. Upon receiving a FIRST packet, host B assigns its own identifier K_B to the new context and communicates this value to the originator within the *xkey* field of its first CNTL packet. The value placed in *xkey* must be *key*-prime. For example, host B's first CNTL packet to A's context would contain the following items for the *key* exchange: [A, B, K_A', R_A'] (CNTL, xkey= K_B').

After a key exchange, each end system knows and uses the other host's *key* value. Hosts use the *return key* value as soon as they know it. After a *key* exchange, both hosts are using *return* values and are performing simple lookups. The route exchange procedure (see section 3.2.5) accomplishes the same kind of simplification for *route* values.

Note that *key* exchange is not possible in multicast mode; it only works between a pair of contexts. A multicast sender ignores the *xkey* field in incoming control packets.

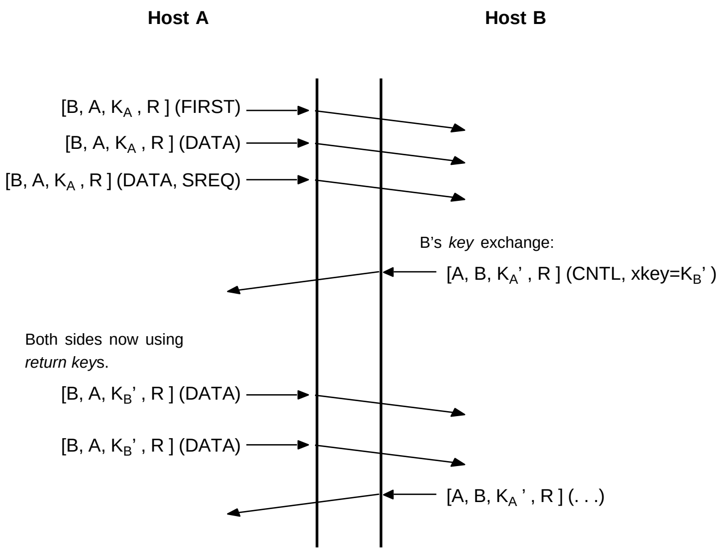

Figure 3-8. Key exchange between hosts.

3.2.3 Context Management: Close

An XTP association between a pair of contexts, A and B, involves a pair of simplex data streams: A-to-B and B-to-A. Each context is the sender for one of the data streams and the receiver for the other one. In order for an association to be terminated, both contexts must be released; each data stream must be closed with a handshake. A handshake consists of two messages (one from each end system) involving the `WCLOSE`, `RCLOSE`, and/or `END` bits. The bits may be manipulated to provide different closure semantics as discussed below. Detailed state machines for the close procedure are given in Appendix C.

The close control bits have the following meanings:

> `WCLOSE`: a data sender wants to close its output data stream.
> `RCLOSE`: a data receiver has closed its input data stream and will accept no further data.
> `END`: the context is being released; no further communication is possible.

A context is **closed** by completing the message exchanges with `WCLOSE RCLOSE` and `END` described below. A context may be **released** when both data streams have been closed and no further handshakes are possible. A context may be **aborted** when it becomes impossible to exchange messages between end systems. The normal close and release procedures are described next in this section, followed at the end by the abort procedures.

Closure can be **graceful** or **forced**, depending on the bits used and the order in which they are set.

A **graceful close** of a data stream occurs when all data for the data stream has been correctly transmitted by its sender and acknowledged and delivered by its receiver. This implies that the receiver has been provided an opportunity to ask for retransmissions. For a data stream to be closed gracefully, the message exchanges must embrace the following sequence of actions:

1. the sender for the data stream initiates the close by setting the `WCLOSE` bit in an output packet that either follows all the sequenced data transmitted on the stream (a CNTL packet) or contains the final segment of sequenced data (FIRST or DATA).

2. the receiver saves the WCLOSE bit as part of its local state, but takes no action until the input queue has been read by the application. The WCLOSE condition is said to be "pending" during this period of time.

3. when the input queue becomes empty and a received WCLOSE is pending, the XTP receiver must deliver an "end-of-context", or "connection-termination", indication to the application. The exact method for doing this is to be defined by the operating system environment. Many implementations use an "end-of-file" (EOF) indication to indicate a closed input data stream by returning a byte count of zero when the application reads the stream.

4. the application processes the connection-termination indication (i.e. reads an EOF indication). The application may immediately issue a system-defined CLOSE command to the XTP subsystem, or it may continue processing. The timing of the CLOSE command response depends on the application. Note that between the time EOF is delivered (by XTP) and CLOSE is issued (by the application) the value of *dseq* sent in any CNTL packets will indicate that all information in the input queue has been delivered to the application.

5. after the XTP receiver delivers EOF to the application the next step is to return RCLOSE, completing the WCLOSE/RCLOSE handshake. If the application has selected the CONFIRM mode, then XTP must wait for the application to issue a system CONFIRM_CLOSE indication or a system CLOSE indication before including the RCLOSE bit in any packets. If the application has not selected CONFIRM (see section 3.1.4), then XTP must begin to include RCLOSE in output packets.

6. these steps constitute one WCLOSE/RCLOSE handshake. The sender must repeat sending WCLOSE at increasing WTIMER intervals until one of the following occurs:
    1. RCLOSE is returned,
    2. a DIAG(no context) or other DIAG message is returned,
    3. a packet containing the END bit is returned, or
    4. no packets are returned at all and the CTIMER/CTIMEOUT/retry_count mechanism (see section 3.5) aborts the context.

The procedure for gracefully ending an association (releasing both contexts) requires two WCLOSE/RCLOSE handshakes; one for each data stream. The data streams do not have to be closed in the same manner; for example, one can be closed gracefully and the other can be a forced termination.

A **forced termination** of a data stream does not guarantee the delivery of all data. Forced termination results from any of the following actions:

1. a sending host receives an `RCLOSE` before it has sent a `WCLOSE`. This sequence aborts any further data transmission;

2. either host receives an *END* bit. This bit aborts all communication for the association since one of the contexts has been released.

3. either host receives a DIAG (code =Invalid context) packet. This message indicates the other host's context is closed. The host receiving the DIAG packet must close its associated context.

A **foreshortened** close procedure is permitted (described in examples below) whereby some of the graceful close handshakes can be implied using the END bit. In all non-foreshortened cases a context may be released whenever the END bit is received for that context. However whenever a system transmits the END bit, whether it is part of a foreshortened close procedure or an abort procedure, the system transmitting the END bit must delay releasing the associated context for at least CTIMEOUT (see section 3.4) seconds. This constraint is needed for proper recovery if the message containing the END bit is lost (see example 5 below).

An implementation can, as an option, reduce the waiting time after sending END by executing some additional tests. If the system sending the END bit is the one that also sent the original FIRST packet, then it knows that the FIRST packet will not be retransmitted. In this case the context can be released immediately. If the system that wants to send END did not originate the FIRST packet, then it will also know whether it has completed a *sync/echo* exchange with its remote peer. If this exchange has occurred, then there is no possibility of a FIRST being retransmitted by the other system. In this case the context can be released without waiting.

A host’s response to the first close message within an association is determined by the format of that message. Table 3-2 summarizes all the possible initial close messages and the responses to them. It is assumed that any message not containing END will also contain SREQ, so SREQ is not explicitly shown in the table. The notation "*END" means "any message that contains END". Since the END bit overrides all the others, it can be considered as a special case.

Initial message	*Possible responses*
WCLOSE	*1. CNTL packet indicating non-empty input queue.* *2. CNTL packet requesting retransmission.* *3. CNTL+RCLOSE if input queue is empty application selected FASTNAK.* *4. CNTL+RCLOSE if application did a system CLOSE.* *5. DIAG message if context is no longer ACTIVE.* *6. any of the above plus WCLOSE if the other data stream is also being closed.* *7. any of the above plus END if the context is aborting.*
RCLOSE	*1. send no more DATA packets, acknowledge with WCLOSE.* *2. aborting with END is possible.*
WCLOSE+RCLOSE	*1. acknowledge RCLOSE with WCLOSE, and send no more DATA;* *also include the possible WCLOSE responses.*
**END*	*1. release context; no further messages, or* *2. DIAG message if context is no longer ACTIVE.*

Table 3-2. Responses possible to an initial close message.

The remaining paragraphs discuss five examples of common closing sequences:

1. graceful closes for both data streams, performed with 4 messages (Figure 3-9),
2. graceful closes for both data streams, performed with 3 messages (Figure 3-10),
3. graceful closes for both data streams involving an error condition before completion (Figure 3-11),
4. a graceful close for the initiating host’s data stream (the one for which it is sender) while forcing termination of the other host’s data stream (Figure 3-12),
5. acknowledged message: a message incorporated in a FIRST packet is responded to with a control packet that terminates both data streams immediately (Figure 3-13).

Example 1: bidirectional graceful closes (4-message version)

A bidirectional graceful close consists of two handshakes resulting in graceful closure for each data stream. This mode of closing provides a **reliable transaction**. Before releasing its context, each host has received both `RCLOSE` and `WCLOSE` from the other host.

Figure 3-9 illustrates these two handshakes, an A/B close handshake followed by a B/A handshake. Host B, the receiving end of the A/B data steam, receives an initiating `WCLOSE`; it verifies that it does not require any retransmissions, sets its `RCLOSE` bit, and incorporates it into all subsequent Headers. Host A receives the responding `RCLOSE` and waits for host B to initiate a close of the B/A data stream. Host B, then initiates a graceful close of its data stream by sending a `WCLOSE`. When A receives this message it verifies that it does not require any retransmissions and sets its `RCLOSE` bit, and responds to B with `RCLOSE`, `WCLOSE`, and `END`. Since A has received both `WCLOSE` and `RCLOSE`, it releases its context. When B receives A's final message (containing the `RCLOSE`, `WCLOSE`, `END`) it too releases its context.

If the final A/B message were to not arrive, B would retransmit its (`WCLOSE`+`RCLOSE`+`SREQ`) message and would receive in reply a (DIAG, code=1 Invalid Context) message which would cause B to release its context.

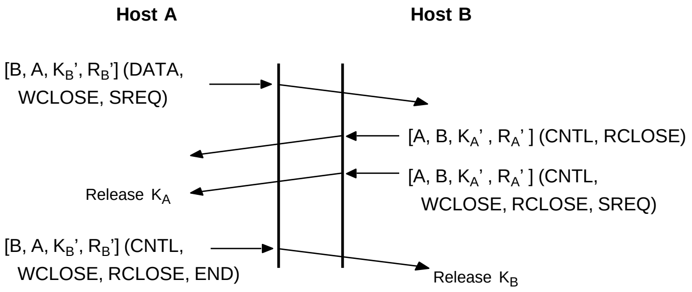

Figure 3-9. Bidirectional graceful closes: 4-message version.

Example 2: bidirectional graceful closes (3-message version)

Figure 3-10 illustrates that two of the four messages in the handshakes can be combined and still result in two graceful closes. This example still involves two handshakes (one A/B and one B/A), however bits are sent simultaneously instead of separately.

As in the first example, host A initiates a graceful close. When B responds with `RCLOSE`, it also sends its own `WCLOSE`. The messages are still error-controlled and acknowledged. This example illustrates the minimum number of messages required for graceful closing of both data streams.

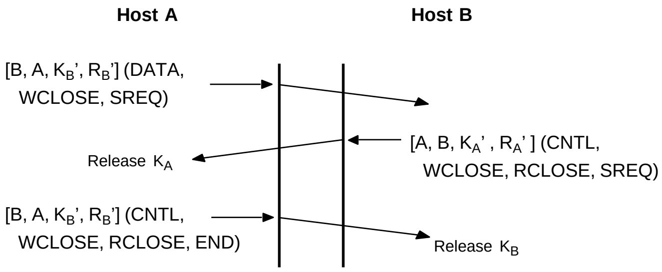

Figure 3-10. Bidirectional graceful closes: 3-message version.

Example 3: unidirectional graceful close involving an error condition

Figure 3-11 illustrates a sequence that might occur during a close when host A indicates it is ready to close and B encounters an error condition.

Host A initiates a graceful close with `WCLOSE` and B responds with an error, such as a request for retransmission of data. Host A retransmits and continues to indicate it wants to close, by using `WCLOSE`. This request/response sequence could be repeated several times. When host B is ready to close, it sends its responding `RCLOSE`, and includes a `WCLOSE` to indicate it is ready to close its own data stream. The inclusion of an `END`, as shown in this example, forces termination of the B/A data stream.

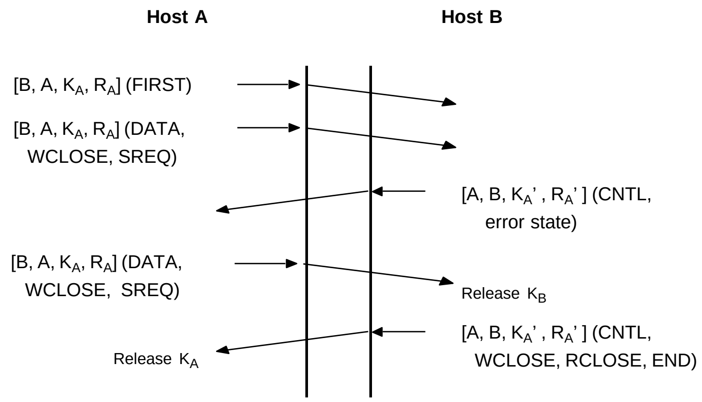

Figure 3-11. Unidirectional graceful close involving an error.

Example 4: unidirectional graceful close and forced termination

Figure 3-12 illustrates Host A initiating a graceful close for the A/B data stream and forcing closure of the B/A data stream. This procedure requires only two messages.

Host A sends a `WCLOSE`, providing host B the opportunity to request retransmissions for the A/B data stream; however, host A also sets `RCLOSE` (closing its own read process) making it impossible for B to transmit, finish transmitting, or retransmit along the B/A data stream.

Figure 3-12 illustrates a case where B sends the `END` bit and must therefore delay releasing its context in case the packet containing END is lost.

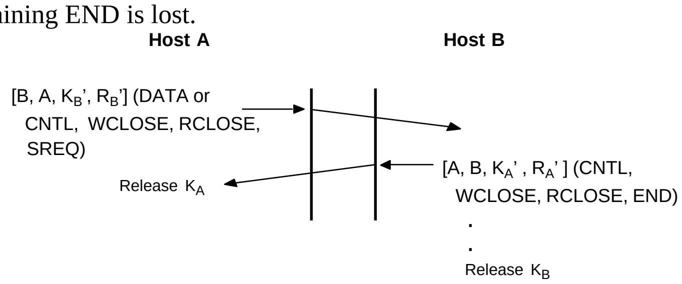

Figure 3-12. Foreshortened graceful close.

Example 5: unidirectional graceful close: acknowledged datagram

Figure 3-13 is similar to Figure 3-12 except that it illustrates a potential aliasing problem with FIRST packets that is avoided by delaying the context release after sending `END`.

Host A sends a FIRST packet that includes `WCLOSE`, thus initiating a graceful close for the newly opened A/B data stream. Transmission errors could be corrected for the A/B data stream if host B were to respond appropriately. However, in Figure 3-13, host B responds with `RCLOSE`, thus gracefully closing the A/B data stream. This action allows host A to know the state of host B’s read process before B terminated its context. In the same message, host B forcefully closes the unused B/A data stream by setting both `WCLOSE` and `END`.

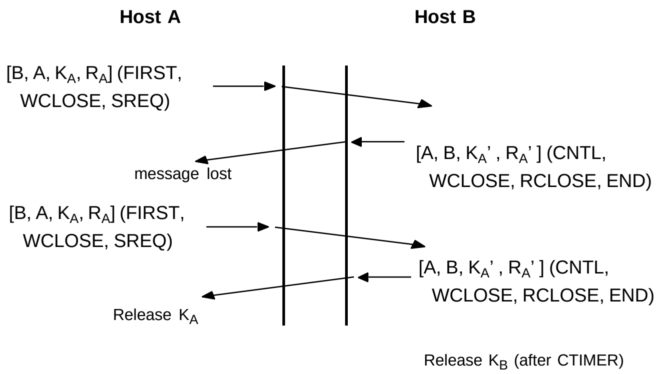

Figure 3-13. Use of `END` bit.

Now suppose that the B-to-A CNTL response is lost as depicted in Figure 3-13. Host A will not know that host B wants to release its context. If host B immediately releases its context and host A retransmits the FIRST packet, then B will incorrectly view the duplicate FIRST packet as a new one. In this situation B must delay releasing its context for CTIMEOUT seconds so that the second transmission of the FIRST packet will be detected as a duplicate and will generate the desired response.

Implementation Note
The two-packet message exchange involving foreshortened graceful closes as depicted in Figures 3-12 and 3-13 were designed to reduce end-to-end messages. This is significant only when processing many short duration associations.

For many implementations of XTP it may be advantageous to multiplex a series of transactions or messages between the same end systems over a single long-lived association rather than attempt to rapidly create and close many contexts. This approach avoids the subtle details of 2-message contexts. More importantly it may eliminate significant context management overhead.

XTP defines a connection timer, called CTIMER, in section 3.5.2 as well as a retry counter, called *retry_count*, in section 3.1.4. A CTIMER value of zero means the CTIMER is disabled. A *retrycount* of zero means the retry counter is disabled. At least one of the two mechanisms must be enabled, and it is permitted for both to be enabled.

When CTIMER is enabled it causes a CNTL(SREQ) packet to be sent if no packets were received in the previous CTIMER interval.

When there is no response to an SREQ, whether or not the SREQ was caused by CTIMER or by normal operation of the protocol, the sender may try again at WTIMER intervals to solicit a response. When the sender enters into the condition of retrying CNTL packets at WTIMER intervals, either the *retrycount* or CTIMER will terminate the process if there continues to be a lack of response. The *retrycount* specifies the number of CNTL retries to try before declaring the association closed. The CTIMER specifies the amount of time to wait while retrying CNTL(SREQ) packets before declaring the association closed.

When XTP is operating in an internet environment, the implementation must attempt to establish communications using a PATH packet before aborting an association. The implementation is given the choice of when to attempt the reconnection. For example, the PATH packet can be sent with the second retry or it may be sent after *retrycount* or CTIMER have expired. In a more aggressive environment the PATH command can be sent when the connection is first established to create either a "hot-standby" path or an active parallel path.

3.2.4 XTP Switching: Path Setup

When establishing an association between hosts A and B in a situation where the two end systems are not on the same local network, the FIRST and subsequent packets must travel through one or more intermediate nodes. XTP intermediate nodes forward packets to their destination using a technique called *cut-through switching*. In this method a FIRST packet leaves a trail of references (a path) behind as it passes between intermediate nodes. Each intermediate node forwards the packet based on the contents of the address segment in the FIRST packet. The forwarding selection procedure is implied by the *aformat* code in the address segment.† Each intermediate node also creates a mapping from <*srcid*, *route*> to an exit port so that subsequent packets will take the same path to the destination as the FIRST packet.

Once created, a path remains in place until released or until it fails. (See Section 3.2.7, "Path Release.")

The following description of path setup at first ignores the fact that intermediate networks may have a smaller frame size, or maximum transmission unit (mtu), than the originating network. The procedure for adjusting the maximum packet size at the originating sender is defined at the end of this section.

The initiating host is responsible for sending the FIRST packet to a local switch, G, and for choosing an arbitrary, implementation-unique value, R_A, for the *route* field in the Header. The syntax for the FIRST packet becomes [G, A, K_A, R_A] instead of [B, A, K_A, R_A] as described in Section 3.2.1, "Context Management: Open." Each node along the path between A and B forwards the FIRST packet by examining the *id* and *route* fields in the FIRST packet, and replacing the *route* with its own value. The *key* must never be modified by an intermediate node.

In the example of Figure 3-14, the path consists of two intermediate nodes, G and H, between hosts A and B. G would forward A's packets to H using the form [H, G, K_A, R_G]. H would forward to B using [B, H, K_A, R_H]. Packets traveling the reverse path from B through H and G to A would use the *return route* values. Host B would send [H, B, K_A', R_H'] to H. H would send [G, H, K_A', R_G'] to G, and G would send [A, G, K_A', R_A'] to A.

Failure notification is accomplished by sending a DIAG packet back to A indicating the reason.

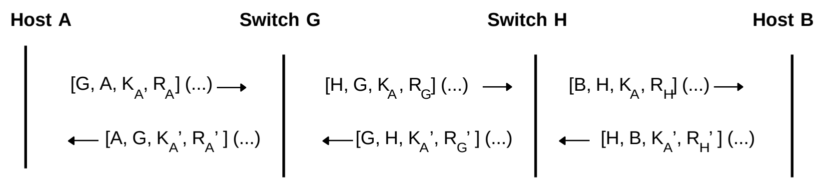

Figure 3-14. Traversing intermediate nodes.

A host setting up a context with a FIRST packet must always specify *id*, *key*, and *route* values. Each *route* value specifies a route table entry in the host where the round trip time and rate control values are

† Many of the address families specified by the *aformat* field in the XTP address segment are associated with one or more routing protocols. An XTP forwarding system must incorporate the routing protocols of the address families that it interprets.

maintained and shared by all the contexts using that *route*. Consequently, the number of active *route* entries that an XTP protocol machine maintains is always equal to the number of hosts in communication with that XTP machine, although the number may be less than the number of active contexts. The number of active *route* entries is not affected by the number of nodes on a path (i.e., whether XTP switches are employed or not).

After performing a *route* exchange (see Section 3.2.5), traffic in both directions can pass through an XTP intermediate node with a minimum of effort. An incoming packet is then processed as follows:

1. map the incoming *route* value to a routing table entry, giving the exit port. If the *instance* field in the *route* indicates that the *route* value is out of date, send a (DIAG, code=5 for Invalid Route) packet back to the sender and don't forward the packet. Otherwise, continue processing.
2. replace the *route* value in the packet with the one from the table.
3. decrement the *ttl* field in the Header by the appropriate amount.
4. queue the packet for the exit port I/O channel.

When a switching node receives a FIRST packet for forwarding, it must examine the *maxdata* field in the *address segment*. This field expresses the upper limit of data segment size that the sender will transmit. The FIRST packet may or may not contain a segment of this size. If *maxdata* contains a value that is too large for forwarding to the destination network, then the switch responds with a (DIAG, code = 10, new maxdata) to the sender and then discards the FIRST packet. The DIAG message indicates that *maxdata* is too large for forwarding, and it supplies a new value for the sender to use.

The originating system may be forced to reduce its value for *maxdata* more than once as a path is being set up. When the path is complete, both end systems will have the same value for *maxdata*. All packets can be forwarded without exceeding the mtu of an intermediate network, thus eliminating the need for fragmentation (and reassembly).

Implementation Note
An implementation can include *maxdata* in the cache of destination host information that it maintains. If this is done, the probing style of determining *maxdata* in an internet environment can be eliminated after the initial trial.

3.2.5 Route Exchange

This section extends the procedures of Section 3.2.4 to incorporate the exchange of an additional *route* value between endpoints. *Route* exchange allows receivers to assign the *route* value used by the adjacent sender. This allows low overhead *route* lookups for intermediate nodes.

The procedure in Section 3.2.4 switches traffic to the next node based on a single value for the *route* field. Message traffic in one direction (A-to-B) uses the *route* values, R_A, R_G, and R_H, while traffic in the other direction (B-to-A) uses the *return route* values, R_H', R_G', and R_A'. Figure 3-15 illustrates the values that would be used for all packets within an association operating in this manner.

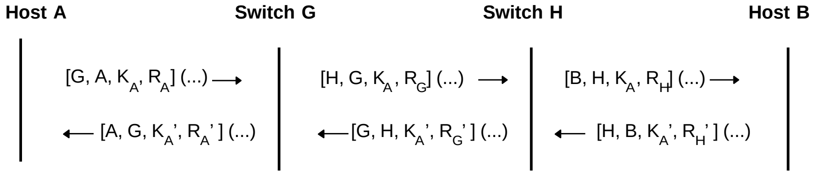

Figure 3-15. Packet exchange using a single key and a single `route`.

The receiving end system, host B, must perform a lookup operation on the triplet <*srcid*, *key*, *route*>, represented in bracket notation as [H, K_A, R_H] and, for left-to-right traffic, the intermediate nodes must do lookups on the pair <*srcid*, *route*>. On right-to-left traffic, both the intermediate nodes and the end system, host A, do *route* lookup with only a single-item, the *return route*. In summary, the route lookups in one direction are simple and in the other direction are more complex.

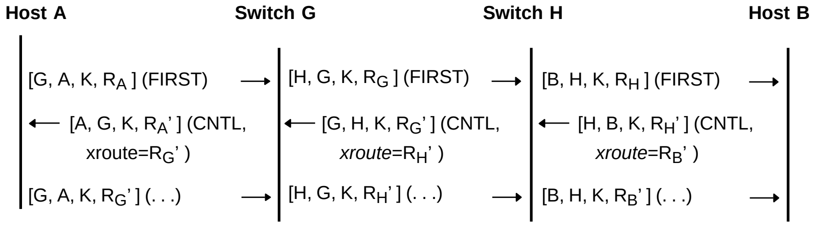

Figure 3-16. `Route` exchange.

Route exchange provides the mechanism for simplifying lookup operations in both directions. Each receiving machine (either intermediate node or endpoint) selects a *route* value for the path (exit port) used by its new context and communicates this value to the adjacent machine. The new value is communicated (with its most significant bit already set) in the *xroute* field of a control packet. In the case of the non-initiating endpoint, host B, this control packet is a CNTL packet. Figure 3-16 illustrates a route exchange where a FIRST packet progresses from Host A to Host B through two switches. As a CNTL packet is returned each node traversed by the packet passes its own *return route* in the *xroute* field. The left-to-right line at the bottom illustrates each node using the return routes communicated in the CNTL packets.

Figure 3-17 presents another view of the same process, showing how A and B view one another through the switches.

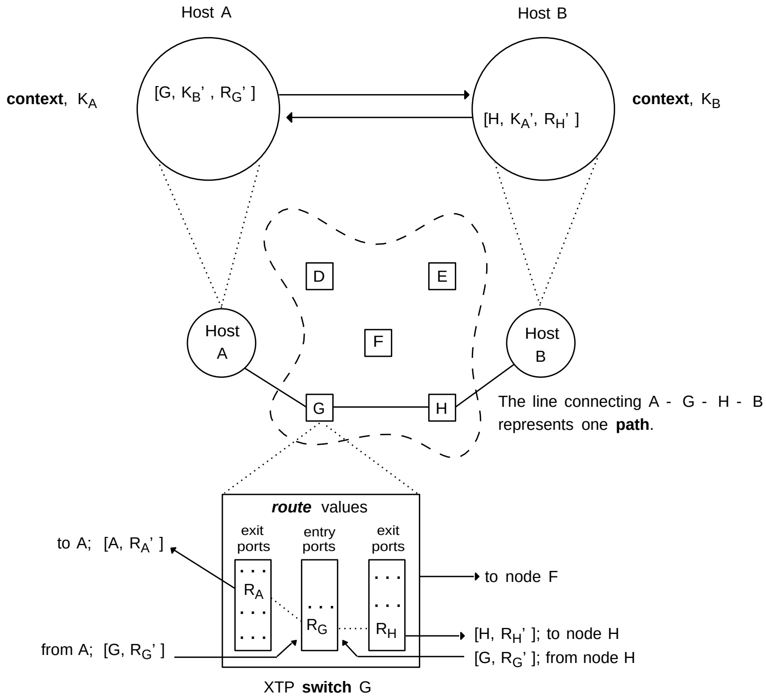

Figure 3-17. Table lookups after `route` exchange.

Upon receiving or forwarding a FIRST packet, each system assigns the association to a newly selected (or an already existing) *route* value and communicates the value to the adjacent machine. The value in *xroute* must be *route*-prime. For example, intermediate node G receives a FIRST packet from A containing [G, A, K_A, R_A]. Node G can use either of two methods for communicating its *route* value to A. G can send an RCNTL packet to A containing R_G' in the *xroute* field, or G can wait until it receives a CNTL packet destined for A, and place its value in that *xroute* field. As soon as A receives the *return route* value, it begins to send packets to G with the form [G, A, K_A, R_G'].

Figure 3-18 illustrates a complete sequence for exchanging both *key*s and *route*s over a path that includes intermediate nodes.

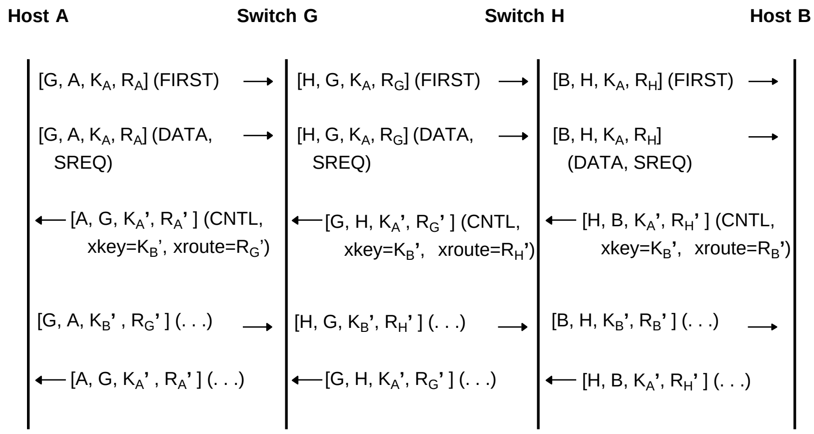

Figure 3-18. `Key` and `route` exchanges across networks.

Note that *route* exchange is not possible in multicast mode; it only works between a pair of contexts. A multicast sender ignores the *xroute* field in incoming control packets.

3.2.6 Route Sharing

The previous sections describe how an association is created between two end systems, A and B, using contexts K_A and .Kk K B . When A wants to add a second association to B using a new context, J_A, it can either create a new *route* value or use the existing one. A new *route* would be created by duplicating the procedure that set up the association for K_A. The existing, active *route*, R_A, can be used by sending a FIRST packet with [G, A, J_A, R_A], where G is the source identifier of the adjacent node, J_A is the new *key*, and R_A is the active *route*. It is very likely that *route* exchange would have already been completed with node G, in which case the FIRST packet would be [G, A, J_A, R_G'], where R_G' is the *return route* value. Figure 3-19 illustrates three contexts K_A, J_A, and I_A sharing a single *route* at host A.

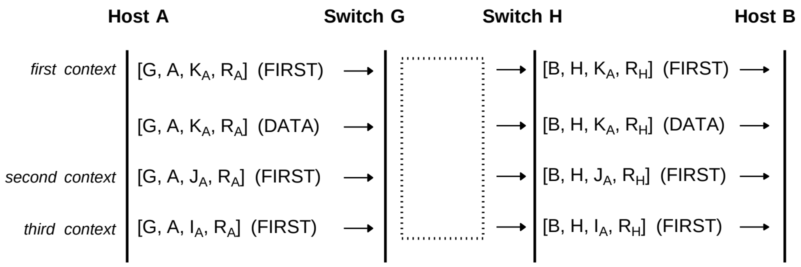

Figure 3-19. `Route` sharing.

Implementation Note
Although an individual *route* could be established for every active context, *route* sharing is intended to be the common case for two reasons. First, it reduces the active table space needed in a switch. Since table space requirements are partly a function of the number of active *route* values, the active table space would be related to the number of pairs of systems communicating through the switch. Secondly, *route* sharing implies rate sharing. The rate control mechanism in XTP regulates the throughput of packets on a particular *route* so that all the contexts sharing a *route* value also share the rate assigned to that *route*. (See Section 3.3.3, "Rate Control.")

3.2.7 Switch Management: Path Release

There is no practical way for an XTP intermediate node to determine when a host has released all the contexts sharing a *route*. Therefore a handshake using ROUTE packets is defined for synchronizing the orderly release of *route* assignments that connect to an intermediate node. The handshake is normally initiated at an endpoint, but can be started by an intermediate node.

Note that an XTP implementation must track several timer variables as part of flow and error control. These are: **rtt** for the local area network as well as a connection timer. (For details, see Section 3.6, "Timers.")

An endpoint must exchange ROUTE packets, (ROUTE, code=1 for RELEASE) and (ROUTE, code=2 for ACKNOWLEDGE) with a switch when it wants to release a *route*. Figure 3-20 illustrates this exchange.

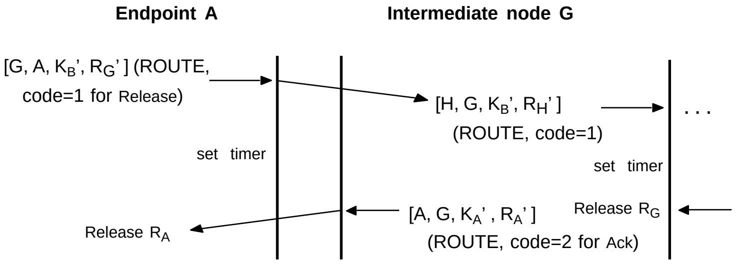

Figure 3-20. Closing `routes`.

Upon sending its "release" ROUTE packet, the sender must set a timer (WTIMER). If a ROUTE "acknowledge" packet does not return before the timer expires, the sender resends the "release" ROUTE packet. The *route* should not be released until an acknowledgement has been received, or until a number of unacknowledged "release" ROUTE packets have been sent. (The exact number is an implementation detail.)

Figure 3-21 illustrates a complete sequence of ROUTE packets for a path consisting of two intermediate nodes, G and H, between the two endpoints, A and B.

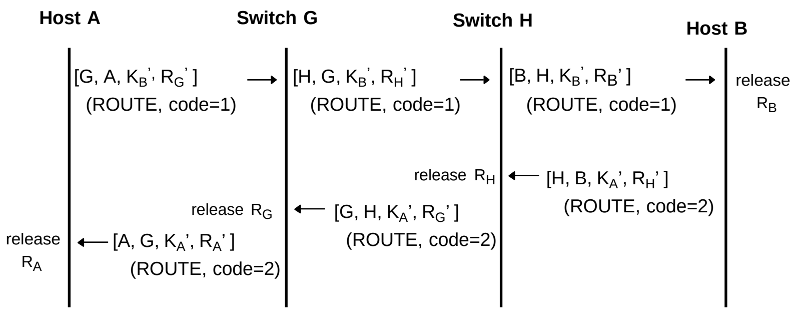

Figure 3-21. Close handshake for a path with intermediate nodes.

Beginning with host A at the left a "release" ROUTE packet traverses the A-to-B path, regenerated by G and H. The rightmost system, B, acknowledges the "release" packet with an "ack" ROUTE packet that traverses the path back to A. Host A starts two timers when the first "release" packet is sent. One timer expires after a WTIMER interval and the other expires after a CTIMER interval (see section 3.5). A uses the WTIMER to time the expected return of the "ack" ROUTE packet. If this timer expires, A resends the "release" ROUTE packet and repeats the process until either:

1. an "ack" ROUTE is returned.
2. CTIMER expires.

When one of these happens, the route data structures can be released or reused.

In addition, to enable release of *route*s when nodes fail, each intermediate system keeps a packet count for each *route* and samples the count after every CTIMER interval. If the packet count does not change during a connection timer (CTIMER) period, the *route* release procedure must be initiated by the intermediate system. This is done by sending (ROUTE, "Release") messages from the intermediate system to both exit ports of a *route*. Release messages should be repeated until there is a reply or until a CTIMER period has elapsed. The expected replies are either a confirming (ROUTE, "Acknowledge"), indicating that the ROUTE can be released, a (DIAG), indicating that the ROUTE has already been released, or a DATA or CNTL packet which keeps the route active.

It is possible that a FIRST packet may overlap the sending of a (ROUTE, "Release"). In this case the packet can be processed in the normal manner: a new exit *route* will be assigned. If a non-FIRST packet arrives, it cannot be forwarded and must be failed by returning a (DIAG, "Invalid Route") message.

If a host sends to a switch using a *route* value that the switch has released, the switch will detect the error and respond with a DIAG packet: (DIAG, code=5 for INVALID ROUTE). The host corrects the situation by sending either a FIRST or PATH packet to rethread the path.

3.3 Data Flow Procedures

This section describes the procedures for managing data flow: control packet processing, flow control, rate control, and error control. "Control packet processing" describes the manner in which an XTP implementation must generate and process control packets. "Flow control" describes the primitives available for use in limiting and regulating the volume of data flow in XTP. "Rate control" describes how the speed of the sender's output is controlled in XTP. "Error control" describes the manner in which information about lost and damaged data is communicated to the sender and the retransmission methods that are available.

3.3.1 Control Packet Processing

CNTL packets can be generated by both the sender and the receiver for a data stream. The sender generally directs the receiver when to generate CNTL packets, but there are cases when a receiver may autonomously generate a CNTL packet. Thus, XTP represents a hybrid of sender-directed and receiver-directed schemes.

CNTL packets exchange acknowledgement and state information between end systems. The sender of a data stream uses the `SREQ` and `DREQ` bits to effect the acknowledgement scheme it wants from the receiver. Whenever the receiver sees `SREQ`, it responds immediately with a CNTL packet containing its current state information. Whenever the receiver sees `DREQ`, it places a DREQ indicator in its input queue (after any data). After any data before `DREQ` in the input queue is transferred to application buffers, the DREQ indicator in the queue will be discovered and an output CNTL message generated. If there should be more than one `DREQ` in the input queue, possibly separated by application data, then an output CNTL packet should be generated for each one.

Conditions that may cause generation of control packets include, but are not limited to, the following:

- Reception of a packet with an `SREQ` or `DREQ` flag set **always** causes a CNTL packet: immediately (before sending any DATA packets) in the case of `SREQ` and after input data delivery in the case of `DREQ`.
- Reception of an out-of-sequence *data* segment may cause generation of a CNTL packet if enabled by FASTNAK mode and if implemented by the receiver.
- When no response has returned from an `SREQ` or `DREQ` and a WTIMER period has elapsed, a CNTL packet may be generated to solicit a CNTL packet in return.
- In reservation (`RES`) mode, a control packet should be sent when the higher-layer application indicates it is ready to receive more data, thus giving XTP more buffer space for the connection. This is a system dependent event usually associated with an application read operation. The XTP subsystem issues a CNTL packet with updated *alloc* values.
- CNTL packets are sent when a connection needs to be closed.

The processing steps for a received CNTL packet are as follows:

1. Locate the Active context using either the contents of <*srcid, key, route*> or the *return key* and *return route*, if provided. If no Active context is found, return a DIAG("no context") message to the sender and abort further processing.
2. Process the *sync* field according to the receiver rules in section 2.3.5. Discard the packet if indicated by *sync* processing, otherwise continue.
3. If the received *seq* value is "less than" (defined in section 2.2.4) the local **hseq** for that data stream (defined in Section 3.3.2), discontinue processing (the control packet is out of order).
4. If the *seq* number is greater than the local **hseq**, replace **hseq** by *seq*. If `NOERR` is set, also replace the outgoing *rseq* and *dseq* with **hseq**.
5. Update local *alloc*, *rseq*, *dseq*, *burst*, and *rate* control variables from the control packet.
6. Update the output *echo* and *techo* fields using the procedures specified in section 2.4 for these

fields if the received CNTL packet has the SREQ bit set.
7. Process *xkey* and *xroute* if non-zero and if the local XTP context is not a multicast sender.
8. Examine *nspan* and, if non-zero, examine *spans* for possible error control retransmission if any of the following are true:
    - FASTNAK bit is set, or
    - received *echo* matches local *sync* or *saved_sync* (refer to section 2.3.5 for *saved_sync*), or
    - received *echo* matches a multicast bucket. (Appendix B describes the bucket algorithm for multicast.)
9. If received *techo* is non-zero, then update the round trip time (**rtt**) estimate. (See Sections 2.4 and 3.6 for more details.)

3.3.2 Flow Control

The *volume* of XTP output is regulated by an end-to-end windowing flow control mechanism described in this section. The *rate* or speed at which XTP sends packets into the local network, as well as the rate at which an intermediate node forwards packets, is regulated by an independent, timer-based mechanism (as described in Section 3.3.3 "Rate Control").

Output flow control is based on a sliding window of sequence numbers. A sequence number is defined for each output byte of the data stream, starting with the initialized sequence value. Figure 3-22 illustrates the relationship between sequence numbers and protocol-related state variables.

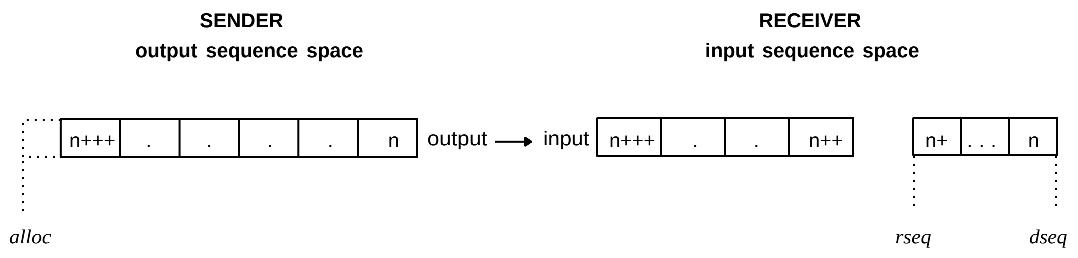

Figure 3-22. XTP flow control variables.

Alloc is the current do-not-exceed output sequence number; *rseq* is one greater than the highest monotonic sequenced byte ever received; and *dseq* is one greater than the highest sequenced byte delivered to the application. (More detail on these protocol fields can be found in Sections 2.3.7 and 2.4.)

Additional state variables, **bseq**, **eseq**, and **hseq**, are defined to clarify the relationship between internal implementation queues and protocol messages. Figure 3-23 shows all the state variables.

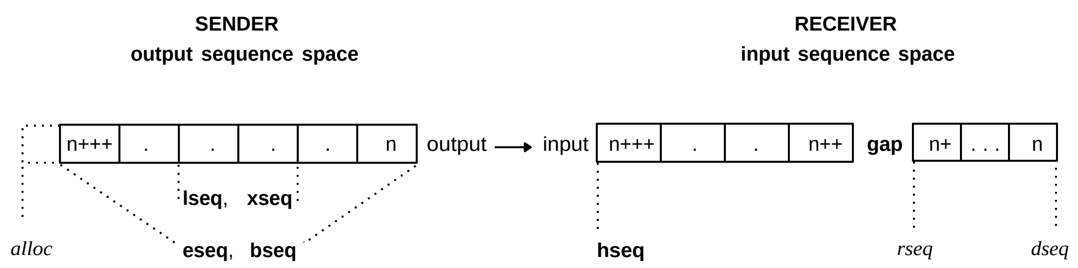

Figure 3-23. All flow control variables.

On the sending end (**bseq**, **eseq**) denote respectively the beginning and end of the sequence number space of the current pending or unacknowledged output. **Bseq** is the sequence number of the initial output byte. **Eseq** is one past the final output byte. **Eseq** increases as additional output buffers are delivered from an application to XTP. **Bseq** increases as data are acknowledged. The interval (**xseq**, **lseq**) denotes the span of sequence numbers currently being transmitted. Initially (**xseq**, **lseq**) is the same as (**bseq**, **eseq**), but this can change during retransmission.

On the receiving side, **hseq** is one greater than the highest sequenced byte ever received. If there are no errors or dropped packets, then **hseq** and *rseq* will have the same value. Otherwise there will be "holes" or "gaps" in the sequence space at the receiver as shown in Figure 3-23.

The receiving host is responsible for communicating *dseq* as it delivers data to the application. Every XTP Header communicates the state of *dseq*. Every CNTL packet also communicates *rseq*. **Bseq** at the sender is updated from the *dseq* value contained in every Header. This moves the "right hand side" of the sender's window.

The sender is not allowed to transmit sequence numbers beyond nor including the value of *alloc*. Initially, *alloc* is set as a default output parameter by the implementation. *alloc* is updated from the *alloc* values contained in CNTL packets received from the network. This moves the "left hand side" of the sender's window. The receiver is responsible for adjusting the sender's window. The allocation can be expressed as *dseq* plus a constant, C, representing the volume of data that the receiver is willing to hold for the active context.

An alternate allocation policy is enabled whenever the reservation mode bit, `RES`, is set. In the default case, *alloc* moves to the left of *dseq* based on the receiver's constant, C, or on the receiver's choice of a heuristic. In reservation mode, the allocation is strictly determined by the number of bytes available in application buffers committed to reading from the context.

A sender in reservation mode must wait for a new allocation from the receiver after sending an EOM or a BTAG. The reason for this is that the EOM/BTAG may truncate the application read operation without filling the entire buffer. The sender's allocation will then be optimistic by an amount equal to the truncated part of the buffer.

The `SREQ` and `DREQ` bits are used by the sending side to effect an acknowledgement policy. Section 3.3.1, "Control Packet Processing" describes this mechanism.

3.3.3 Rate Control

The output packet rate (speed) is regulated by two variables, (**credit** and *burst*) and a refresh timer (RTIMER). These variables are calculated from the *rate* and *burst* fields received in CNTL packets. The *rate* value specifies the maximum data rate in bytes per second. The *burst* value specifies the maximum number of bytes to be sent in a burst of packets. *Rate* divided by *burst* gives the number of burst-size transmissions per second or the rate at which the **credit** variable (see below) is refreshed. *Burst* divided by *rate* gives the time period for RTIMER. (See Section 3.6, "Timers" for more detail on RTIMER.)

With each output transmission **credit** is decremented by the size in bytes of the user data transmitted. Data transmission must cease when **credit** becomes zero or negative. Upon each expiration of RTIMER, the internal variable **credit** is updated with the value *burst*. That is, **credit** is updated approximately *rate/burst* times per second. The update procedure is as follows:

1. if **credit** is zero or negative, add *burst* to the previous value of **credit**.
2. if **credit** is positive, then replace it with *burst*.

A value of zero for *burst* means that both rate and burst are not constrained, and the host can transmit at will. A value of zero for *rate* halts DATA transmission completely, although CNTL packets at (approximately) WTIMER intervals are permitted.

Except when *rate* is zero, **credit** is decremented by the size of the information segment in each transmission. Output is permitted as long as **credit** remains greater than zero, and is suspended when **credit** becomes zero or negative. The suspended state lasts until **credit** is refreshed at the next RTIMER interval.

At the time a connection is established, a context is initialized from default parameters supplied by system management. The parameters come from the table of *route* variables if there is an active *route*. Otherwise, the values are initialized from a set of defaults. This includes the initial values for *burst*, RTIMER, WTIMER, and *rate*. Switches are permitted to decrease the values of *rate* and *burst* in CNTL packets as they pass through. As CNTL packets flow back to a sending host, updated values for *rate* and *burst* will arrive and will affect the output rate.

An intermediate node does not have to wait for a passing CNTL packet to send information to a host. It can send an RCNTL packet containing *rate*, *burst*, and *xroute* at any time.

Note that the rate control in one direction along an XTP path is distinct from and may differ from the rate in the opposite direction.

If an intermediate node records the *rate* and *burst* numbers that are passed along in CNTL messages for each active *route*, then it can also maintain the sum of these numbers. The sum represents the total bandwidth commitment made by the node. The *rate* values forwarded by a switch constitute an allocation mechanism because they affect the bandwidth commitment. Since the switch can control this sum by modifying CNTL packets, it is able to exert bandwidth allocation.

> ***Implementation Note***
> If XTP is being driven from a constant source, such as a stream or audio/video stream, then the *credit*/*burst* calculation could conceivably be omitted.

> ***Implementation Note***
> An XTP switch is expected to read the *service* field within the address segment of a FIRST packet. This field indicates the general kind of traffic that will be forthcoming. Adaptive algorithms in a switch may be able to accommodate many kinds of traffic without prior knowledge. However, it seems particularly useful to know when a new stream will consume constant bandwidth for its lifetime, hence the *isochronous* service type.

3.3.4 Error Control

In XTP, error control is based on the exchange of information regarding lost or damaged data and the retransmission of this data. Error control can be either go-back-n or selective. Retransmission is completely disabled when the sender sets the `NOERR` bit.

The error detection procedure is split between the sender and the receiver. A receiver detects missing packets by checking its incoming packet stream for sequence number gaps as illustrated in Figure 3-24. The state of the receiver input queue is communicated by CNTL packets which are transmitted to the corresponding sender in two cases: (1) on demand when an `SREQ` or `DREQ` bit is received, (2) when the receiver detects a sequence gap in the input stream and the FASTNAK bit is set. The sender is responsible for setting `SREQ/DREQ` and interpreting the CNTL packets.

The sender defines a timer, WTIMER, that detects when requested CNTL packets are not received within the expected period. WTIMER is started whenever a packet containing `SREQ` is sent. The value for WTIMER should be no less than the current round trip time, **rtt**, and include a factor accounting for system queuing delays. A setting of twice the estimated **rtt** is a reasonable place to start. (Refer to Section 3.6, "Timers" for more details.)

When WTIMER expires, one of the following conditions will hold:

1. CNTL messages were received and indicate no errors,
2. CNTL messages were received and errors are indicated,
3. no CNTL messages were received during the timer interval.

No action is needed in case (1). In case (2) the contents of the CNTL packet indicate the number of gaps (*nspan*) and which spans of output sequence numbers have been received. In this case retransmission of the gaps may begin. In case (3) the sender starts the synchronizing handshake procedure, sending send (CNTL,`SREQ`, sync=a value) messages until either:

a. a synchronizing handshake completes (see section 3.6, "Synchronizing Handshake").
b. the *retry_count* (see sections 3.1.4 and 3.5.2) is exceeded.
c. the retry timeout period, CTIMEOUT, is exceeded (see section 3.5.2).

If the synchronizing handshake completes, then either case (1) or case (2) will apply. If the retry_count or CTIMEOUT are exceeded, then the context is aborted.

Figure 3-24 depicts an input queue with two gaps in the sequence space. XTP provides for *selective acknowledgement* in this kind of situation. A receiver in this state would send CNTL messages with the following information:

1. *nspan* set to two.
2. *spans* with two pairs of numerals defining spans of sequentially received bytes: (a, b) and (c, d). Upon receiving this CNTL packet, the sender for the data stream would calculate from *rseq* and the values in *spans* so as to know which gaps to retransmit. In this example, these would be intervals (*rseq-a*) and (*b-c*). Note that the value in the second half of the final *spans* pair is always equal to **hseq**.

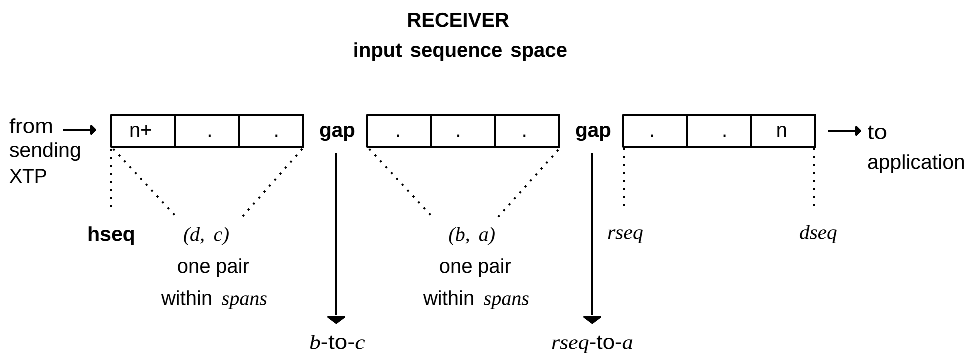

Figure 3-24. Receiver's specification of errors.

A receiver transmits its input queue *spans* list in a CNTL packet in two cases:

1. whenever the receiver responds to a received `SREQ` or `DREQ`, or
2. if the `FASTNAK` bit is set, the receiver has the option (it is not required) of transmitting its *spans* list **one time** in an unsolicited CNTL packet upon detecting an input data stream sequence gap.

The receiver is only allowed to exercise this option when *nspans* when the value of *nspan* is changing from zero to one.

Case (1) is the normal, conservative case that is compatible with out-of-order packet arrival. Case (2) represents a quicker-acting (more aggressive) error control policy that is possible only when out-of-order packet delivery is not possible. The `FASTNAK` bit enables the quicker error control reporting. The error reporting option is limited as described in option (2) in order to avoid excessive error reporting.

Maintaining selective acknowledgement queues, tracking spans and gaps, can lead to considerable complexity and overhead. This overhead can be justified when there are high network error rates coupled with low bandwidth transmission or long propagation delays (with or without high bandwidth). In these circumstances overall network throughput can be improved by selective retransmission. However, in high bandwidth, low delay, and low error rate networks, there may be no detectable performance difference between using selective retransmission or using simple go-back-n.

When the cost of selective acknowledgement queues is not justified, an XTP receiver is allowed to maintain just *dseq*, *rseq*, and **hseq**. Since the CNTL packet design contains all the state information for both policies, various kinds of XTP implementations can interoperate. A receiver that implements selective acknowledgement will interoperate with a sender that only does go-back-n, and a receiver that records only go-back-n information can interoperate with a sender that is capable of selective retransmission.

In a go-back-n receiver, whenever **hseq** differs from *rseq*, **hseq** must be returned in CNTL packets as the span interval (**hseq**, **hseq**) with *nspan* set to 1. This tells the corresponding sender that the entire interval (*rseq*-**hseq**) should be retransmitted. Without **hseq**, or a synchronizing handshake, the sender cannot tell from *rseq* alone whether there are any gaps in the remote queue. (This rule does not apply when the sender has turned error control off.)

In the most conservative form of error correction, a sender would only retransmit after a synchronizing CNTL handshake. This would normally happen only after a WTIMER interval. However, in the more aggressive form of error control allowed by `FASTNAK`, the sender can initiate retransmission after receiving certain CNTL packets. But the retransmission strategy should avoid pathological looping behavior in which errors during retransmission trigger extra retransmissions.

Implementation Note
One technique to avoid looping with go-back-n retransmission is to retransmit only once any missing data indicated by an received CNTL packet, but to not retransmit any data a second time unless an end-to-end synchronizing handshake is completed after the first retransmission. This can be accomplished by keeping an additional sequence variable, **kseq**, in the sender which is set to the maximum of *dseq* and the most recent retransmission sequence numbers. **Kseq** is reset to *dseq* after any synchronizing handshake. No retransmission is started at a sequence number less than **kseq**.

Error control (retransmission) is disabled by the `NOERR` bit in the Header. In NOERR mode, the sender ignores the *spans* field, and the receiver always sets *rseq* to one past the highest sequence number received (**hseq**), thus masking any gaps in the input queue. The sender will see no errors, but will still observe flow and rate control. The receiver must indicate to the reader (higher-layer application) whether any sequence gaps exist in data delivered to an application read buffer. This revision of XTP does not define how that should be done.

3.4 Multicast Procedures

XTP multicast is intended for media that define a broadcast or multicast facility [‡]. The multicast service delivers data to receivers that are active for the duration of the multicast transmission. Delivery is specified as reliable or best effort. Enumeration of the set of receivers or noting changes in the set of receivers requires the inverse of multicast, or *concentration*, as discussed below in Section 3.4.8. The multicast description is partitioned into the following areas: syntax, control packet processing, addressing, context management, flow and rate control, error control, joining an in-progress multicast conversation, and concentration.

3.4.1 Multicast Syntax

Multicast packets obey the same syntax rules as non-multicast packets: Header, Trailer, Control, and Information segments are identical. Multicast packets differ from unicast ones in the following ways:

1. all multicast packets have the `MULTI` bit turned on, and non-multicast packets clear the bit.
2. multicast packets utilize group addresses rather than point-to-point addresses,
3. multicast packets always use `SREQ`, not `DREQ`.

All other options and flags defined for unicast transmission are defined for multicast. The definition of how a receiver should respond to the various options is the same as in unicast except where noted below.

3.4.2 Control Packet Processing when Multicasting

In multicast mode, a sender must positively associate returned control packets with past events if continuous output streaming is desired. This can only be accomplished by matching returned *echo* values with local *sync* values. The procedures specified in section 2.3.5 keep track of *sync* and one previous value. These procedures are extended by the **bucket algorithm** (see Appendix B) to encompass a greater number of past events.

CNTL packets sent by multicast receivers in response to received `SREQ`s are multicast to the group, not unicast to the sender. This is necessary for slotting and damping algorithms. (Appendix B describes these mechanisms in detail.)

As noted in Section 3.2.2, *key* exchange is not possible in multicast mode. Consequently, CNTL packets to the multicast group that contain *return key*s must be those generated by the multicast receivers when they send CNTL packets to the group. Conversely CNTL packets containing non-primed *key*s are generated only by the multicast sender. This distinction is used to simplify organize control packet processing at a reader and at the sender.

At a multicast reader if a received CNTL packet came from the multicast sender, then the same procedures are applied to it as defined in section 3.3.1. If a received CNTL packet came from one of the other multicast readers, then the damping and slotting algorithms may be applied (see Appendix B).

At the multicast sender if a received CNTL packet came from one of the readers it is processed with the bucket algorithm. Otherwise it is a copy of a local transmission and can be ignored.

‡ Multicast service can be extended to non-multicast media via a link layer that provides output replication to a set of destinations. Such a replication layer, like a security layer, would be considered part of the media-specific encapsulation, and would be defined as part of the XTP encapsulations in section 2.7. Definition of such a replication layer is outside the scope of this document.

3.4.3 Multicast Addressing

XTP does not define how to assign a multicast group address. It is assumed that address assignments are accomplished by an outside mechanism: agreement between peers, or an address management protocol. Multicast addresses may appear within XTP in three ways:

1. in the *address segment*, Internet format as an Internet Class D address, or
2. in the MAC header as the destination address (*dstid*), or
3. in the *address segment*, Direct Address format as an implementation-dependent address.

Internet Class D addresses are defined in RFC 1112. To summarize: Class D addresses define a 28-bit space between 224.0.0.0 and 239.255.255.255. Class D also defines a mapping to MAC addresses. For 48-bit IEEE-compatible MACs, the low-order 23 bits of the IP address are placed in the low-order 23 bits of the Ethernet multicast address 01-00-5E-00-00-00. For FDDI the mapping is the same. This means that many class D addresses map into the same MAC address.

In XTP, a Class D address would appear in the address segment of a FIRST packet, and the RFC 1112 mapping of that address would appear as the destination MAC address. This is the only method whereby a 48-bit IEEE-compatible group address can be uniquely mapped back into an Internet Class D address.

3.4.4 Context Lookup

In multicast mode, as in non-multicast (unicast) mode, a received packet must be associated with a context. In previous versions of XTP the context lookup for multicast packets was different from non-multicast (unicast) lookup. Unicast lookup is based on <*srcid, context, route*>, and multicast lookup was based on <*dstid, context, route*>. This was an unnecessary complication and has been eliminated.

The mapping procedure is the same in both cases: <*srcid,context,route*> are mapped to a context.

The multicast <*dstid*> does not require any special handling or consideration within XTP. The only purpose of the multicast <*dstid*> is to pass through the hardware (or software) address filters that check incoming destination MAC addresses, which is the same purpose served by unicast destination addresses. Once a packet has passed through the MAC address filters, XTP doesn’t need to know whether the destination MAC address was a multicast address or a unicast address because context lookup is based on the source MAC address, or *srcid*.

Multiple senders on the same end system can send to the same multicast address and still be distinguished by receivers: because they share the same XTP subsystem, they will use different keys, and possibly different routes.

Multiple senders located on different end systems but sending to the same multicast address can also be disambiguated: their *srcid*’s will differ.

Multiple senders of XTP frames to the same group address over multicast-IP are similarly disambiguated: in this case the *srcid* used by XTP is the 32-bit source IP address in the IP packet.

Multiple senders to the same multicast MAC address that happen to have selected identical *key* and *route* values will be distinguished by their source MAC addresses as stated above. If these packets are allowed through an XTP switch, the switch logic (which assigns route values for new paths) would assign different route values based on *srcid*. In this case the forwarded packets would be disambiguated by the route fields even though the forwarded keys and source MAC addresses would be the same.

3.4.5 Multicast Context Management

A normal FIRST packet context setup is employed by the multicast sender. This means that the sender selects a key and route that are not otherwise in use and incorporates their values into a FIRST packet. The sender may then use any of the *options* bitflags to achieve the desired semantics. The sender can execute a `WCLOSE/RCLOSE` shutdown handshake with the receivers - although this is not required.

An application wishing to receive multicast packets sets up a waiting context that will accept the expected packets. The receiver waits for a multicast FIRST packet and proceeds in the same manner as unicast.

Some applications may not wish to wait for a FIRST packet, but may prefer to join an in-progress multicast association where the FIRST packet has already been processed. The PATH packet can be used in this circumstance, as defined below in section 3.4.7, to locate and join the in-progress association.

3.4.6 Multicast Flow Control

The multicast sender obeys the same rules for flow control and rate control as a unicast sender. To summarize: the sender begins with default or inherited values for allocation, rate control, and wait interval. As the sender adds `SREQ` to outgoing DATA or CNTL packets, new values for *alloc* and *rate* are returned from the receivers. The sender also periodically updates the round trip time estimate by observation using a control packet handshake. Thus the message exchanges between sender and receivers are identical to those between a sender and single receiver. However, generation and interpretation of the messages require special consideration in multicast mode.

If `NOERR` is set by the sender, output buffers can be released as soon as they have been transmitted. If error control is used, then output buffers must be retained until it is known that retransmission is not required. In unicast mode buffers can be released as soon as a received DSEQ indicates correct reception by the remote receiver. In multicast mode a received DSEQ only indicates that one receiver correctly received the data. The **bucket algorithm** defined in Appendix B provides a strategy for sending `SREQ`, interpreting control messages, and releasing output buffers.

When multicasting to a large set of receivers, it may be desirable to limit the number of duplicate or redundant control messages sent by the receivers. Two heuristics, **damping** and **slotting**, that are effective when used together are defined in Appendix B. These heuristics eliminate many of the redundant CNTL messages that would otherwise contribute to network overhead.

3.4.7 Multicast Error Control

The same error control procedures defined for point-to-point XTP are available in multicast mode. Three error control methods are possible: (1) no error control at all (using the `NOERR` bit), (2) go-back-n, and (3) selective retransmission. If NOERR is selected, the multicast receivers apply the standard algorithm: all sender `SREQ` requests are acknowledged with *rseq* and *dseq* values set to the highest sequence number. Otherwise a receiver manipulates *rseq*, *dseq*, *nspan* and *spans* according to the normal unicast rules and reports its state to the sender in control packets.

The multicast sender has the option of interpreting error control CNTL messages as go-back-n or as selective retransmission. The bucket algorithm described in Appendix B is defined as a go-back-n procedure. However, an implementation is not prohibited from defining a more sophisticated procedure based on selective acknowledgement and retransmission.

3.4.8 Joining an In-Progress Multicast Conversation

When a multicast receiver wants to join (or rejoin) a multicast conversation that is in-progress, the following procedure should be used. When a multicast LISTEN has been outstanding for some implementation specific time period without receiving a FIRST packet for the multicast address, the waiting receiver sends a PATH packet, (PATH, *address*= multicast address, *key*=0), indicating in the *address* field the connection it wants to join. Other multicast receivers must never respond to a PATH packet containing a *key* value of zero. The multicast sender responds to the PATH packet by sending a PATH packet containing the *key* for the multicast context and the usual values for all other fields. Upon receiving a PATH packet in which the *key* is not zero, the new receiver can begin receiving data at sequence number that is current for the conversation at that point in time, i.e. the first sequence number observed. That is, the rejoining receiver induces the multicast sender to generate a PATH packet that is then interpreted as if it were a FIRST packet.

3.4.9 Concentration

Concentration is defined as the reliable transmission of arbitrary messages from a set of hosts in a multicast group to a single host which is normally the sender to the group. If data from N hosts is to be concentrated at one host, then N contexts are required at the concentration host. No special facilities are defined in XTP for concentration; each concentration data stream is implemented as a new XTP association. Appendix B suggests an optional cloning technique that can improve the efficiency of creating new associations for concentration.

3.5   Timers

This section defines a counter, *retry_count*, and five timers used in XTP procedures: **rtt**, CTIMER, CTIMEOUT, WTIMER, and RTIMER.

3.5.1 rtt: round trip time

The **rtt** value is continuously maintained by ongoing control packet exchanges between Active contexts. It is calculated from *techo* whenever a received control packet contains a non-zero value for *techo*. When this happens, the received *techo* is subtracted from the current value for *time*; the difference is an estimate for **rtt**. The protocol implementation should use a smoothed value derived from **rtt** as defined in WTIMER.

3.5.2 CTIMER and CTIMEOUT: connection timers

Connection timers allow XTP to recover from system and network failure by measuring inactivity. The CTIMER is a long-duration clock timer used to generate keepalive messages. The maximum allowed value for CTIMER is 3600 seconds. The CTIMER procedure is as follows:

1. enable CTIMER when a context first becomes active.
2. count the number of packets received during each CTIMER interval. If at least one packet was received, do nothing.
3. if no packets were received during a CTIMER interval, then initiate a synchronizing handshake (see section 3.6).

CTIMEOUT is a separate timer, typically much shorter than CTIMER, which is used to bound the amount of time XTP will retry a message exchange. Another implementation parameter, *retry_count*, is used with CTIMEOUT to provide an upper bound on the number of times XTP will retry a message exchange. CTIMEOUT and retry_count are always used together. They are both enabled when the WTIMER (see below) expires while waiting for a response to an SREQ. If either CTIMEOUT or retry_count expires while XTP is retrying an SREQ message exchange, then XTP aborts the context.

All three of these variables - CTIMER, CTIMEOUT, and retrycount - should be settable by an application. Appendix D provides an implementation example.

3.5.3 WTIMER: wait timer

The WTIMER, wait interval or wait timer value, is the amount of time an XTP sender waits for a response to an `SREQ` before (re)transmitting `SREQ`. It is determined by applying the smoothing procedure suggested by Van Jacobsen to the round trip time values obtained from any XTP control handshake.

Define **rtt** to be the single round trip time measurement. Define WTIMER to be the wait timer interval which is a function of SRTT, the smoothed **rtt**, and RTTV, the **rtt** variance. For each **rtt** observation, compute the following:

```
SRTT = SRTT + 1/8(rtt - SRTT)
RTTV = RTTV + 1/4(abs(rtt-SRTT) - RTTV)
WTIMER = SRTT + 2*RTTV
```

WTIMER is used as an approximation to RTT to determine the expected arrival time of responses to SREQ.

3.5.4 RTIMER: rate control timer

The RTIMER, rate control or refresh timer, is used to control updating **credit** for use in rate control.

> ***Implementation Note***
> XTP implementations are not expected to implement RTIMER with resolution greater than the expression 0.1*MIPS where MIPS is the number of millions of instructions per second available to the network subsystem. This assumes that only 1% of the available MIPS should be dedicated to RTIMER processing and that RTIMER events complete within 1000 instructions. Thus if 10 MIPS are available for network processing, then it should be possible to operate 100 RTIMER events per second and use 1000 machine instructions per event.

3.6 Synchronizing Handshake

The term "synchronizing handshake" refers to an exchange of control packets between sender and receiver after which the sender is certain of the receiver's state. The synchronizing handshake is based on the rules for a sender and receiver regarding the *sync* and *echo* fields as stated in section 2.3.5, "Sync Field". The procedure is intended to eliminate spurious decisions that might be caused by network timing anomalies and packet loss. Retransmission of missing data indicated by *spans* fields may commence after a synchronizing handshake. The procedure is as follows:

1. enable CTIMEOUT and retry_count.
2. set an exponential backoff constant, k, to 1.
3. send a CNTL(SREQ) packet.
4. if any received CNTL packet contains a non-zero *techo* field and the *echo* field matches either the current value of *sync* or *saved_sync*, then the synchronizing handshake is complete.
5. if packets are received, but there are no matches for *sync* or *saved_sync*, then restart CTIMEOUT and retry_count, multiply the backoff constant K by two and go to step (2).
6. if no packets are received after K*WTIMER seconds, then decrement the retry_count, multiply the constant K by two and go to step (2).
7. if either retry_count or CTIMEOUT expire, abort the context.

The synchronizing handshake procedure is entered after a WTIMER failure. The objective is to probe with CNTL messages at exponentially increasing time intervals until there is a successful result or the CTIMEOUT/retry_count safeguards abort the context.

Appendix A: Checksum Function

The 32-bit XTP check function, called CXOR, is defined as the catenation of two 16-bit functions: XOR and RXOR. The CXOR result for a block of data consists of a 32-bit word containing the XOR of the block of data in the low-order 16 bits and the RXOR of the data in the upper 16 bits. XOR and RXOR are computed as defined below.

XOR:
1. the XOR result is initially set to zero.
2. each 16-bit short word of data is exclusive-or'd (XOR'd) into the result

An equivalent statement is: all the even-numbered bytes of data are XOR'd into the low byte of the result, and all the odd-numbered bytes are XOR'd into the next byte.

RXOR:
1. the RXOR result is initially set to zero.
2. for each successive 16-bit short word of data, rotate the result left by one bit, then XOR the data word into the result.

When assembling unordered buffers, it is desirable to compute the CXOR over a pair of buffers from the individual CXOR's of the two buffers. If the two buffers are named A and B, the combination of the buffers, A followed by B, is named C, and the CXOR components are labeled A.xor, A.rxor, B.xor, B.rxor, C.xor, and C.rxor, and len(B) is the number of 16-bit words in B, then the C.xor and C.rxor can be determined from the A and B values in the following manner:

APPEND:
1. C.xor is the XOR of A.xor and B.xor,
2. compute T as A.rxor rotated by len(B),
3. C.rxor is the XOR of T and B.xor.

In this method A.rxor is rotated by the amount it would have been rotated if the two buffers were joined and C.xor had been calculated directly.

A new CXOR of a buffer, B, caused by replacing one 16-bit word, W, located REM(B) 16-bit words from the end of B by a new value, NW, may be determined as follows:

UPDATE:
1. compute D, the difference, as the XOR of N with NW.
2. XOR D into B.xor.
3. compute T as D rotated by REM(B).
4. XOR T into B.rxor.

High-performance software implementations of CXOR can be constructed by using the following techniques:
1. unrolling loops,
2. calculating XOR's on words that would have the same RXOR rotation and saving the rotate operations until the end,
3. calculating XOR's over 32 or 64-bit words instead of 16-bit words and "folding" the results down to 16 bits.

Although the frame design of XTP ensures that checksum boundaries will be on multiples of 4 bytes, some operating systems may break packets into small pieces that contain odd numbers of bytes. A CXOR function that must operate in this environment should incorporate a test at both the beginning and end of processing a buffer to accommodate an extra byte.

CXOR, the catenation of XOR and RXOR is a form of parity function over a block of data. XOR is often called the "vertical" parity function. RXOR forms the parity function of 16 spirals through the data as depicted in Figure A-1.

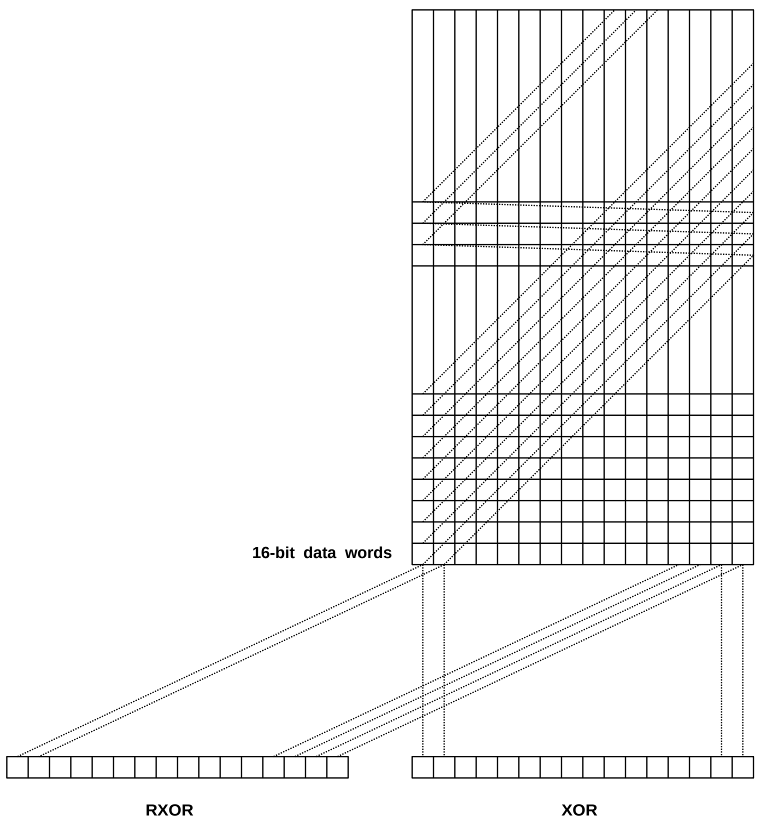

Figure A-1. CXOR bit pattern.

The failure modes of CXOR may be described simply. A failure is defined when the CXOR of a buffer containing bit errors produces the check value for the non-error buffer.

Since CXOR is a parity-based function, it will detect all errors caused by an odd number of error bits in an arbitrary length buffer. CXOR will fail to detect some error patterns consisting of an even number of error bits. This can only happen when both the XOR and RXOR functions fail.

In order for both RXOR and XOR to fail, the pattern of error bits must be at intersection points between parity spirals and parity columns such that there is an even number of error bits in each parity column and an even number of error bits in each spiral.

Appendix B: Multicast

A number of heuristics have been studied to augment the basic multicast procedure of Section 3.4. These are optional procedures described here as an implementation aid.

1. B.1 Bucket Algorithm

Control messages received in unicast mode are acted upon without delay: *dseq* is used to release output buffers, *rseq* is used to start go-back-n retransmission, and *alloc* is used to advance the output window. In multicast mode some additional processing and delay must be inserted between the arrival of control messages and when action is taken. The purpose of the delay is to allow a reasonable time period for a multiplicity of receivers to respond to the `SREQ` or to report errors. The *bucket algorithm* described below manages this time delay.

A **bucket** is defined whenever the sender transmits a control packet with `SREQ`. The bucket refers to all the data packets, possibly none, transmitted since the end of the previous bucket. This includes both transmissions and retransmissions. Each bucket is identified by the sender's *sync* value which is computed by incrementing **sync** by 1 for every transmitted data packet.

When control messages are received from receivers they can be related to a particular bucket by their *echo* values, since *echo* is the returned value for *sync*. Responses are accumulated for each bucket as follows:

1. minimum of received *alloc*,
2. minimum of received *dseq*,
3. minimum of received *rseq*,
4. maximum of computed **rtt**.

If there are B buckets, then after a certain time period the accumulated *alloc*, *dseq*, *rseq*, and **rtt** values for the oldest bucket are acted upon and the oldest bucket becomes the newest bucket. The details of how this is done are described below.

The algorithm is parameterized with respect to the following four variables:

1. WT (same as WTIMER defined in Section 3.6), the current wait timer interval,
2. N, an implementation-selected value,
3. ST, switching time, determined as WT/(N+.5),
4. E, the desired multicast error tolerance.

N, which must be 1 or greater, determines the number of `SREQ` control messages that will be transmitted within a WT time interval. E determines the error tolerance, which is defined as the number of consecutive lost or damaged error control packets from a receiver to a multicast sender that will be tolerated. A small value for E reduces message traffic, but increases the total amount of buffer space needed to run a multicast stream continuously (i.e., without falling into stop-and-wait behavior). If no more than E packets are lost, then the retransmission algorithm will recover by retransmitting the lost data. If E is exceeded, then the sender's ability to retransmit has been exceeded and the receiver will never be able to recapture the missing data. Receiver's must have methods for "dropping out" of multicast conversations when this occurs.

If a receiver drops out, it can indicate this failure to the application. This allows the application to indicate whether or not to execute the multicast rejoin procedure in order to reconnect to the group.

The bucket algorithm details can now be stated:

1. there are a total of N+E+1 buckets for an active multicast output context.
2. a control `SREQ` message is transmitted every ST seconds by the multicast sender. This is done even if no data is transmitted.
3. a new bucket is created for each `SREQ`.
4. receivers multicast control responses to `SREQ`.
5. incoming control messages at the multicast sender are associated with a bucket by matching returned *echo* value with the *sync* value on each bucket.
6. The bucket variables, *alloc, dseq, rseq, and rtt* are updated from the control packet if the new values are less than *alloc*, *dseq*, and *rseq* or greater than **rtt**.
7. after N+E+1 buckets have been created, each new ST interval will relabel the "oldest" bucket as the "newest" bucket and the remaining buckets "age" one ST time unit. When this happens, the accumulated bucket variables are acted upon by the multicast transmitter. The values that are used are minimum *dseq*. *rseq*, and *alloc*, over all buckets, and maximum **rtt** over all buckets.

The total amount of buffer space needed to operate the bucket algorithm continuously is

Space = (Number of buckets) * ST * B

where ST is the switch time defined earlier and B is the network bandwidth. This can be transformed as follows:

Space = (N+E+1) * (WT/(N+.5)) * B
= K * WT * B

where K is a multiplier (N+E+1)/(N+.5), and WT is the current wait timer (derived from rtt). For increasing values of N, the multiplier K approaches 1 which gives the minimum buffer space with the maximum number of buckets. If the number of buckets is reduced by decreasing N to 1, then K becomes (E+2)/1.5 which leads to maximum buffer requirement. At both extremes for K the error tolerance, E, is the same.

2. Damping

Since receivers multicast their control messages, all receivers can observe the control messages of their peers. In an error-free multicast group, a control message sent by a multicast receiver is expected to be duplicated by all the other receivers in response to an `SREQ`. In case of errors, receivers experiencing similar errors would send similar error control messages. In either case, some percentage of control messages will be duplicates. The percentage is presumed to be higher for low error rate networks, although receivers observing consistently identical errors would keep the percentage high.

With damping, a receiver does not send a control message if it would duplicate a message already observed on the network. This technique reduces the total number of control messages transmitted by the number of duplicates that are cancelled. It does this at the expense of having receivers read all control messages. When the set of receivers is "large", say greater than 10, this is a worthwhile compromise, especially if a slotting techniques (see below) is used to increase the probability of effective damping.

3. Slotting

The slotting technique imposes a random delay before sending `SREQ`-induced control messages. There are many techniques for determining the slot times and for selecting a transmission slot. For example, the low-order bits of the local MAC address can be used to select a slot. The exact method is implementation-dependent.

The results of slotting are twofold. First, the number of control messages is distributed in time rather than concentrated in time. This enhances performance because it avoids an instantaneous increase in network offered load. Second, the distribution of messages in time increases the probability that an implementation will be able to exercise damping.

Implementation experiments show that damping or slotting alone are not effective, but when used together can reduce the total control message traffic. This is viewed as a necessary optimization for applications that may include a large number, say hundreds, of multicast receivers.

4. Cloning

If a large number of concentration channels are needed, there may be an advantage to creating the additional contexts automatically instead of using explicit association setup procedures. The simplest method is to implement a persistent LISTEN operation that "clones" a sequence of active contexts in response to incoming FIRST (or DATA) packets.

Appendix C: Close State Machine

A detailed specification for the XTP close procedure is presented below. The specification consists of two state machines: one for the input data stream and one for the output data stream. Transitions are specified that take each machine from the ACTIVE state, through intermediate states, to an INACTIVE state. When both machines are in the INACTIVE state the context as a whole is considered inactive and both machines transition to the NULL state which is representative of the context being released.

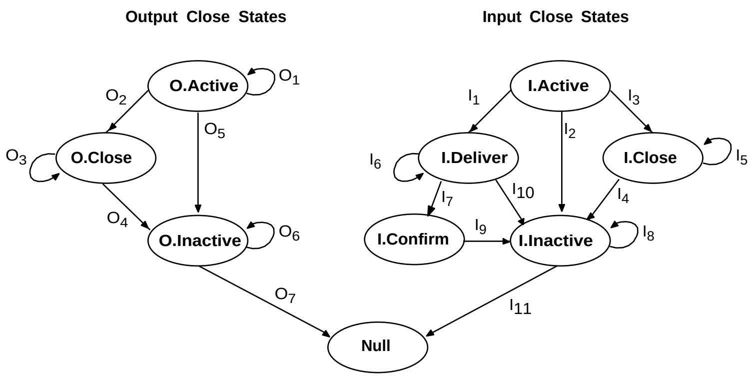

Figure C-1. Close State Machines.

When an association is established, both state machines normally enter the ACTIVE state, but this is not required. It is possible for XTP to operate in a unidirectional manner with one data stream active and the other closed. Multicast mode provides an example of this asymmetric behavior. No special attempts are made to model asymmetric situations because the independence of the two machines makes it unnecessary.

The state transitions are defined by a number of predicates defined over a set of inputs. The inputs are listed in Table C-1. The input bits model common activities in a protocol subsystem. The RCV bits are set when the RCLOSE, WCLOSE, or END bits are received in an XTP packet. The LOCAL events are caused by the local service interface or operating system. The SET flags cause the indicated bits to be sent in output packets. The TIMER event happens after the START_TIMER command (see below) has been called and the timer interval has expired. RQEMPTY and XQEMPTY reflect the state of the input and output data queues, respectively. The I_INACTIVE and O_INACTIVE represent whether the state of the input or output machine is INACTIVE. The two MULTI bits are used to guide the state machines around message exchanges that are not permitted in multicast. MULTISEND is set if the application is a multicast sender. MULTIRECV is set if the application is a multicast receiver.

Input Bits	
RCV_RCLOSE	received a packet with the RCLOSE bit set
RCV_WCLOSE	received a packet with the WCLOSE bit set
RCV_END	received a packet with the END bit set
LOCAL_RCLOSE	local application closes input data stream
LOCAL_WCLOSE	local application closes output data stream
LOCAL_END	application or XTP abort terminates context
SET_RCLOSE	set RCLOSE bit on output packets
SET_WCLOSE	set WCLOSE bit on output packets
SET_END	set END bit on output packets
TIMER	local wait timer event
RQEMPTY	local receive queue becomes empty
XQEMPTY	local output queue has been transmitted
O_INACTIVE	output state becomes or is INACTIVE
I_INACTIVE	input state becomes or is INACTIVE
MULTISEND	application is a multicast sender
MULTIRECV	application is a multicast receiver
CONFIRM_MODE	application wants to confirm RCLOSE
CONFIRM	application confirms RCLOSE

Table C-1. Close Machine Inputs.

The input bits are edge-triggered and "sticky". That is, they cause an input event when their values change, cycling the state machines. Also, they stay set unless cleared and may be examined at any time as variables in predicates. For example, after the RCV_END event has happened, the RCV_END bit will be set. Consequently the predicate (RQEMPTY & !RCV_END) means "if RQ is EMPTY and END has not been receiving." Both state machines can cause the following actions:

Actions	
START_TIMER	enable timer event
DISCARD_RQ	release receive queue (RQ)
DISCARD_XQ	release transmit queue (XQ)
FLUSH_RQ	deliver RQ to user, discard out of sequence data
DELIVER_RQ	deliver RQ to user, wait for retransmissions if needed
SEND_DATA	transmit the output queue as normal
SEND_CNTL_SREQ	send a control packet with SREQ
SET_RCLOSE	set RCLOSE in outgoing packets
SET_WCLOSE	set WCLOSE in outgoing packets
SET_END	set END in outgoing packets

Table C-2. Close Machine Actions.

Input Close Transitions			
transition	**event**	**predicate**	**action**
I1	RCV_WCLOSE | RCV_END		
I2	LOCAL_END	!MULTIRECV	SET_END DISCARD_RQ
I2	LOCAL_END	MULTIRECV	DISCARD_RQ
I2	LOCAL_RCLOSE	MULTIRECV	SET_RCLOSE
I3	LOCAL_RCLOSE	!MULTIRECV	SET_RCLOSE START_TIMER DISCARD_RQ
I4	RCV_WCLOSE | RCV_END		
I5	TIMER		SEND_CNTL_SREQ START_TIMER
I6	!RQEMPTY	!RCV_END	DELIVER_RQ
I6	!RQEMPTY	RCV_END	FLUSH_RQ
I7	RQEMPTY&CONFIRM_MODE		
I8	!O.Inactive		
I9	CONFIRM		SET_RCLOSE
I10	RQEMPTY&!CONFIRM_MODE		
I10	LOCAL_END		DISCARD_RQ
I10	RCV_END		FLUSH_RQ
I11	O.Inactive		

Table C-3. Input Close Transitions.

The transition tables are read from left to right. For example transition I2 is triggered if there is a LOCAL_END event when the state machine is in the state that sources transition I2. Actions taken after the LOCAL_END event depend on the MULTIRECV predicate. If !MULTIRCV then the SET_END and DISCARD_RQ actions are taken. If MULTIRCV then the DISCARD_RQ action is taken. In both cases the next state is indicated by the transition graph in Figure E-1 to be the I.Inactive state.

Output Close Transitions			
transition	**event**	**predicate**	**action**
O1	**LOCAL_WCLOSE**	**!XQEMPTY**	**SEND_DATA**
O2	**LOCAL_WCLOSE**	**XQEMPTY**	**SET_WCLOSE** **START_TIMER**
O3	**TIMER**		**SEND_CNTL_SREQ** **START_TIMER**
O4	**RCV_RCLOSE | RCV_END**		**DISCARD_XQ**
O4	**LOCAL_END**		**DISCARD_XQ** **SET_END**
O5	**RCV_RCLOSE**	**!MULTISEND** **& !RCV_END**	**SET_WCLOSE** **DISCARD_XQ**
O5	**RCV_END**	**!MULTISEND**	**DISCARD_XQ**
O5	**LOCAL_END**		**SET_END** **DISCARD_XQ**
O6	**!I.Inactive**		**SET_END**
O7	**I.Inactive**		

Table C-4. Output Close Transitions.

Appendix D: Application Programming Interface

A software interface for controlling services, modes, and options is presented here. The design is taken from an XTP implementation. It provides an example of how the disparate aspects of a service interface can be unified. It also provides an informal service definition for XTP, omitting only read and write i/o primitives and context create and delete primitives, which usually provided by the operating system environment, and mechanisms for accessing the EOM and BTAG bits.

This application programming interface (API) defines a small number of transactions that all use a data structure defined below as *typedef xtp_service*. The address of an instance of this structure is to be passed to XTP along with an integer command that identifies the transaction and an integer that identifies the context. XTP executes the command, which may require reading from the data structure, and always copies status information into the structure before returning to the application. The system call interface (in C) might appear as follows:

r = xtp_entry(x, command, &xtp_service);

where *r* is a success/failure return code, *x* identifies the context, *command* selects the transaction, and *xtp_service* provides the location of the communication structure.

The transactions are described below.

1. LISTEN: listen for a FIRST packet that matches the specifications contained in the xtp_service structure. Wait (sleep) until the listening context becomes active.
2. LISTEN_I: establish a listening context as in (1), but return immediately without waiting.
3. CONNECT: connect to another end system according to the parameters expressed in xtp_service; wait until the context is active or until the attempt fails before returning.
4. CONNECT_I: Connect as in (3), but do not wait for completion.
5. WAIT: Wait for a Listen or Connect operation to complete.
6. GET_PARMS: Read the current state of the context into the xtp_service structure.
7. PUT_PARMS: Write the contents of the xtp_service structure into XTP internal state variables and return the current state of the context in the same structure.
8. CONFIRM_CLOSE: Confirm a read close operation to XTP.
9. ACCEPT_FIRST: Accept the FIRST packet received by a pending LISTEN and make the context active.
10. REJECT_FIRST: Reject the FIRST packet received by a pending LISTEN, send a DIAG packet as indicated in the xtp_service structure, and continue listening.

The details of each transaction are defined by the contents of xtp_service which is defined below.

```
typedef struct xtp_service {
        u_long          mode_flags;             /* Specifies XTP options flags */
        u_long          sort_value;             /* sort value for context */
        u_long          listen_flags;           /* specifies LISTEN semantics */
        u_long          field_flags;            /* enables writing the remaining fields  */
        char            service;                /* Service type requested/provided */
        u_int           rbufsize;               /* read buffer size */
        u_int           wbufsize;               /* write buffer size */
        u_int           maxdata;                /* maximum output segment size */
        u_int           rwindow;                /* default allocation permitted by receiver */
        u_int           retry_count;            /* max number of handshake attempts */
        long            input_burst;            /* input burst size */
        long            input_rate;             /* input rate */
        long            output_burst;           /* output burst size */
        long            output_rate;            /* output rate */
        u_int           ctimeout;               /* ctimeout value */
        u_int           ctimer;                 /* ctimer value */
        u_int           wtimer;                 /* wtimer value */
        xtp_addr address_segment;               /* connect,listen,or FIRST addr segment */
        u_short         mc_drops;               /* number of multicast drops (bucket algorithm) */
        u_short         mc_sreqs;               /* number of multicast sreqs (bucket algorithm)*/
        u_short         mc_rttgain;             /* multicast RTT gain */
        u_short         mc_rttvgain;            /* multicast RTT var gain */
        u_long          code;                   /* DIAG code */
        u_long          value;                  /* DIAG value */
} xtp_service;

/*
 * The following are defined for use with mode_flags.
 * The indicated modes may be set with a PUT_PARMS command.
 * Current modes are returned by all commands.
 */

#define  XMODE_SORT             0x1
#define  XMODE_FASTNAK          0x2
#define  XMODE_NOFLOW           0x4
#define  XMODE_RES              0x8
#define  XMODE_NOERR                            0x10
#define  XMODE_NOCHECK          0x20
#define  XMODE_MULTICAST        0x40
#define  XMODE_CLOSE_CONFIRM                    0x100   /* Don't RCLOSE automatically */
```

```
/* The following flags identify which fields in the struct are being set */

#define  XFIELD_SERVICE          0x1
#define  XFIELD_RBUFSIZE         0x2
#define  XFIELD_WBUFSIZE         0x4
#define  XFIELD_MAXDATA          0x8
#define  XFIELD_RWINDOW          0x10
#define  XFIELD_RETRY_COUNT      0x20
#define  XFIELD_INPUT_BURST      0x40
#define  XFIELD_INPUT_RATE       0x80
#define  XFIELD_OUTPUT_BURST     0x100
#define  XFIELD_OUTPUT_RATE      0x200
#define  XFIELD_CTIMEOUT         0x400
#define  XFIELD_CTIMER           0x800
#define  XFIELD_WTIMER           0x1000
#define  XFIELD_PEER_ADDR        0x2000
#define  XFIELD_MC_DROPS         0x4000
#define  XFIELD_MC_SREQS         0x8000
#define  XFIELD_MC_RTTGAIN       0x10000
#define  XFIELD_MC_RTTVGAIN      0x20000
#define  XFIELD_CODE             0x40000
#define  XFIELD_VAL              0x80000

/* The following flags specify listen parameters */

#define XLISTEN_FASTNAK     0x1     /* Accept only if FASTNAK option matches */
#define XLISTEN_NOFLOW      0x2     /* Accept only if NOFLOW option matches */
#define XLISTEN_RES         0x4     /* Accept only if RES option matches */
#define XLISTEN_NOERR       0x8     /* Accept only if NOERR option matches */
#define XLISTEN_NOCHECK     0x10    /* Accept only if NOCHECK option matches */
#define XLISTEN_CONFIRM     0x20    /* Permit the application to confirm connection
                                       acceptance ("manually" accept connections) */

#define  XSERVICE_CONNECT   1       /* Connection service */
#define  XSERVICE_TRANS     2       /* Transaction service */
#define  XSERVICE_UNACKDG   3       /* Unacknowledged datagram service */
#define  XSERVICE_ACKDG     4       /* Acknowledged datagram service */
#define  XSERVICE_ISOCHR    5       /* Isochronous stream service */
#define  XSERVICE_BULKD     6       /* Bulk data service */
```

Appendix E: Packet Structure Summary

Packet syntax and bit definitions are reproduced here for reference.

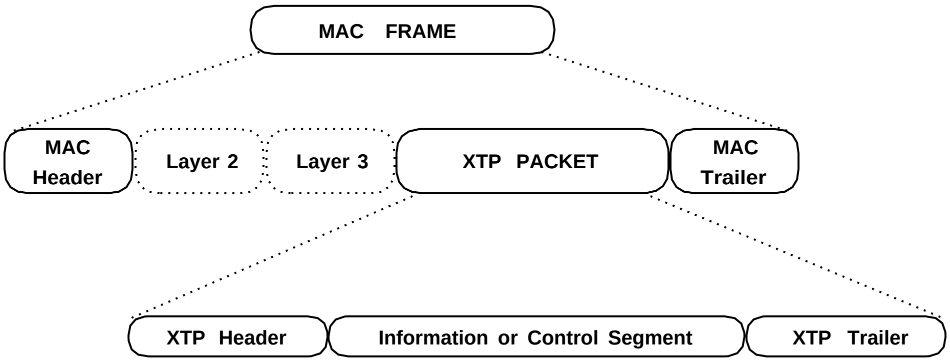

Figure E-1. XTP packet within MAC frame.

The matrix portrayed in Table E-1 summarizes the format required for the middle segment of each type of XTP packet (as defined in the *pformat* field of the XTP Header):

Packet Type	Information Segment	Control Segment	Undefined
DATA	**X**		
CNTL		**X**	
FIRST	**X**		
PATH	**X**		
DIAG	**X**		
MAINT			**X**
MGMT			**X**
ROUTE	**X**		
RCNTL		**X**	

Table E-1. Middle Segments by Packet Types.

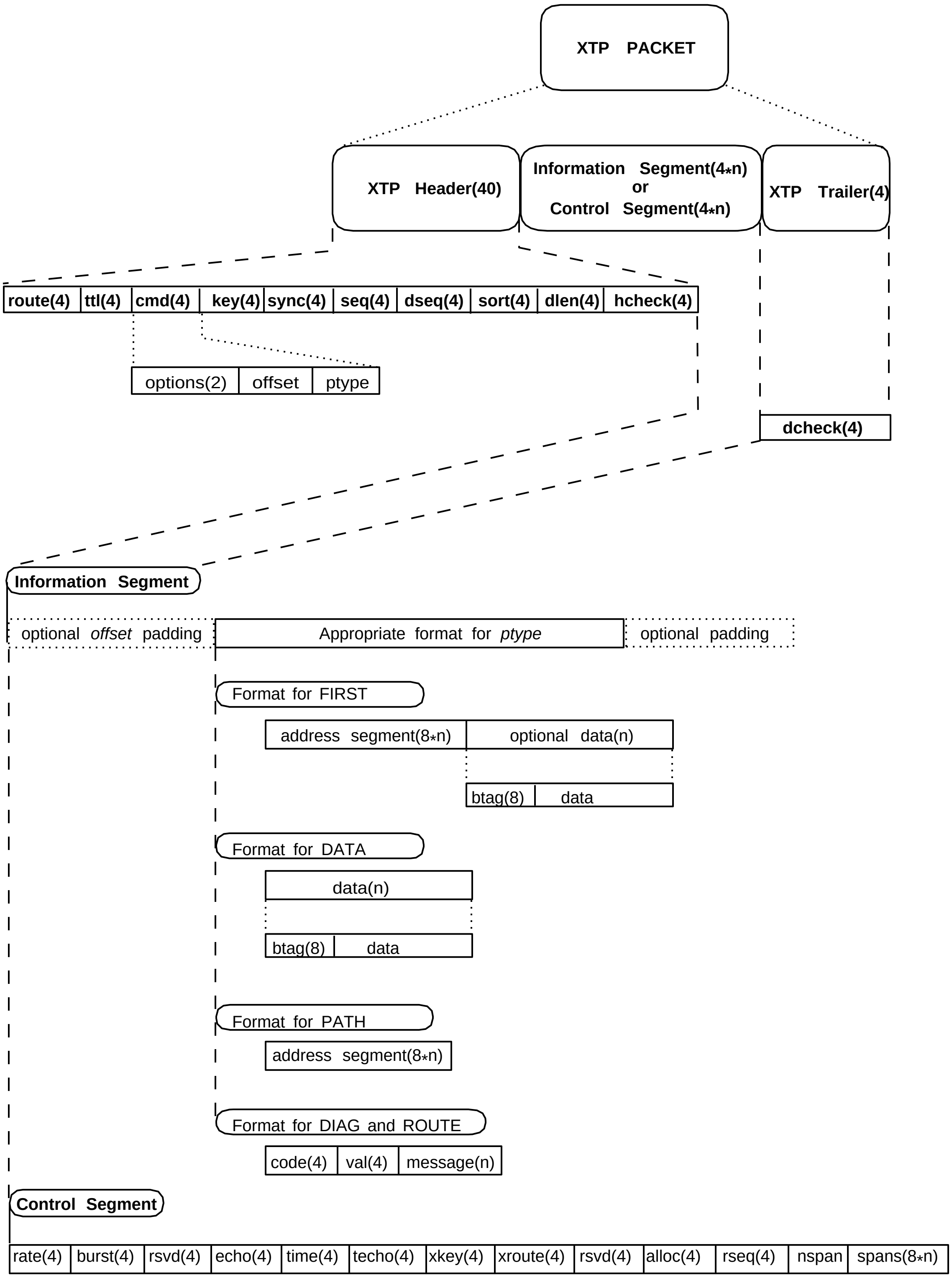

Figure E-2. Syntax summary.

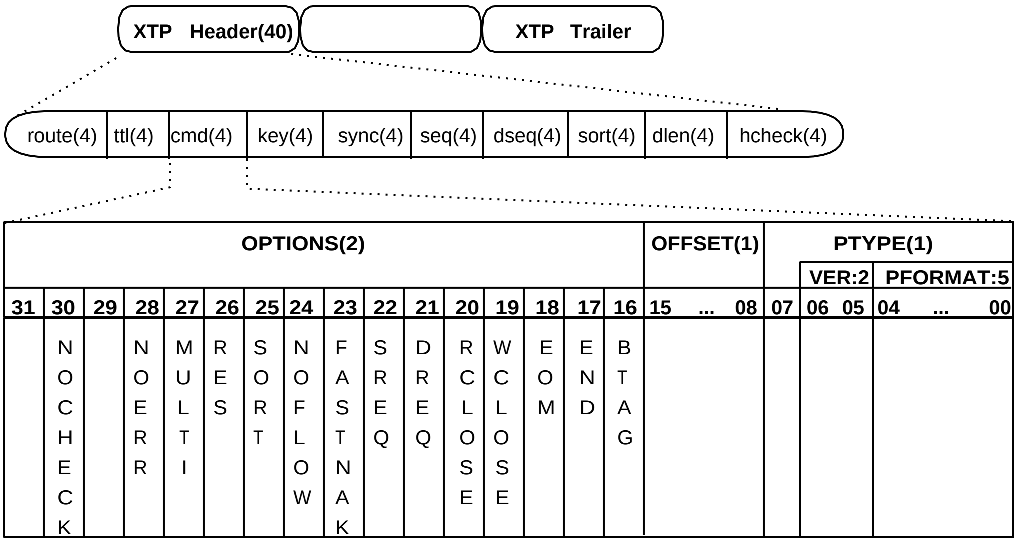

Figure E-3. `Cmd` field of Header.

Summary of *pformat* values for the *cmd* field of the header:

Pformat	*Decimal Value*	*Hex Value*	*Definition*
DATA	0	0x00	Packet containing user data
CNTL	1	0x01	Control packet communicating status information
FIRST	2	0x02	The first packet sent for an association
PATH	6	0x06	Packet for threading a path or joining an in-progress conversation
DIAG	8	0x08	Diagnostic packet
MAINT	10	0x0a	Packet for communicating network maintenance info
MGMT	14	0x0e	Packet for communicating management information
ROUTE	18	0x12	Packet for communicating with an XTP switch
RCNTL	19	0x13	Switch-generated control packet

Table E-2. *Pformat* **summary.**

Summary of header *options* field bit declarations:

BIT	*Mask*	*Description*	*Potential Change*	*Expectation*
	0x8000	not used	MBZ	
`NOCHECK`	0x4000	Disable checksum function	Per packet	Per context
	0x2000	not used	MBZ	
`NOERR`	0x1000	Disable error control	Per packet	Per context
`MULTI`	0x0800	Multicast mode	Per association	
`RES`	0x0400	Reservation mode	Per packet	Per context
`SORT`	0x0200	Enable sorting	Per packet	Per context
`NOFLOW`	0x0100	Disable flow control	Per context	
`FASTNAK`	0x0080	Enable aggressive error control	Per packet	Per context
`SREQ`	0x0040	Status requested	Per packet	
`DREQ`	0x0020	Delivery status requested	Per packet	
`RCLOSE`	0x0010	Reader closed	Per packet	
`WCLOSE`	0x0008	Writer closing	Per packet	
`EOM`	0x0004	End of message	Per packet	
`END`	0x0002	End of association	Once	
`BTAG`	0x0001	Beginning data tag	Per packet	

Table E-3. Header `options` bits.

Table E-4 indicates which fields in the Header are meaningful for each *type* of XTP packet.

Y = yes; item is used and has meaning with this type of packet
N = no; item never has any meaning with this type of packet
E = item has meaning if enabled

<table>
<tr><th></th><th>DATA</th><th>CNTL</th><th>FIRST</th><th>PATH</th><th>DIAG</th><th>MAINT</th><th>MGMT</th><th>ROUTE</th><th>RCNTL</th></tr>
<tr><td>Route</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td colspan="2" rowspan="10">undefined</td><td>Y</td><td>Y</td></tr>
<tr><td>Ttl</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr>
<tr><td>Cmd</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr>
<tr><td>Key</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>N</td><td>N</td></tr>
<tr><td>Sync</td><td>Y</td><td>Y</td><td>Y</td><td>N</td><td>N</td><td>N</td><td>N</td></tr>
<tr><td>Seq</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>N</td><td>N</td><td>N</td></tr>
<tr><td>Dseq</td><td>Y</td><td>Y</td><td>Y</td><td>N</td><td>N</td><td>N</td><td>N</td></tr>
<tr><td>Sort</td><td>E</td><td>E</td><td>E</td><td>E</td><td>N</td><td>N</td><td>N</td></tr>
<tr><td>Dlen</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr>
<tr><td>Hcheck</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td><td>Y</td></tr>
</table>

Table E-4. Header Fields.

Table E-5 indicates which bits in the *cmd* word are meaningful for each *type* of XTP packet.

Y = yes; item is used and has meaning with this type of packet
N = no; item never has any meaning with this type of packet

Command Options:	DATA	CNTL	FIRST	PATH	DIAG	MAINT	MGMT	ROUTE	RCNTL
NOCHECK	Y	Y	Y	Y	Y			Y	Y
NOERR	Y	N	Y	N	Y	u		N	N
MULTI	Y	Y	Y	Y	Y	n		Y	N
RES	Y	Y	Y	N	N	d		N	N
SORT	Y	Y	Y	Y	N	e		N	N
NOFLOW	Y	Y	Y	Y	Y	f		Y	Y
FASTNAK	Y	Y	Y	Y	N	i		N	N
SREQ	Y	Y	Y	Y	N	n		N	N
DREQ	Y	Y	Y	Y	N	e		N	N
RCLOSE	Y	Y	Y	Y	Y	d		N	N
WCLOSE	Y	Y	Y	Y	Y			N	N
EOM	Y	N	Y	N	N			N	N
END	Y	Y	Y	Y	Y			N	N
BTAG	Y	N	Y	N	N			N	N

Table E-5. Cmd Option Bits.

Table E-6 summarizes the conditions under which each Control Segment field within a CNTL packet is interpreted and the existence of any XTP-defined values for the field.

This table does not cover RCNTL packets; when the packet *type* is RCNTL, only *rate*, *burst*, and *xroute* are interpreted as described below while all other fields are DC.

	Interpreted by receiver	Comments about Conditional Interpretation	XTP-defined value
rate	Conditional	when *rate* != 0xffffffff	Zero means "stop sending DATA packets." 0xffffffff means "unconstrained."
burst	Conditional	when *rate* != 0xffffffff	Zero means "stop sending; *burst*=0xffffffff means unconstrained."
echo	Conditional	when *techo!=0*	Contains returned *sync* value.
time	Always		
techo	Always		When field does not contain a copy of a received *time* =MBZ.
xkey	Conditional	Not interpreted when `MULTI` is set.	When field contains sender's chosen *key*, most significant bit of *key* value must be set. When field does not contain sender's own primed *key* =MBZ. After successful completion of *key* exchange =MBZ.
xroute	Conditional	Not interpreted when `MULTI` is set.	When field contains sender's chosen *route*, most significant bit of *route* value must be set. When field does not contain sender's own primed *route* =MBZ. After successful completion of *route* exchange =MBZ.
rsvd	Never		MBZ
alloc	Conditional	Not interpreted in NOFLOW mode.	specifies flow control window.
rseq	Always		When `NOERR` bit is set =highest sequence number seen on data stream. When no data has been received =sequence number for first byte of data stream.
nspan	Conditional	ignored when sender is in NOFLOW mode. Interpreted only on initial error or after synchronizing handshake.	
spans	Conditional	Only interpreted when *nspan* > 0. Interpretation is optional at all times. Truncation with partial interpretation does not violate protocol.	When *nspan* is 0 =DC. Always must be multiple of 8 bytes.

Table E-6. Interpretation and XTP-defined values for Control Segment in CNTL packets.

Key for table:

DC = don't care; may contain any value.

MBZ = must be zero.

Table of Contents