XTP 95-20

Xpress Transport Protocol Specification

XTP Revision 4.0

March 1, 1995

XTP Forum
1394 Greenworth Place
Santa Barbara, CA 93108
USA

Xpress Transport Protocol, Revision 4.0

This document is available in printed form at a cost of $40 USD, Air Mailed internationally, from:

XTP Forum
1394 Greenworth Place
Santa Barbara, CA 93108
USA

FAX: (805) 565-0699
Email: info@xtp.com

Disclaimer

This specification represents a consensus of the broad expertise within XTP Forum, as well as those experts outside of XTP Forum that have participated. The use of this specification is wholly voluntary. XTP Forum disclaims any liability arising from the use of this specification, with no warranty expressed or implied. This specification is stable, but is subject to change without notification.

Acknowledgements

Editor-in-Chief
Tim Strayer, Sandia National Laboratories

Editors
John Fenton, Mentat
Bruce Carneal, Mentat
Bill Atwood, Concordia University
Alf Weaver, University of Virginia and Network Xpress
Robert Simoncic, University of Virginia and Network Xpress
Bert Dempsey, University of Virginia
James McNabb, Network Xpress
Greg Chesson, Silicon Graphics, Inc.
Phil Irey, Naval Surface Warfare Center, Dahlgren Division
David Marlow, Naval Surface Warfare Center, Dahlgren Division
Merle Neer, NRaD
Bernard Metzler, Technical University of Berlin
Klaus Rebensburg, Technical University of Berlin
Chase Bailey, Efficient Networks
Larry Green, XTP Forum

This document has been made possible by the support of the **XTP Forum Members and Research Affiliates**.

Membership includes:

Apple Computer
Efficient Networks
GTE Laboratories
Hughes Information Technology Company
Interphase
Linotype-Hell AG
Lockheed
Mentat
Network Xpress
Northrop
Philips Research
SBE
Silicon Graphics, Inc.
Suri Giken Co. Ltd.
UNISYS

Research Affiliates include:

Bowman Gray School of Medicine
Concordia University
Defense Research Agency
Ecole Nationale Superieure des Telecommunications
Electronics & Telecommunications Research Institute (ETRI)
Georgia State University
GMD-FOKUS
Korean Advanced Institute of Science & Technology
Microelectronics Center of North Carolina (MCNC)
National Hsing Hua University
Naval Surface Warfare Center (NSWC), Dahlgren Division
Naval Command Control and Surveillance Center (NCCOSC) RDT&E Division (NRaD)
Olivetti Research, Ltd.
Sandia National Laboratories
Southeast University, People's Republic of China
Technical University of Aachen
Technical University of Berlin
Universidad Politecnica de Madrid
University of Karlsruhe
University of Stuttgart
University of Virginia
University of the Witwatersrand

Table Of Contents

List of Figures

5 Multicast Functional Specification

6 Encapsulation

Appendix B Additional Traffic Specifier Formats

Appendix D Service Profile Definitions

Appendix E Multicast Extensions

List of Tables

1 Introduction

Computer communication protocols, especially those “in the middle of the stack,” are driven by emerging advances in physical signalling and user applications. ATM is joining FDDI and Ethernet as dominant LAN technologies. Switched networks are making inroads where routers were prevalent. Clusters of workstations are being treated as multicomputers, and traditional host-to-host networks are looking more like MPP interconnection networks. Multimedia and telepresence are blurring the distinction between television, telephony, and computing. Telecommuting is a reality.

Each time the networking infrastructure changes, the appropriateness of the networking protocols is called again into question. Yet TCP/IP has been remarkably successful through more than twenty years of Internet growth and change. TCP’s success, however, has not detoured researchers from critical examination of how TCP provides transport services, and what services TCP fails to provide. Delta-t (for connection management), NetBLT (for bulk data transfer), VMTP (for transactions), and others are the result of identifying and fixing some deficiency in TCP. The Xpress Transport Protocol joins this list with contributions in: orthogonal protocol functions for separating paradigm from policy, separation of rate and flow control, explicit first-class support for reliable multicast, and data delivery service independence.

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

Separation of rate and flow control. Flow control operates on end-to-end buffer space. Rate control is a producer/consumer concept that considers processor speed and congestion. TCP does not provide rate control, and combats congestion with slow-start and other heuristics. XTP provides mechanisms for shaping rate control and flow control independently.

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

Data delivery service independence. XTP is a transport protocol, yet with the advent of switched networks rather than routed internetworks, a traditional network layer service may not be appropriate in every instance. XTP requires only that the underlying data delivery service provides framing and delivery of packets from one XTP-equipped host to another. This could be raw MAC, IP, AAL5, or something else. XTP also employs parametric addressing, allowing packets to be addressed with any one of several standard addressing formats.

Other features of XTP include:

- implicit fast connection setup for virtual circuit paradigm
- key-based addressing lookups
- message priority and scheduling
- support for encapsulated and convergence protocols
- selective retransmission and acknowledgement
- fixed-size 64-bit aligned frame design
- 64-bit sequence and connection identifiers
- parameterized traffic and quality of service negotiation

1.1 Protocol Concepts

XTP defines the mechanisms necessary for delivering user data from one end-system to one or more other end-systems. Well-defined packet structures, containing user data or control information, are exchanged in order to effect the user data transfer. The control information is used to provide the requested level of correctness and to assist in making the transfer efficient. Assurance of correctness is done via error control algorithms and maintenance of a connection state machine. Flow and rate control algorithms, certain protocol modes, and traffic shaping information are used to provide the requested quality of service as efficiently as possible.

The collection of information comprising the XTP state at an end-system is called a *context*. 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. There may be many active contexts at an end-system, one for each active conversation.

Each context manages both an outgoing data stream and an incoming data stream. A data stream is an arbitrary length string of sequenced bytes, where each byte is represented by a sequence number. The aggregate of active contexts and the data streams between them is called an *association*. The XTP communication model is shown in Figure 1-1 (for simplicity, this figure shows a unicast association; a multicast association is similar with more endpoints).

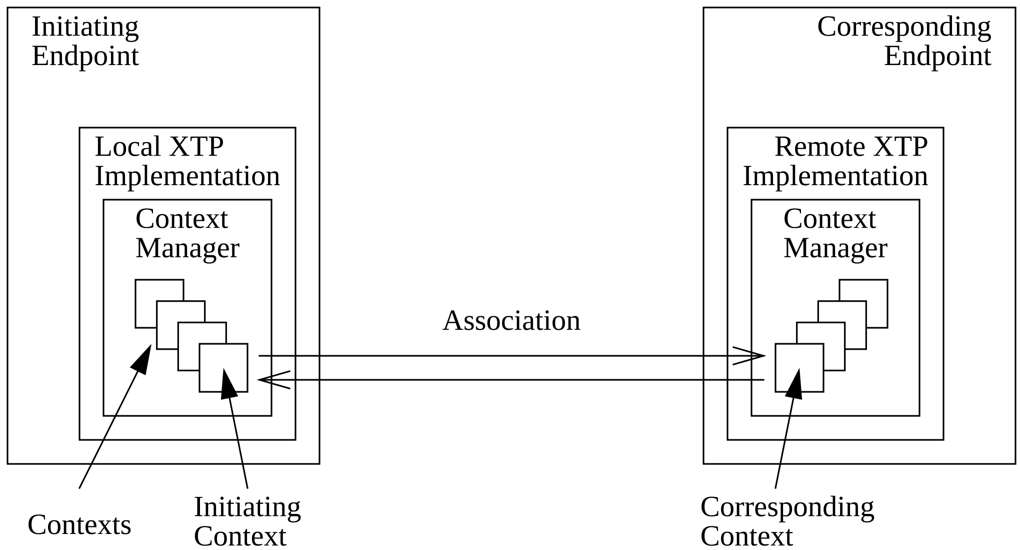

Figure 1-1 — XTP Communication Model

Figure 1-2 shows the establishment of an association (again, this is a unicast association for simplicity). A context at an end-system is initially in a quiescent state. A user awaiting the start of an association requests that the context be placed into the listening state (1). The context now listens for an appropriate FIRST packet. The FIRST packet is the initial packet of an association. It contains explicit addressing information. The user must provide all of the necessary information for XTP to match an incoming FIRST packet with the listening context.

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

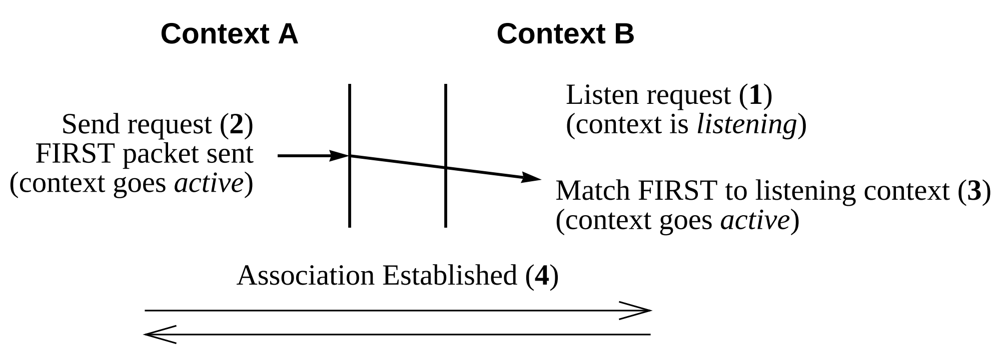

Figure 1-2 — Association Establishment

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

Once all of the data from one user have been sent, that data stream from that user's context can be closed. Sentinels in the form of options bits in a packet are exchanged to gracefully close the connection. Other forms of less graceful closings are possible by abbreviating this exchange. When both users are done, and both data streams closed, the contexts move into the inactive state. One of the contexts will send a sentinel that causes the association to dissolve. At this point, both contexts return to the quiescent state.

All of XTP's packet types use a common header structure. All of the information necessary to steer the packet's payload to the proper point of processing is carried in the header. Much of how an XTP context operates is controlled by bitflags concentrated in one field in the packet header. Fifteen flags are defined, including bitflags to facilitate connection shutdown, set the control policies, and place markers in the data stream.

XTP flow control is based on 64-bit sequence numbers and a 64-bit sliding window. 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. Rate control is considered a traffic shaping parameter; a traffic segment provides a means for specifying the shape of the traffic so that both end-systems and intermediate systems can manage their resources and facilitate service quality guarantees.

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 retransmission. Only data that is shown to be missing via control messages may be retransmitted. This avoids spurious and possible congestion-causing retransmissions. The error control algorithm can also be aggressive: a method for a quick-acting error notification is provided. The error control algorithm in multicast is identical to the unicast algorithm, although additional sophistication is required to manage state variables and achieve continuous streaming.

1.2 Capsule Changes from XTP 3.6

Name Change. With the removal of all routing mechanisms from the XTP (see below), XTP looks more like a transport protocol rather than a transfer protocol. As such, the name is now officially XTP, the Xpress Transport Protocol.

Header Format. The header is now 32 bytes long, ending on an 8-byte boundary. Several of the fields have changed size and positioning from XTP 3.6 to XTP 4.0, and some fields have been dropped, as shown in Figure 1-3. These changes are discussed in detail below.

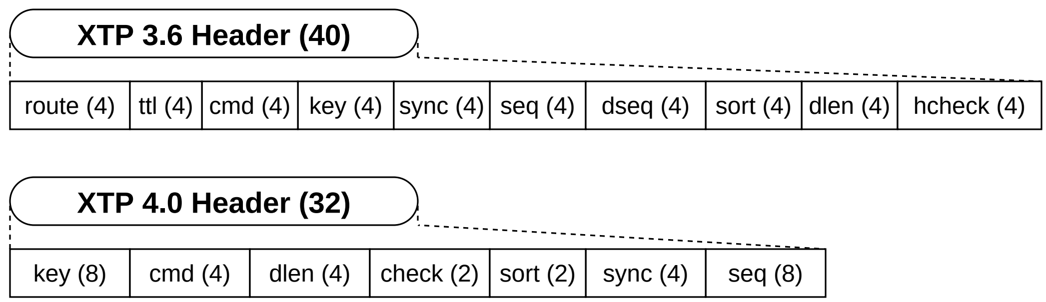

Figure 1-3 — XTP 3.6 and XTP 4.0 Header Comparison

Sixty-Four Bit Context and Sequence Space Identifiers. The *key*, *seq*, and related fields, both in the header and the control packets, are now 64 bits long. The 64th (most significant) bit in the *key* field represents the “return key indicator.”

Command Field Format. The command field *cmd* has been simplified. The *offset* field is no longer present, so the *options* field is expanded to include this byte. The *ver* field is expanded from 2 bits to 3 bits. The version number for XTP 4.0 and all future XTP specifications that are backward compatible with XTP 4.0 is `001`, changed from `000` in XTP 3.6.

All Routing Mechanisms Dropped. The new header format reflects the removal of the *route* and *ttl* fields. The routing functionality is no longer supported. (This implies that the full context lookup now requires only the *key* field and the unique host identifier from the source portion of the frame that will carry the XTP packet, see below.)

The* dseq *Field Eliminated. The *dseq* field has been removed from the header. There is no replacement for its primary function of notifying the sender of fully delivered data. The secondary function of specifying the starting sequence number for the data stream in the return direction has been replaced by a requirement that both sequence spaces start at zero.

Trailer Eliminated. The trailer segment from XTP 3.6, which contained only the *dcheck* field, has been eliminated from XTP 4.0. The checksum field is now in the header and is called the *check* field. The checksum field covers either the header alone (`NOCHECK` is set) or the entire packet (`NOCHECK` is cleared).

New Checksum Algorithm. The checksum algorithm now used in XTP 4.0 is the IP checksumming algorithm. There is no provision for including a “pseudo-header” in the checksum calculation.

The* sync *Field Definition. The *sync* field is now defined to be incremented by one for each outgoing packet with the `SREQ` bit set. The outgoing packet carries the newly incremented value.

The* sort *Field Definition. The *sort* field, used to indicate a priority ordering during packet processing, is now 16 bits wide, with zero being the lowest priority value.

Address Segment Format. The Address Segment format has changed slightly. The *service* field has been moved from the Address Segment to the Traffic Specifier (a new segment); in its place is an address domain demultiplexer called *adomain*. The *adomain* field is used to distinguish between address spaces that have the same address format. The standard address formats defined by the XTP Forum have changed to reflect the fact that no source routing addresses are needed. As with all segments, the Address Segment must end on an 8-byte boundary.

Traffic Specifiers. A new segment, called the Traffic Specifier, is defined in XTP 4.0. This segment is similar to the Address Segment in that it has a length and format field, and various formats are defined by the XTP Forum. The *service* field has been moved to this segment.

FIRST and JOIN Packets. The FIRST and JOIN packets formats have changed to include the Traffic Specifier segment. The JOIN packet is the new name for the PATH packet. The JOIN packet is now used exclusively for joining an in-progress multicast association. The FIRST packet also carries an indication of the allocation for the return direction.

Control Packets. Control packet information is now split into three more focused packets, a CNTL (common control) packet, an ECNTL (error control) packet, and a TCNTL (traffic control) packet. The CNTL packet is used to exchange flow control and other commonly needed parameters. The ECNTL packet is used only when an error in the data stream is detected. The TCNTL packet is used for traffic specification negotiation.

Semantics of the* `RCLOSE` *Bit. The `RCLOSE` definition is modified in XTP 4.0 to reflect the following semantics. The `RCLOSE` bit is only sent (1) in the FIRST packet and all subsequent outgoing packets, or (2) as a confirmation of all data received upon receipt of the `WCLOSE`. In the first case, the initiator of the association indicates that this conversation is simplex. In the second case, the `RCLOSE` implies that all data have been gracefully received. Upon receipt of a `WCLOSE`, the receiver must take action to acquire any missing data before answering the `WCLOSE` with an `RCLOSE`. A received `RCLOSE` without previously sending WCLOSE, except when the RCLOSE is sent in the FIRST packet, is a protocol error.

A New* options *Bit. The *options* bit EDGE is defined to elicit control packets without using a timer-based SREQ request. The transmitter changes the state of the EDGE bit, from 1 to 0 or from 0 to 1, when a control packet is desired. The receiver generates a control packet when it notices the change in the state of the EDGE bit.

The Zero-Length DATA Packet Indication Eliminated. XTP 3.6 dedicated a special *btag* value for the zero-length DATA packet. This special case has been eliminated in XTP 4.0; zero-length DATA packets are not permitted.

The MAINT and MGMT Packet Types Eliminated. These packet types were never used.

The ROUTE and RCNTL Packet Types Eliminated. These packet types will never be used since routing mechanisms have been eliminated from XTP 4.0.

Full Context Lookup. The full context lookup procedure is now defined to take two, rather than three, values. The *key* and a source host identifier are all that is required for matching this packet to the appropriate context. The source host identifier must uniquely identify the host among all hosts in the network.

Reliable Multicast. Reliable multicast associations depend on the XTP multicast transmitter processing the control packets from a known set of active receivers, hence reliability is based on both group management and data stream integrity. This is in contrast to XTP 3.6, where group management was viewed as an auxiliary function. The XTP 1-to-*N* multicast facility enables construction of more complex services, such as multiple source to multiple receiver (*N*-by-*M*).

1.3 Notational Conventions

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, “bitflag” denotes a 1-bit field contained within another field, and **“**segment” denotes an object consisting of one or more fields.

The bits in a field are represented left to right from most to least significant, unless otherwise stated. Likewise with bytes. Bits referred to by name are annotated in capital letters in courier font (e.g., RTN). A field name is given in italic font (e.g., *key* field).

The number of bytes in a field is given by a number in parentheses (e.g., *key*(8)). When a field has variable width, a variable such as “*n*” is used as the number of bytes or a factor of the number of bytes (e.g., *name*(n) and *name*($8n$)). The number of bits in a bitfield is given by a number following the field name and a colon (e.g., *pformat*:5).

Occasionally a local context variable will be defined for aiding in the explanation of XTP's functionality. These variables, highlighted in the text in bold (e.g., **saved_sync**), are not intended to dictate an implementation method.

Throughout the text, comparison of the magnitude of one unsigned integer with another follows this convention: The value of *A* is said to be "less than" the value of *B* if

$$(B - A - 1) < 2^{n-1}$$

where n is the number of bits in the field (note that the subtraction is an unsigned integer operation). This handles the logic when *B* rolls over (overflows and returns to 0) before *A* does, and assumes that any two values, when compared, will never have a distance of more than half of the number space. In this light, the phrase "*B* is a higher sequence number than *A*" means that *A* is "less than" *B*.

1.4 Glossary of Terms

The following terms, listed alphabetically, are used throughout this document.

active receiver a **multicast receiver** whose control information is used by the **multicast transmitter** when the transmitter runs its control algorithms.

application the software running in the higher-layer client that uses XTP for communication services.

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**.

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

concentration a form of **multicast association** 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** combine to overrun a **receiver**.

connection a long-lived **association**.

context the set of state variables representing an instance of the use of XTP at an **endpoint**; one half of an **association**. A context can be both a **sender** (on the outgoing **data stream**) and a **receiver** (on the incoming **data stream**).

datagram usually an unreliable, connectionless service over which unacknowledged **message**s, possibly multi-packet messages, may be transmitted. 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 *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.

endpoint a **host** participating in an association.

end-system a **host** equipped with an XTP implementation.

end-to-end inclusion of all processing for sending data from one **endpoint** to another **endpoint**.

error control the procedures used to detect and possibly correct lost data.

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, or **allocation**, to a remote host.

go-back-*N* retransmission a 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.

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

host a computing device that contains an XTP implementation and can participate in the exchange of XTP packets.

link a direct hardware path.

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

MTU Maximum Transmission Unit, defined as the maximum size for an XTP packet within a specific physical medium.

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

multicast association an association in which a **multicast transmitter** sends to one or more **multicast receiver**s. In XTP, the data flow is simplex from the transmitter to the receivers.

multicast receiver a **context** that receives data sent on a multicast address from the **multicast transmitter**. In XTP, a multicast receiver joins the multicast group either at its inception (receiving for a FIRST packet) or by joining an in-progress **multicast association** (sending a JOIN packet).

multicast transmitter a **context** that transmits to a group of receivers. In XTP, this context must maintain knowledge of the **active receiver** set, and resolve the control information from that set of active receivers.

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.

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.

receiver the **context** for which particular data are incoming.

reservation mode a flow control behavior wherein a transmitter 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 quite different from 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 available receive buffers. This form of control is demonstrated by both the NETBLT and VMTP protocols.

rtt round-trip time, defined as 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 data are resent, rather than. all data from a particular point in the data stream.

sender the **context** for which particular data are outgoing.

sequence number an identifier assigned to each byte of data in the **data stream**, incremented by one for each byte starting at an initial sequence number.

transaction a term with many definitions; in the world of transport protocols it generally implies a request/response **handshake**.

1.5 Synopsis

The next section, “Packet Structures,” describes the syntax of the structures that make up XTP packets. Section 3, “Packet Types,” shows each of the these packet structures as they are used in specific XTP packet types. In Section 4, “Unicast Functional Specification,” and Section 5, “Multicast Functional Specification,” the various protocol algorithms are discussed. Section 6, “Encapsulation,” shows how XTP packets become the payload for various underlying data delivery services. This document ends with several appendices for information that does not really fit in the main flow of the text.

2 Packet Structures

A packet is the basic unit for information exchange between the endpoints of an association. The fields within a packet hold the information pertaining either to the state of the association or to the data being transferred. The layout, or structure, of the packet facilitates retrieval of the information held within; there is no inherent structure to the packet once it is handed off to the underlying data delivery service. The receiving endpoint parses the packet only by knowing the packet structures *a priori*.

Each XTP packet carries a common header structure followed by a payload segment, as shown in Figure 2-1. There are two basic types of packets in XTP, *control packets* and *information packets*. The control packets carry a Control Segment and are used to exchange protocol state information between contexts in an association. The information in control packets is not given directly to the user, but is used by the context to effect the control algorithms. User information, including user data and protocol diagnostic messages, are carried in the Information Segment of information packets.

This section discusses the various segments and structures that comprise the XTP packet syntax. XTP packet design ensures that all major segments and all 8-byte fields start on 8-byte boundaries, and all 4-byte fields start on 4-byte boundaries. Variable length fields are avoided, and variable length segments always have a direct method for determining length. Control information is partitioned into information that is used in a common, error-free case (such as flow control information), and information that is not transmitted as frequently (such as error control and traffic shaping information). Addressing information is not carried in each XTP packet, but rather is carried only once per association.

2.1 Byte Order

All protocol fields in XTP packets must be sent and received in network standard byte order, where the most significant byte of a 32-bit field is sent first. (This is also known as “big-endian” byte order.) All header, control, addressing, and traffic information is affected by this rule; only user data is not. Systems whose native byte order representation is not the same as the network standard byte order must permute the bytes of protocol fields after reception and before transmission.

2.2 Header Format

All XTP packets use a fixed header syntax consisting of the following fields: *key*, *cmd* (command), *dlen* (data length), *check* (checksum), *sort* (priority), *sync* (synchronizing number), and *seq* (sequence number). The *key* field steers the packet to the proper destination context. The *cmd* field dictates how the packet is to be processed. The *dlen* and *seq* fields identify this packet’s contents with respect to the data stream. The *check* and *sync*

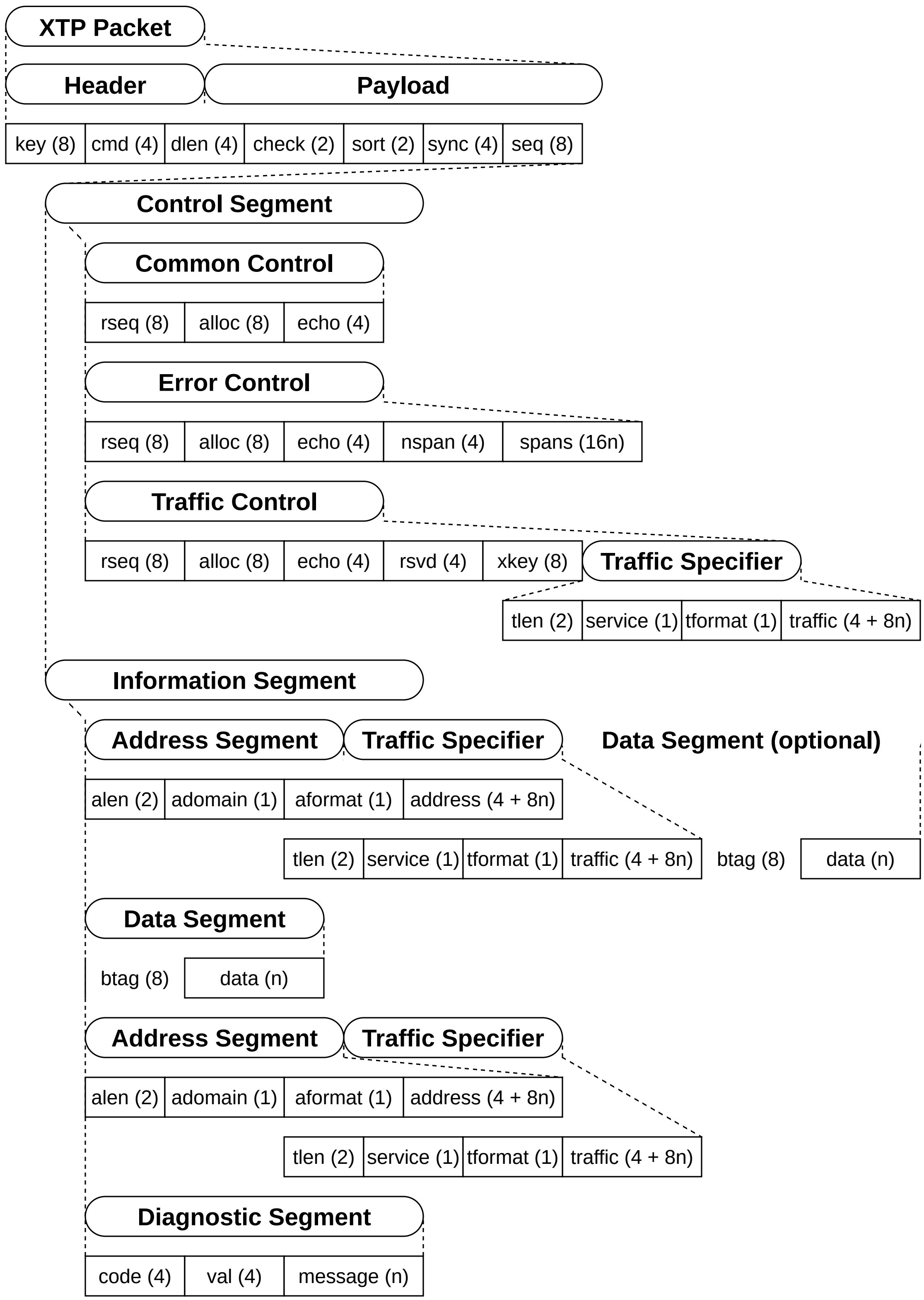

Figure 2-1 — XTP Packet Structure Overview

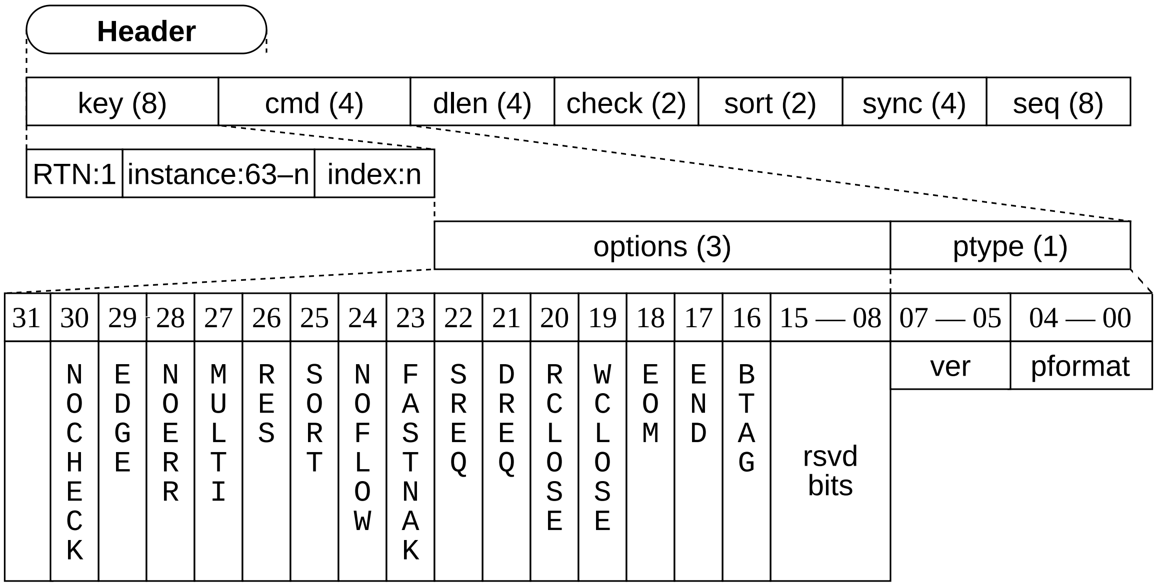

Figure 2-2 — Header Fields

fields are used to determine the validity of the packet, and the *sort* field orders the act of parsing the packet among all contending activities. The header and its fields are shown in Figure 2-2.

2.2.1 Key Field

The *key* field associates a packet with a context. It is similar to the connection or transaction identifier found in most protocols. 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 in any state but quiescence, and “inactive” when its context is quiescent. The *key* field is always interpreted and must contain a meaningful value in all packet types.

The *key* field is 64 bits wide. As shown in Figure 2-2, it contains a 63-bit *key* value and reserves the most significant bit (`RTN`) as a flag. The `RTN` bit directs the interpretation of a *key* in received packets: if the `RTN` bit is not set, then the *key* identifies a context on the end-system sending the packet containing the *key*; if the `RTN` bit 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 `RTN` bit set is called a *return key.*

When a context is instantiated, a *key* value is assigned to that context such that the `RTN` bit is cleared and the 63-bit value is unique within that host’s XTP implementation. When the context sends a packet, it fills in the *key* field of the packet header with its *key* value so that the receiver of the packet can, along with other information, determine which context sent the packet and infer, therefore, which context is to receive the packet (see Sec-

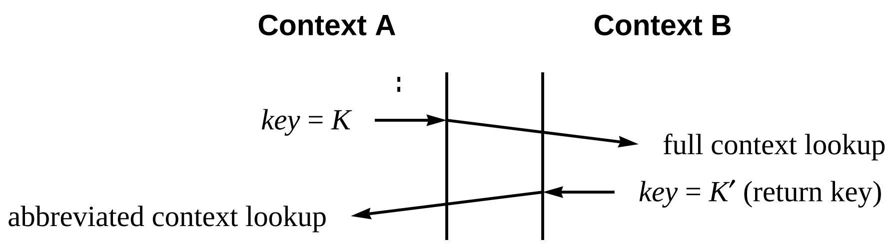

Figure 2-3 — Use of the *key* Value

tion 4.2.2, “Full Context Lookup”). The receiving context notes the packet’s *key* value and uses it, with the `RTN` bit set, in all of its outgoing packets. Such a packet, when received, can be mapped directly to the appropriate context because that *key* value was generated for that context, and is therefore unique within the host (see Section 4.2.3, “Abbreviated Context Lookup”). This is illustrated in Figure 2-3, where *K* is the key value generated at Context A. The packets sent from Context B use the return key value *K′*.

A *key* value of zero is illegal except in a JOIN packet where it has special significance (see Section 5.3.4, “Joining an In-Progress Multicast Association”).

Implementation Note

Key aliasing is a condition that occurs when a new use of a *key* value, arriving in a packet, matches an old use of the *key* at the receiving host. The packet is mistakenly given to the wrong context. To avoid this and aid with the mapping and uniqueness properties of *key* values, the *key* can be further divided into an *instance* part and an *index* part, as shown in Figure 2-2. The number of bits in each part is determined by the implementation, subject to the number of contexts supported. The *index* value is used to select the context. The *instance* value validates active *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* and its context can be distinguished from previous uses of the same *index*. As *key* values and contexts are reused, the *instance* field will eventually wrap around to reuse *instance* values. The time required for this to occur is called *the wrap time for instances*. In order to avoid key aliasing, the minimum wrap time must 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 bounded by twice the length of the CTIMER interval (see Section 4.3).

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, in contexts per seconds, by the system and its applications). An expression for minimum wrap time is:

$$\left(2^{I}\right)\frac{N}{R}$$

To avoid key aliasing, the implementor must ensure that the CTIMER interval is much smaller than one-half of the value of the above expression. (The expression assumes that quiescent contexts are activated in FIFO order. A simple free list that 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.)

For example, assume that there are 32 bits given to the *instance* field, but there is only one context in the implementation. Also assume that the rate of context use is one per millisecond, or 1000 per second. The formula above yields approximately 2^{22} seconds as the minimum wrap time, so the CTIMER interval must be much less than 2^{21}, or 2 million, seconds. As a practical matter, however, the CTIMER should be set in the one to several hour range.

2.2.2 Command Field

The 32-bit *cmd* field carries the options set for this packet, the version of the protocol that generated this packet, and the format of the packet. Figure 2-2 shows the syntax of the *cmd* field. The *cmd* field must always contain meaningful values for all its subfields and bits.

2.2.2.1 Command Options

The bitflags in the *options* field are given in Table 2-1. These bits select XTP operating modes and mechanisms. The logic convention is positive: a function is enabled if the corresponding bit is set (value is 1), it is disabled if the bit is cleared (value is 0). Zero or more bits may be set in each header.

`NOCHECK`

When set, this bit indicates that the checksum is calculated over the header fields only, and the rest of the packet is not summed. When cleared, the checksum is calculated over the whole packet. The *check* field in the header contains the result of the checksum (see Section 2.2.4, “Checksum Field”).

`EDGE`

When a packet is received, the value of the `EDGE` bit is compared with the value of the `EDGE` bit from the most recently received packet. If their values differ, a control packet is issued in response. If they are the same, nothing is done. The sender can toggle this bit during a stream of packets to request control packets without using the `SREQ` or `DREQ` mechanisms. This status request is not guarded by a timer, in contrast with the case when `SREQ` is used.

`NOERR`

When set, this mode bit informs the receiver that the sender will not retransmit data, and directs the receiver to disable error correction processing (see Section 4.6, “Error Control”). This is called “no-error mode.” When status is requested, the receiver must always acknowledge the highest received sequence number. Setting

Bit	*Mask*	*Description*	*Potential Change*	*Expectation*
	`0x800000`	not used, must be cleared		
`NOCHECK`	`0x400000`	Disable checksum function	Per packet	Per context
`EDGE`	`0x200000`	Edge-triggered status requests	Per packet	
`NOERR`	`0x100000`	Disable error control	Per packet	Per context
`MULTI`	`0x080000`	Multicast mode	Per association	
`RES`	`0x040000`	Reservation mode	Per packet	Per context
`SORT`	`0x020000`	Enable sorting	Per packet	Per context
`NOFLOW`	`0x010000`	Disable flow control	Per packet	Per context
`FASTNAK`	`0x008000`	Enable aggressive error control	Per packet	Per context
`SREQ`	`0x004000`	Status requested	Per packet	
`DREQ`	`0x002000`	Delivery status requested	Per packet	
`RCLOSE`	`0x001000`	Reader closed	Per packet	
`WCLOSE`	`0x000800`	Writer closing	Per packet	
`EOM`	`0x000400`	End of message	Per packet	
`END`	`0x000200`	End of association	Once	
`BTAG`	`0x000100`	Beginning data tag	Per packet	

Table 2-1 — Header Options Bits

this bit does not prevent the receiver from sending control packets for other reasons. This bit takes precedence over the `FASTNAK` bit; even if both bits are set, error control is disabled.

`MULTI`

When set, this bit indicates use of multicast mode (see Section 5, "Multicast Functional Specification"). The value of this bit must be the same in all packets over the lifetime of the association.

`RES`

When set, this bit enables reservation mode. By setting this bit, the sender indicates to the receiver that the *alloc* values provided by the receiver in its control packets must represent actual client buffer space available, not XTP internal buffer space (see Section 2.3.1.2, "Allocation Field"). The purpose of reservation mode is to avoid overflowing XTP buffers during bulk transfers.

`SORT`

When set, this bit indicates that the value in the *sort* field of the header should be interpreted and used for sorting/prioritizing the packet (see Section 2.2.5, "Sort Field"). This is called "sort mode." When this bit is cleared, the *sort* field must contain the value zero.

`NOFLOW`

When set, this mode bit indicates that the sender does not observe flow control restrictions (see Section 4.4, “Flow Control”). Specifically, the allocation limit imposed by the receiver (in the *alloc* field of control packets) does not constrain the sender.

`FASTNAK`

This bit indicates that the receiver should provide aggressive error notification (or “fast negative acknowledgement,” see Section 4.6.2, “Acknowledgements and Retransmission”). The `FASTNAK` bit should be set by a sender only when the protocol layers below XTP do not reorder packets excessively. If a receiver detects an out-of-order packet and the `FASTNAK` bit is set, then the receiver immediately returns an ECNTL packet to the sender to indicate the error. The `FASTNAK` bit has no effect when the `NOERR` bit is set.

`SREQ`

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

`DREQ`

When set in a data-bearing packet, the receiver must send a control packet after all enqueued data, up to and including any in this packet, have been delivered to the higher-layer application. When set in a control packet, this bit indicates that the sender requests that the receiver send a control packet after all data with sequence number less than the value in the *seq* field of this packet have been delivered to the higher-layer application.

`WCLOSE` and `RCLOSE`

These bits are the basis for disconnect handshakes carried out by the close state machines (see Section 4.2.7, “Association Termination”). The `WCLOSE` bit indicates that no more data will be written to the outgoing data stream. The `RCLOSE` bit indicates that all bytes written to the incoming stream have been received with respect to the error control policy in use.

`EOM`

This bit is used to delimit message boundaries in a data stream; when set, this bit denotes the end of a message. A data-bearing packet containing `EOM` carries the trailing bytes of a message. Control packets must not carry an `EOM` bit. The `EOM` bit is not manipulated by XTP, nor does XTP create 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, this bit indicates that the sending context is being released. It is used in the last packet of a closing handshake and also when an association is aborted (see Section 4.2.7, “Association Termination”).

`BTAG`

When set, this bit indicates that the first eight bytes of the Data Segment within the Information Segment become the *btag* field, used to contain tag information for the higher-layer application (see Section 2.4.2.2, “Beginning Tag Field”). The `BTAG` bit is meaningful only in data-bearing packets — it must be cleared in all other packets. Like `EOM`, `BTAG` is not manipulated by XTP.

An XTP transmitter controls the settings of *options* field in every packet. Some bit-flags 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 2-1. *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. For example, 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 exactly once in the lifetime of a context (unless the bit is included in a retransmitted packet).

Even though some bits may 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 2-1. *Per context* in this column indicates this expectation.

The bits `NOERR`, `MULTI`, `RES`, `SORT`, and `NOFLOW` are called “mode bits”; changing mode bits during the association (except for `MULTI`, which is not allowed to change) can cause unpredictable behavior unless there is a synchronization of the contexts prior to changing the mode bits (see Section 4.2.6, “Changing Modes”).

2.2.2.2 Packet Type Field

The least-significant byte of the *cmd* field is called the *ptype* field. Within this byte, bits 0 through 4, called the *pformat* field, identify the XTP packet type. Bits 5 through 7 indicate the XTP version (*ver* field). The *pformat* field is always interpreted and must contain a meaningful value. The *ver* field is also always interpreted and, in XTP 4.0, must be set to binary `001`.

The *pformat* values are enumerated in Table 2-2. FIRST, DATA, JOIN, and DIAG packets use an Information Segment. The FIRST (Section 3.1) packet is the initial packet of an association and contains an Address Segment (Section 2.4.1), a Traffic Specifier (Section 2.3.4), and optionally a Data Segment (Section 2.4.2). DATA packets (Section 3.2) are

pformat	*Decimal*	*Hex*	*Definition*
DATA	0	`0x00`	user data packet
CNTL	1	`0x01`	state exchange control packet
FIRST	2	`0x02`	initial packet of an association
ECNTL	3	`0x03`	error control packet
TCNTL	5	`0x05`	traffic control packet
JOIN	6	`0x06`	multicast join packet
DIAG	8	`0x08`	diagnostic packet

Table 2-2 — Types of XTP Packets

used for subsequent data transfers, and contain only the Data Segment. These are the two packet types responsible for user data transfer.

CNTL (Section 3.3), ECNTL (Section 3.4), and TCNTL (Section 3.5) packets use a Control Segment. The CNTL packet conveys control information such as flow control window values through the Common Control segment (Section 2.3.1). The ECNTL packet additionally conveys error control information through its Error Control segment (Section 2.3.2). The TCNTL packet is used to negotiate a traffic specification through its Traffic Control segment (Section 2.3.3). These are the three packet types responsible for state information exchanges between contexts.

The JOIN packet (Section 3.6) is used to join an in-progress multicast association. Its format is the same as a FIRST packet without the optional Data Segment. The DIAG packet (Section 3.7) uses a Diagnostic Segment (Section 2.4.3) to convey diagnostic information.

Design Note

Control packets have odd *pformat* values and Information packets have even *pformat* values. This is to allow an XTP engine to switch on the least significant bit for steering the packet into the control information processors or the user data processors.

2.2.3 Data Length Field

The *dlen* field specifies the number of bytes immediately following the header, that is, the number of bytes in the payload segment. Since zero-length DATA packets are not allowed, the *dlen* field will always have a non-zero value.

2.2.4 Checksum Field

The *check* field contains the result of the XTP check function (see Appendix A for the check function algorithm). If the `NOCHECK` bit is set, the *check* field contains the checksum over the packet header and nothing more. If the `NOCHECK` bit is cleared, the *check* field contains the checksum over the whole packet, including the header and the payload segment.

2.2.5 Sort Field

The *sort* field is intended to provide an expedited service for selected packets. The *sort* field is interpreted only when the SORT bit in the header is set. When the SORT bit is cleared, the *sort* field must contain zero.

As the XTP subsystem scans all of the active contexts on the transmitter side, it gives preference to contexts that have priority data to transmit versus those that do not. The contexts with priority traffic emit packets whose SORT bit is set and whose *sort* field is meaningful; those without priority data emit packets whose SORT bit is not set and whose *sort* field contains the value zero. On the receiver side, data are delivered to clients in an order consistent with the priority of the incoming data.

The *sort* field is an unsigned 16-bit number that specifies an integer ordering. A *sort* value of zero indicates the lowest priority, while increasingly positive *sort* values represent increasingly higher levels of priority.

The XTP implementation must serve all active contexts in priority order. When there is an opportunity to transmit, the packet with the highest priority is selected, consistent with the rate and flow control restrictions of each context. This selection procedure is repeated at each transmission opportunity. Contexts not operating in sort mode (i.e., their packets have the SORT bit cleared) are serviced in FIFO order after all contexts that are operating in the sort mode.

On the receiver side, sort mode operation implies that, of all packets received, enqueued, and awaiting processing, the next packet to be processed will be the one with the highest priority. Packets with the SORT bit cleared are processed in FIFO order after packets whose SORT bit is set.

More sophisticated queueing policies, such as preemptive or destructive preemptive queueing, may be used to implement the sort mode. An implementation is responsible for stating input and output queue behavior, and specifying the maximum time bounds for preemption for both input processing and output processing.

Implementation Note

Deadline-driven processing can be emulated by quantizing the deadline into a 16-bit value, inverting the bits of the 16-bit value, then using this as the *sort* value.

2.2.6 Synchronizing Handshake Field

The 32-bit *sync* field provides the basis for the synchronizing handshake (see Sections 4.6.3 and 5.4.2). Each context keeps a record of the last sent *sync* value in a variable **saved_sync**. When a packet is sent with its `SREQ` bit set, the **saved_sync** value is incremented by one, and this new **saved_sync** value is placed into the *sync* field of the outgoing packet. The rules for a sender are as follows:

1. The value of the *sync* field in outgoing packets increases only when the `SREQ` bit is set in an outgoing packet, and remains constant at the most recently transmitted value for all other packets.
2. Retransmitted packets must also abide by Rule 1; the *sync* values used are not the ones originally transmitted, but values generated under Rule 1.

An XTP receiver saves the highest received *sync* values into a local variable **rcvd_sync**. The value of **rcvd_sync** is later placed into the *echo* field (see Section 2.3.1.3) of each outgoing control packet. The rules for a receiver are as follows:

1. The local context variable **rcvd_sync** is initialized with the value of the *sync* field from the first packet received for this association.
2. The **rcvd_sync** value is placed into the *echo* field of all outgoing control packets.

When a control packet is received, the *sync* field from the control packet is compared to the value in **rcvd_sync**; if the control packet contains a *sync* value that is greater than or equal to the **rcvd_sync** value, the control packet is processed normally. If not, the control packet has information older than information received in previous control packets. In this case, the `SREQ` and `DREQ` bits should be responded to if set, but no other processing should be done on the control packet.

If the WTIMER expires, indicating a status request (sent `SREQ` bit) has not been answered, the context enters the synchronizing handshake.

2.2.7 Sequence Number Field

The *seq* field is an unsigned 64-bit quantity representing a sequence number for the outgoing data stream. A sequence number is associated with every byte of data in that stream. The *seq* field is meaningful and interpreted in all packet types with two exceptions: in the FIRST packet, the *seq* field is used to convey an initial allocation value for the return data stream, and in the JOIN packet sent from a multicast receiver wishing to join a multicast association, where the *seq* field is used to convey the receiver's identity.

For DATA packets, the value in the *seq* field is the sequence number of the first meaningful byte of the Information Segment. The range of sequence numbers representing the data in a DATA packet begins with *seq* and extends to $seq + dlen - 1$.

The value of *seq* in control packets is the next sequence number to be sent on the outgoing data stream. Since control packets do not carry data, no sequence numbers are consumed by a control packet's payload.

Associations always begin both data streams at sequence number 0. Since a sequence number is not necessary to identify the first byte of the outgoing data stream (it is always zero), the *seq* field in a FIRST packet is used to indicate the do-not-exceed sequence number for the incoming data stream of the sender of the FIRST packet. This is the initial allocation value for the return data stream; subsequent control packets carry updated allocation information thereafter.

Example

A host intends to send a 300-byte message (this byte count includes user data and address information). Assume that the message is transmitted as three packets, as shown in Figure 2-4. The *seq* field in the FIRST packet carries the initial return allocation for A's incoming data stream. The first DATA packet's *seq* field carries the value 100 since the FIRST packet consumed 100 bytes of sequence space. The sequence number of the last byte of this DATA packet is 199, so the *seq* field in the CNTL packet contains the next sequence number to be used, which is 200. The next DATA packet, therefore, also carries 200 in the *seq* field.

The *seq* field of the retransmitted DATA packet again contains the value 100, since this identifies the data within. However, the CNTL packet following the retransmission carries 300 in the *seq* field, since this is the next new sequence number to be sent.

The use of the *seq* value in a JOIN packet is described in Section 5.3.4, "Joining an In-Progress Multicast Association." The JOIN packet is used by a context wishing to join an in-progress multicast association as a multicast receiver. The *seq* field on this initial JOIN packet carries the context's return key value as a method for identifying a single receiver in a multicast group. The *seq* field in the JOIN packet sent by the multicast transmitter in response to the join request carries the current sequence number of the in-progress association.

The *seq* field of a DIAG packet must contain the *seq* value of the incoming packet that caused the error that the DIAG packet is reporting. If the DIAG packet is generated for some reason other than an error in an incoming packet, the *seq* field must contain the value zero.

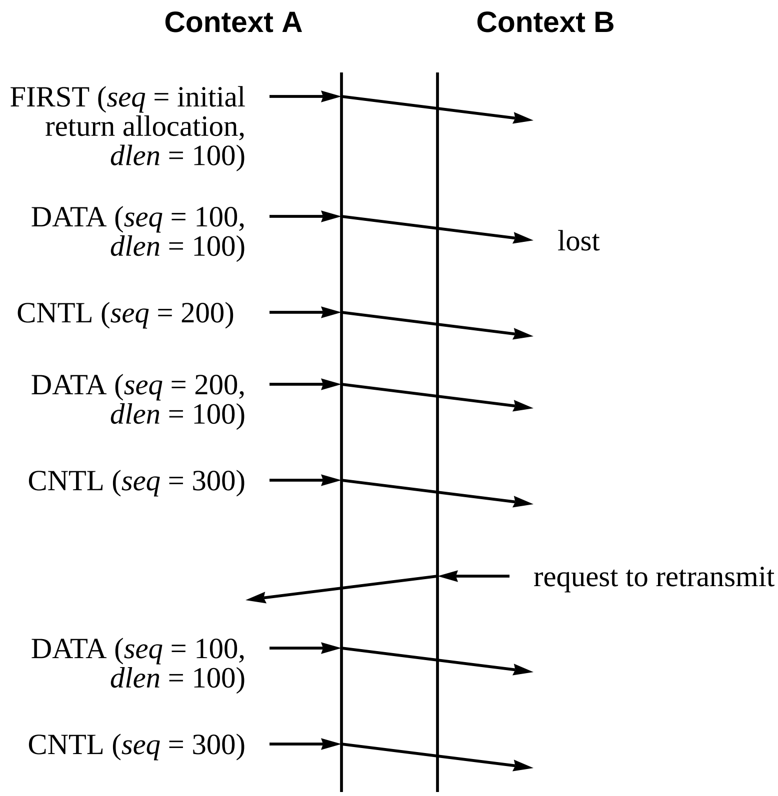

Figure 2-4 — Use of the *seq* Field

2.3 Control Segment

A Control Segment reports the state of the context that sent it. XTP packets containing a Control Segment as their payload are referred to as *control packets*. Control packets are used to exchange state information between XTP endpoints.

The Control Segment is included in CNTL, ECNTL, and TCNTL packets. These three packet types correspond to the three possibilities for the Control Segment as shown in Figure 2-1. A CNTL packet contains a Common Control segment with information most commonly needed, such as flow control information. The ECNTL contains an Error Control segment, which contains all of the fields of the Common Control segment but also holds error control information. The TCNTL packet contains a Traffic Control segment, and is used to negotiate traffic specifications.

Design and Implementation Note

Control packets are split into these three types to avoid sending unnecessary information — only when errors occur is an ECNTL packet used, and only when traffic specification needs to be negotiated is a TCNTL packet used. If all of the error control information were in a common CNTL packet, the error control fields would have to be parsed and checked for each CNTL packet, even if no error had occurred. Further, the packet type implies what type of processing will be required, so an XTP implementation can steer the packet to the proper processing elements as soon as it knows what packet types it has received.

2.3.1 Common Control Segment

The format of a Common Control segment is shown in Figure 2-5. These three fields represent the state information for which a control packet is most commonly needed. The fields in this segment are common to all three control packet types. The first two fields, *rseq* and *alloc*, are flow control parameters and together define the flow control window. The third field, *echo*, is used to identify a particular control packet as a response to a packet sent with the `SREQ` bit set.

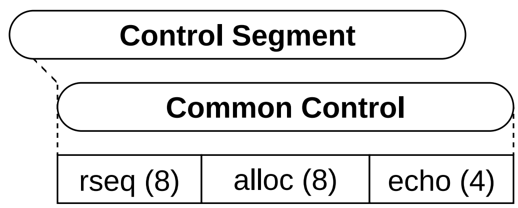

Figure 2-5 — Common Control Segment Fields

2.3.1.1 Received Sequence Number Field

The *rseq* field is interpreted and must contain a meaningful value in all control packets. The *rseq* field holds the sequence number of the next in-sequence byte expected on this data stream, so it is one greater than the highest contiguously received data byte. The *rseq* value serves as the lower edge of the flow control window and, consequently, acknowledges all data whose sequence numbers are less than the value of *rseq*. Whenever error processing has been turned off (the last received packet had its `NOERR` bit set), the value of *rseq* is one past the highest sequence number ever received on the data stream. For a context in which no data have as yet been received, *rseq* is the starting sequence number for the data stream, which is zero.

Example

Suppose a series of seven 100-byte DATA packets with *seq* fields of 100 through 700 are sent to a receiver that sees packets with *seq* fields 100, 200, 300, 600, and 700, but packets with *seq* values of 400 and 500 are not received.

If `NOERR` is cleared, any control packets generated by the receiving context would have the value 400 in the *rseq* field. This means that bytes up to and including 399 were received in order.

If `NOERR` is set, any control packets generated by the receiving context would have the value 800 in the *rseq* field.

2.3.1.2 Allocation Field

The *alloc* field is interpreted when flow control is enabled. The value of the *alloc* field is a sequence number that limits the amount of data a sender may transmit; a sender may send data with sequence numbers up to, but not including, the sequence number in *alloc*. The *alloc* value represents the amount of data the receiver is willing to accept, which may or may not correspond to internal buffer space of the receiving XTP machine. When the `RES` bit is set, however, the *alloc* value represents the buffer space reserved for this association.

Sequence-based flow control is disabled by `NOFLOW` mode. The *alloc* field of received control packets is ignored by a sender in `NOFLOW` mode.

2.3.1.3 Synchronizing Handshake Echo Field

The *echo* field is valid in all control packets. It is used to match a control packet with the status request (sent `SREQ` bit) that may have caused the control packet to be generated. The value of the *echo* field is the highest seen *sync* value from all incoming packets (see Section 2.2.6). The highest *sync* value yet seen is held in a context variable called **rcvd_sync**; when a control packet is generated, the value of **rcvd_sync** is placed into the *echo* field of the control packet. Thus the value in the *echo* field of an incoming control packet indicates the highest seen *sync* value at the time of its transmission.

If the *echo* value is less than the **saved_sync** value, the control packet is not a response to the most recent status request. If the *echo* value is equal to the **saved_sync** value, the information in the incoming control packet is no older than the time at which last status request was generated. Therefore, the *echo* value is useful for:

- taking round-trip time samples
- stopping a synchronizing handshake, if one is in progress
- aggregating control information from the set of multicast receivers

Round-trip time estimates can be collected if the current time is saved when a packet with the `SREQ` bit set is sent (and its *sync* value is recorded in **saved_sync**). This saved time is subtracted from the current time when a control packet arrives whose *echo* value is equal to the **saved_sync** value. (Note that this only works if the control packet is in response to an `SREQ`.) The difference in the times is a round-trip time observation. The value used to load the WTIMER is a value derived by smoothing observations of round-trip times.

Implementation Note

Implementors may use whatever smoothing functions they wish. The following is from Van Jacobson. For each observation of the round-trip time **rtt**,

$$\mathrm{SRTT} = \mathrm{SRTT} + (\mathrm{rtt} - \mathrm{SRTT})/8$$
$$\mathrm{RTTV} = \mathrm{RTTV} + (\mathrm{abs}(\mathrm{rtt} - \mathrm{SRTT}) - \mathrm{RTTV})/4$$

SRTT is the smoothed round trip time, and **RTTV** is the round-trip time variance. The value loaded into WTIMER is

$$\mathrm{WTIMER} \leftarrow \mathrm{SRTT} + 2\mathrm{RTTV}$$

The synchronizing handshake is a procedure used to cause the endpoints of an association to synchronize their state information. Sections 4.6.3 and 5.4.2 specify the procedures for interpreting *echo* and *sync* field values.

In a multicast association, control information coming from multiple receivers must be resolved by the transmitter. Since control packets can arrive at any time and out of order, the age of the control information must be ascertained. The *echo* field of an incoming control packet allows a multicast transmitter to determine the relative age of the control information.

2.3.2 Error Control Segment

The format of the Error Control segment is shown in Figure 2-6. An Error Control segment includes all of the fields of the Common Control segment with two additional fields, *nspan* and *spans*. These fields specify what data have been lost by listing the spans of data that have been received. The Error Control segment is used in the ECNTL packet.

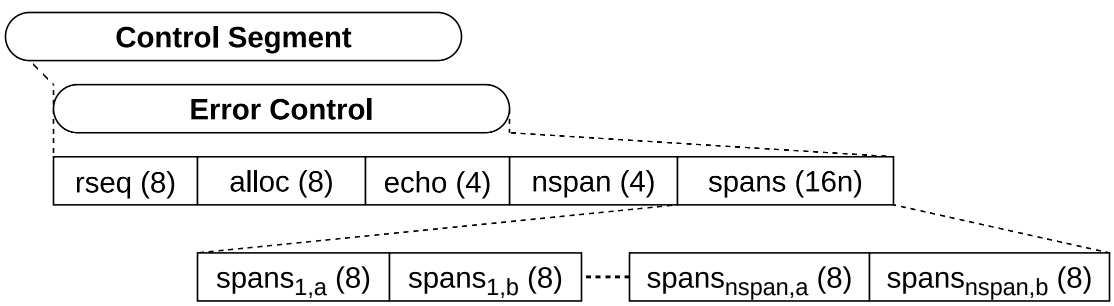

Figure 2-6 — Error Control Segment Fields

2.3.2.1 Number of Spans Field

The *nspan* field specifies the number of spans in the ECNTL packet. Since an ECNTL packet indicates that data are missing, the *nspan* field will have a value of at least one.

2.3.2.2 Spans Field

The *spans* field consists of pairs of sequence numbers defining unbroken sequences (spans) of received data within a data stream. The number of pairs is indicated by the *nspan* field.

The pairs of sequence numbers in the *spans* field indicate ranges of received data that occur after the sequence number in the *rseq* field. 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 ($spans_{n,a}$, $spans_{n,b}$) within *spans* describes an intact range (one span) of received sequence numbers that is bordered on one or both sides by missing data (a gap). The first number in a pair, $spans_{n,a}$, is the lowest sequence number for the span; the second number, $spans_{n,b}$, is one greater than the highest sequence number for that span. The second sequence number must not be less than the first. The first *spans* pair must describe the span with the lowest sequence numbers, subsequent pairs have increasing sequence numbers. The pairs themselves must be in order so that they describe sequentially ordered spans. An XTP receiver is allowed to keep record of fewer spans than actually occur for implementation reasons.

Example

Figure 2-7 illustrates the construction of a *spans* field for a data stream consisting of eleven 100-byte packets with *seq* fields of 0, 100, 200, etc. The receiver sees packets with *seq* field values 0, 100, 200, 300, 600, 700, 900, 1000. Since bytes 0 to 399 are a contiguous sequence, the receiver fills the *rseq* field of an ECNTL packet with 400. The receiver then describes all other intact spans using the *spans* field. Here, two spans exist, so the value in the *nspan* field would be 2 and the pairs within its *spans* field are (600, 800) followed by (900, 1100). This combination of *rseq*, *nspan*, and *spans* makes it possible for the data stream's sender to calculate that sequence numbers 400 through 599 and 800 through 899 should be retransmitted.

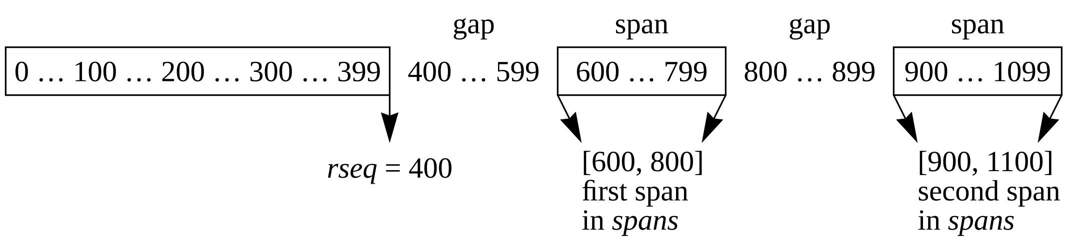

Figure 2-7 — Spans in a Data Stream

An XTP transmitter is allowed to ignore some of the spans information, effecting a go-back-*N* retransmission policy by retransmitting all data starting from *rseq*. An XTP receiver may likewise simplify its retransmission request by setting *nspan* to one and placing the highest sequence number yet seen in both $spans_{1,a}$ and $spans_{1,b}$.

Example

For go-back-*N* retransmission, the receiver in the above example would set *nspan* to 1 and $spans_1$ to (1100, 1100).

2.3.3 Traffic Control Segment

The format for the Traffic Control segment is given in Figure 2-8. A Traffic Control segment includes all of the fields of the Common Control segment with two additional fields and a Traffic Specifier segment. The two additional fields are *rsvd* (reserved) and *xkey* (the exchange key field). The Traffic Specifier segment contains format-specific fields that provide traffic shaping parameters.

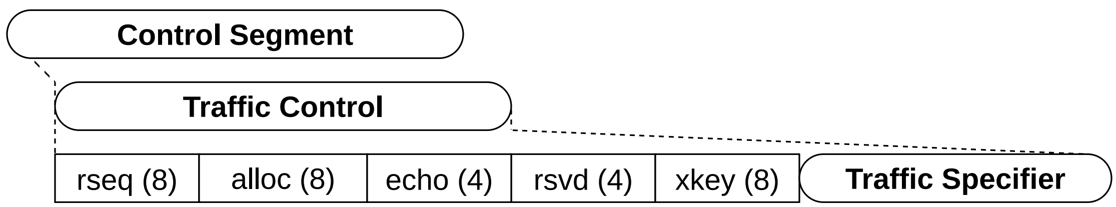

Figure 2-8 — Traffic Control Segment Fields

The TCNTL packet carries the Traffic Control segment as its payload. The TCNTL packet is used to negotiate traffic specification, which usually happens at or near the beginning of the association. The *xkey* field is included in the Traffic Control segment since the key exchange, discussed in Section 4.2.4, also usually happens at or near the beginning of the association.

2.3.3.1 The Reserved Field

The *rsvd* field is reserved and must be zero for all TCNTL packets.

2.3.3.2 Exchange Key Field

The *xkey* field is interpreted in all TCNTL packets. The value of the *xkey* field must be the return key value (i.e., the `RTN` bit must be set) for the context sending the packet. The value in the *xkey* field supports the key exchange mechanism described in Section 4.2.4, "Key Exchange."

2.3.4 Traffic Specifier Segment

The Traffic Specifier segment, shown in Figure 2-9, contains the traffic specification fields for the FIRST and TCNTL packets. These fields are used to negotiate traffic shaping information. (A Traffic Specifier segment is used in a FIRST packet to make the "offer,"

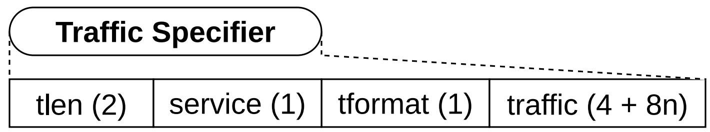

Figure 2-9 — The Traffic Specifier Segment Fields

the “reply” to the offer comes in the Traffic TCNTL packet. Also, once established, the endpoints of the association can use TCNTL packets to renegotiate the traffic specification.)

The length of the Traffic Specifier segment is given in the *tlen* field. The *service* field indicates the type of traffic expected for this association. The Traffic Specifier syntax is determined by the *tformat* field. Two traffic specifications are mandatory, a null traffic specifier, and a rate control traffic specifier (see Figure 2-10).

2.3.4.1 Traffic Segment Length Field

The *tlen* field contains the total length of the Traffic Specifier, including the 4-byte descriptor. This Traffic Specifier must be a multiple of eight bytes, so the minimum value for *tlen* is eight.

2.3.4.2 Service Field

The *service* field is used to indicate the type of transport service used for the duration of this service instantiation. This information is provided by the application and passed along in the FIRST packet to the destination endpoint. The *service* field is transmitted to the receiving context, which uses the value to select an interface for the duration of the association. The defined values for service are enumerated in Table 2-3.

service		*Type of Service*
Decimal	*Hex*	
0	`0x00`	Unspecified
1	`0x01`	Traditional Unacknowledged Datagram Service
2	`0x02`	Acknowledged Datagram Service
3	`0x03`	Transaction Service
4	`0x04`	Traditional Reliable Unicast Stream Service
5	`0x05`	Unacknowledged Multicast Stream Service
6	`0x06`	Reliable Multicast Stream Service

Table 2-3 — Service Type Values

The *service* value zero is used by a listening context to accept any incoming service type, and by a FIRST packet to indicate that the service to be used on this association is unspecified.

Service profile definitions are given in Appendix D. Each service profile defines the expected values of the options bits in the XTP packets used for that service type, as well as the expected packet exchange sequences. An implementation claiming to conform to a service defined in Appendix D must adhere to the packet formats and sequences defined there. The service types defined here are listed because they provide definite hints to the sender and receiver.

It is expected that services in addition to those defined in Appendix D will be defined. To ensure interoperability among services, the *service* values will be assigned by the XTP Forum. Until such assignment is made, other modes of operation should be initiated with *service* value zero.

2.3.4.3 Traffic Field

The *traffic* field format depends on the value of the *tformat* field. There are two mandatory specifications, shown in Figure 2-10. The first specification, *tformat* `0x00`, is used when no traffic shaping parameters are necessary or desired. This may be the case if the underlying data delivery service has no admission policies, and transport level rate control is not desired.

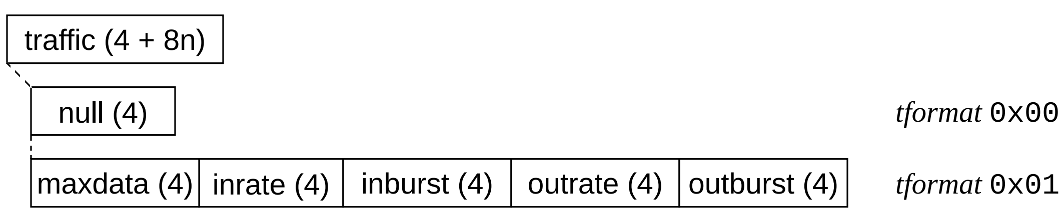

Figure 2-10 — Traffic Field

The second specification, *tformat* `0x01`, is used to convey rate control and other information. The *maxdata* field conveys the maximum Information Segment size that the sender expects to transmit during the lifetime of the association. The *inrate* and *inburst* fields are rate control parameters for the incoming data stream. The *outrate* and *outburst* fields are the (suggested) rate control parameters for the outgoing data stream.

Additional *traffic* field formats are assigned by the XTP Forum, see Appendix B.

2.4 Information Segment

The Information Segment encapsulates user and other protocol and diagnostic information. XTP packets containing an Information Segment as their payload segment are called *information packets*. DATA, FIRST, JOIN, and DIAG packets are information packets corresponding to the possibilities in Figure 2-1. The first two types can contain higher-layer (user) data. These are referred to as *data-bearing packets*. The other two contain transport layer messages only.

There are four segments used in information packets, the Data segment, the Address Segment, the Traffic Specifier, and the Diagnostic Segment. These segments are used in various combinations to construct the several information packets. The Data Segment, Address Segment, and Diagnostic Segment are discussed below; the Traffic Specifier has already been defined in Section 2.3.4.

Information Segments of DATA and FIRST packets always consume sequence space; those of JOIN and DIAG packets do not.

2.4.1 Address Segment

The Address Segment format is shown in Figure 2-11. The Address Segment contains destination and source addressing information. Two packet types require an Address Segment within their Information Segment: FIRST and JOIN. An Address Segment of a FIRST packet consumes sequence space, but that of a JOIN packet does not.

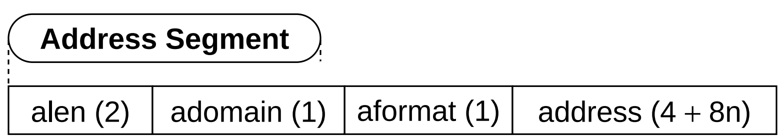

Figure 2-11 — Address Segment Fields

Instead of defining a single XTP-specific addressing scheme, XTP provides parametric addressing where one of several formats can be used to express the source and destination addresses. These formats closely resemble the addressing structures used in several standard network protocols. The intent is to free XTP from the need for an address administration authority and an allocation policy, and to minimize the differences between an XTP service interface and a service interface to one of these protocols.

The Address Segment is carried in FIRST and JOIN packets only. These packets are used to initiate or join an association; once the association has been established or joined, the addressing information is no longer explicitly carried by packets in subsequent packet exchanges.

The addressing information carried in an Address Segment can be thought of as a pattern. Listening contexts submit filters that mask this pattern. A comparison of the Address Segment from a FIRST or JOIN packet and the filter determines if the packet can be accepted by the context that submitted the filter. This operation is independent of the addressing function for the underlying data delivery service, although the addressing information used is often the same.

An Address Segment consists of a 4-byte descriptor followed by a variable-length *address* field. The *aformat* field specifies the format of the address, and the *alen* field specifies the segment's total length.

2.4.1.1 Address Format Field

The *aformat* field identifies the address syntax according to Table 2-4. Address format values 0 through 127 are reserved for use by the XTP Forum. Values for *aformat* in the range of 128 to 255 are for use by the implementor and may be defined for local installations of XTP. There is no guarantee that formats in this range will be compatible across implementations.

aformat		*Address Syntax*
Decimal	*Hex*	
0	`0x00`	Null Address
1	`0x01`	Internet Protocol Address
2	`0x02`	ISO Connectionless Network Layer Protocol Address
3	`0x03`	Xerox Network System Address
4	`0x04`	IPX Address
5	`0x05`	Local Address
6	`0x06`	Internet Protocol version 6 Address

Table 2-4 — Address Formats

Design Note

The address formats given in Table 2-4 are mostly network layer addresses which include host identifiers. Including the host identifier is not strictly necessary for a transport layer protocol, but in this case it avoids the need for a TCP-style pseudo-header while calculating the checksum. As long as the destination host identifier is meaningful in an Address Segment, the receiving host can check that the destination host identifier from the FIRST packet does, indeed, match this host's identifier. Note that this protection is lost if `NOCHECK` is on in a FIRST packet.

2.4.1.2 Address Domain Field

The *adomain* field is an address demultiplexer. Some address formats are actually used by two or more addressing domains. The *adomain* value disambiguates which address domain is being used.

Example

UDP, TCP, XTP, and any other protocol recognized by IP (has a protocol number) use the same address format, but the port numbers are allocated from separate port spaces. Port 155 in UDP is different from port 155 in TCP. To disambiguate them, the *adomain* field would carry the address format-specific demultiplexer. In this case, *adomain* would be 6 for TCP, 17 for UDP, and 36 for XTP.

2.4.1.3 Address Segment Length Field

The *alen* field specifies the length of the whole Address Segment, including the 4-byte descriptor. The Address Segment must be a multiple of eight bytes, so the minimum value for *alen* is eight bytes.

2.4.1.4 Address Field

The *address* field holds the addresses for the source and destination endpoints. The *address* field syntax is determined by the *aformat* field; the list of possibilities are given in Table 2-4.

Implementation Note

Often there is a mapping from the address format used in the Address Segment to the addressing information needed by the underlying data delivery service. For example, if XTP is running over IP, and *aformat* `0x01` (Internet Protocol Address Format) is used, the mapping is trivial. If XTP is running directly over Ethernet, and *aformat* `0x01` is used, the mapping may require an ARP lookup.

Null Address Format

The Null Address Format, *aformat* `0x00`, is designed to be used in embedded systems where matching a FIRST packet to its listening context is a trivial matter. An example is a dedicated point-to-point link carrying packets for a single pair of communicants.

The syntax for the Null Address Format is shown in Figure 2-12. The *null* field is four bytes to preserve alignment. The value carried in the *null* field must be zero.

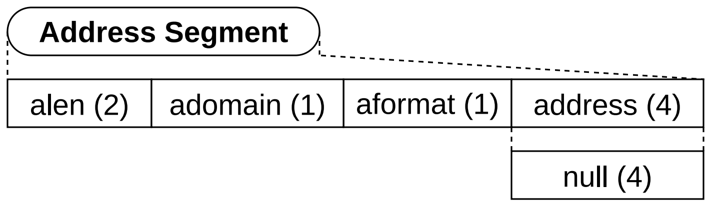

Figure 2-12 — Null Address Format

Internet Protocol Address Format

The Internet Protocol Address Format, *aformat* `0x01`, is shown in Figure 2-13. The 12-byte *address* field contains the destination and source IP addresses (*dsthost* and *srchost*) and the destination and source port numbers (*dstport* and *srcport*).

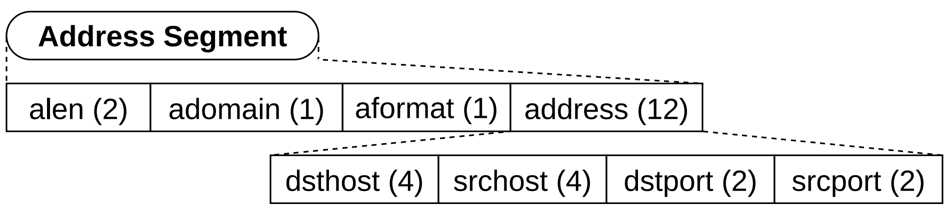

Figure 2-13 — Internet Protocol Address Format

The *dsthost* and *srchost* fields are each 4-byte IP addresses as defined by RFC 791. The escape code for extended addressing mode, undefined in IP, is explicitly disallowed in XTP. The 2-byte *dstport* and *srcport* fields are socket numbers.

ISO Connectionless Network Layer Protocol Address Format

The ISO Connectionless Network Layer Protocol Address Format, `0x02`, specifies an NSAP-address and T-selector value for each endpoint's address, as shown in Figure 2-14. In this format, destination and source network service access point (*dstnsap* and *srcnsap*) 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 20 octets, the NSAP must occupy the most significant bytes of the *dstnsap* and *srcnsap* fields with zeros for padding (*dstnsap* and *srcnsap* are left justified). The length of the transport layer selector fields for the destination (*dsttsel*) and originator (*srctsel*) are given by the values in the *dsttsellen* and *srctsellen* fields. Although T-selector addresses could potentially be longer than 32 octets, the current maximum defined for ISO 8073 (Connection Oriented Transport protocol) is 32 octets. The *pad* field is used to ensure that the Address Segment ends on an 8-byte bound-

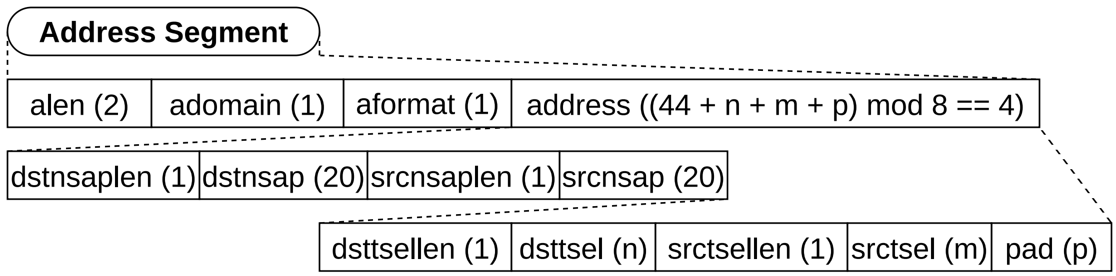

Figure 2-14 — ISO Connectionless Network Layer Protocol Address Format

ary; its contents must be zero. The size of the *address* field, modulo 8, must equal 4 so that its length, when added to the address descriptor, is a multiple of 8 bytes in length.

Xerox Network System Address Format

The Xerox Network System Address Format, `0x03`, is shown in Figure 2-15. The *dstnet* field uniquely identifies the destination network. The *dsthost* field identifies a specific host. The *dstsocket* field identifies the destination socket, and represents the service access point to the application. The source information is given in a symmetric 12-byte structure. The contents of the *pad* field must be zero.

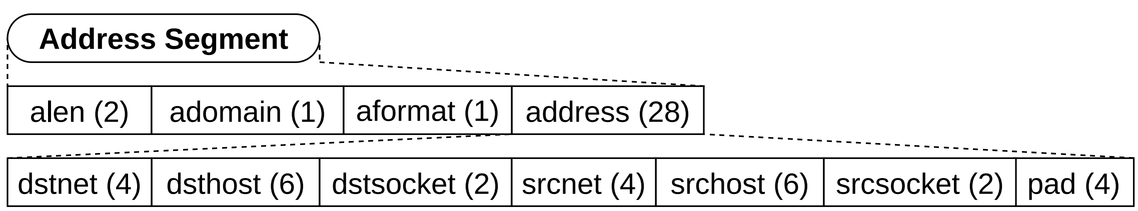

Figure 2-15 — Xerox Network System and IPX Address Formats

IPX Address Format

The syntax for the IPX Address Format, `0x04`, is the same as XNS, shown in Figure 2-15. The *dstnet* field uniquely identifies the destination network. The *dsthost* field identifies a specific host. The *dstsocket* field identifies the destination socket, and represents the service access point to the application. The source information is given in a symmetric 12-byte structure. The contents of the *pad* field must be zero.

Local Address Format

Address format `0x05` 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.

Internet Protocol version 6 Address Format

The format of *aformat* `0x06`, Internet Protocol version 6 Address Format, will follow the IETF definition for the address structure employed by IP version 6.

2.4.2 Data Segment

The Data Segment, show in Figure 2-16, is a variable-length segment. Data Segments must consume sequence space so that error control, which is based on sequence numbers, can be applied. Data Segments are intended for transmission of higher-layer application (user) data. Two packet types can include Data Segments: FIRST and DATA.

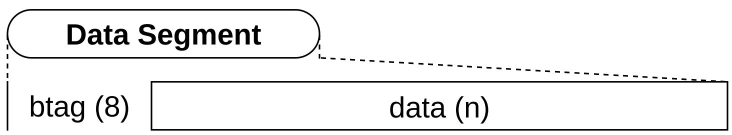

Figure 2-16 — Data Segment Fields

2.4.2.1 Data Field

The *data* field of the Data Segment is a variable length field that holds user data. The contents of this field are not interpreted by XTP.

2.4.2.2 Beginning Tag Field

The first eight bytes of a Data Segment may be marked, or "tagged," for special use by higher-layer applications. These bytes are known as the *btag* field. The options bit `BTAG` indicates the presence of the *btag* field; when the `BTAG` bit is not set, the *btag* field is not present. The *btag* field is opaque to XTP, meaning that XTP transmits the contents of the *btag* field but does not look inside or interpret it. The *btag* field is intended to support higher-layer applications such as encapsulation and convergence protocols.

Application Note

The *btag* field provides a means for applications to mix control information with data while avoiding the imposition of another layer of framing within the Data Segment. Here are three examples.

An application wishing to send a mix of audio, video, and text frames could use the *btag* field to indicate what type of frame this packet contains. The receiving application steers the packet according to the value in the *btag* field.

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.

2.4.3 Diagnostic Segment

A Diagnostic Segment is used in the DIAG packet type to convey information that is not necessary for the correctness of the protocol, but is useful either to the protocol or some other agent (possibly the user). The format for the Diagnostic Segment consists of the following three fields: *code*, *value*, and *message*, as shown in Figure 2-17. The *code* and *val* fields are 32-bit unsigned integers, and the *message* field is a variable-length string. The Diagnostic Segment does not consume sequence number space.

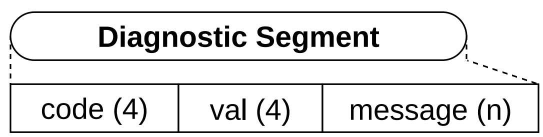

Figure 2-17 — Diagnostic Segment Fields

2.4.3.1 Diagnostic Code Field

The *code* field specifies a type or category of error that caused the generation of the DIAG packet. The *code* field is required and must have a meaningful value. The *code* specifies the situation that caused the generation of the DIAG packet. The *code* values defined are discussed in Section 4.6.4, “Error Notification.”

2.4.3.2 Diagnostic Value Field

The *val* field is defined to more finely specify the type or category of error. A *val* value can modify the interpretation of the *code* value, or it can give additional information. For example, a *code* value may be given the meaning “invalid context,” and the *val* value can specify why the context was invalid.

Like the *code* field, the *val* field is required and must have a meaningful value. The *val* values defined for DIAG packets are discussed in Section 4.6.4, “Error Notification.”

2.4.3.3 Diagnostic Message Field

The *message* field is an optional variable-length string. Receivers are not required to interpret this field. The *message* string gives the textual interpretation of the *code* and *val* values in a form more conducive to giving to the user or recording in a log. No user data may be included in this field.

3 Packet Types

The seven packet types in XTP provide the mechanism for information exchange between endpoints in an association. This information consists of protocol and user data. Certain packet types are used exclusively to exchange protocol state information. These packets are generically called *control packets*. The other packet types are used to carry information, either in the form of user data or some non-state information. These packets are called *information packets*.

The packet type is determined by the *pformat* field in the *cmd* field of the header. FIRST packets initiate an association. Subsequent data is transferred with the DATA packets. Flow control and other state information required often during an association are conveyed via the CNTL packet. ECNTL packets add error control information to the control information, and TCNTL packets add traffic control information. DIAG packets are used to notify the recipient of error conditions at the sending end-system.

The header syntax is common to all packets, and the fields within the header are usually parsed and interpreted in the same way for each packet type. The text will call out situations where this is not true.

In general, the *key* field is used to map the packet to the appropriate context. The *cmd* field carries options for this packet and the modes for the association, as well as the packet type identifier and the protocol version number. The *dlen* field is the length of the payload segment, which is anything that follows the header. The *check* field contains the checksum over the whole packet unless the `NOCHECK` bit is set in the *options* field, in which case the *check* field contains the checksum over only the header fields. When the `SORT` bit is set, the *sort* field represents the priority of the packet, otherwise this field must be zero. The *sync* field contains the count of the number of packets with `SREQ` set sent from the context sending this packet. The *seq* field, last of the header, indicates the next byte expected on the sender's outgoing data stream for control packets, or the first byte in the information segment for data-bearing packets.

3.1 FIRST Packet

The syntax for a FIRST packet is given in Figure 3-1. The FIRST packet carries all of the information necessary to find a listening context at a destination host, and establish an association between that context and the sending context. To effect this, the FIRST packet includes an address specification. Once a listening context is found, the traffic specification in the FIRST packet is examined to determine if this listening context can support the type of traffic specified. If the listening context can support this traffic, the data in the FIRST packet, if any, are given to the context to be delivered to the user. If not, an appro-

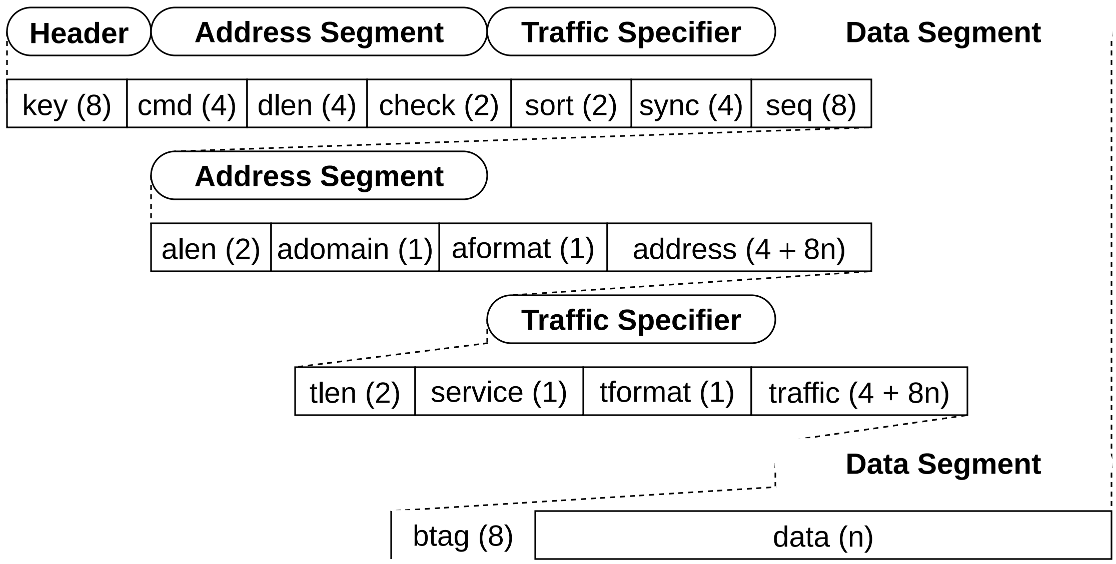

Figure 3-1 — FIRST Packet Syntax

priate DIAG packet is sent (see Section 3.7, "DIAG Packet," for a description of the diagnostic codes for the various error situations).

The entire payload of the FIRST packet is part of the sender's outgoing data stream, so every byte after the header is assigned a sequence number. The *dlen* field indicates the number of bytes in the payload.

Since data streams in both directions begin with sequence number zero, the *seq* field in the FIRST packet is used to specify the initial allocation value to be used for the return data stream.

The Address Segment must be matched against the address filter of each of the listening contexts at the destination host. Section 2.4.1.4, "Address Field," gives the syntax of each of the address formats. Section 4.2.1, "FIRST Packet Matching," describes how a FIRST packet is matched with a listening context.

The Traffic Segment carries traffic shaping parameters. The FIRST packet carries the "offer," including what values the sender can maintain and what values the sender wishes to see in return. The receiver of the first packet must decide if the traffic specification can be met. There are three possibilities: either the traffic specification can be met outright, or it is within an acceptable range, or it cannot be met at all. These possibilities follow the rules of traffic negotiation described in Section 4.2.5, "Traffic Specification Negotiation."

A FIRST packet does not have to carry user data, but if it does, there is no limit on the amount carried except as dictated by the underlying data delivery service MTU. If the `BTAG` bit in the *options* field is set, the first eight bytes of the Data Segment is the *btag* field and its contents is tagged as out-of-band data.

A FIRST packet is intended to be used only once during the lifetime of the association, assuming it or its acknowledgement is not lost. The address is needed only once; subsequent packets will use the *key* field to identify the proper recipient.

3.2 DATA Packet

The syntax for the DATA packet is shown in Figure 3-2. After establishment of the association, subsequent data transfers in both directions use the DATA packet. The *seq* field indicates the beginning sequence number for the data contained in the Data Segment. The *dlen* field gives the length of the data. Every byte in the Data Segment is counted toward consuming sequence space.

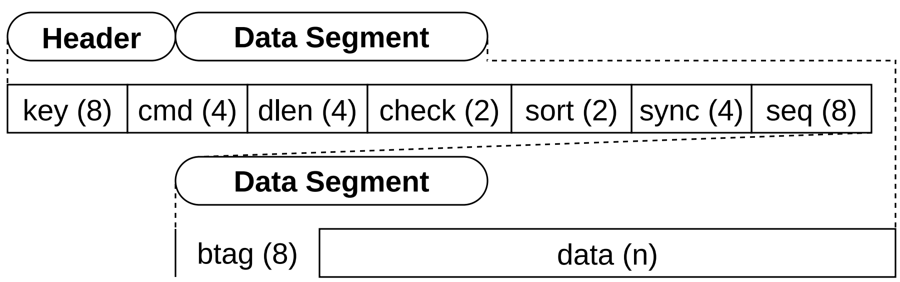

Figure 3-2 — DATA Packet Syntax

As with the FIRST packet, the `BTAG` bit in the *options* field indicates the presence of the *btag* field.

3.3 CNTL Packet

The common control packet is the CNTL packet, shown in Figure 3-3. The *seq* field holds the sequence number of the next untransmitted byte to be sent on the outgoing data stream of the sender of this packet. The CNTL packet, however, does not consume sequence space. The receiver of the CNTL packet can use the *seq* value to compare it with the next sequence number expected on its incoming data stream (if `FASTNAK` is set, this comparison will indicate the need to generate an ECNTL packet as a fast negative acknowledgement).

The *rseq* and *alloc* fields are used to convey flow control information for the sending context's incoming data stream. The *rseq* field holds the highest sequence number for

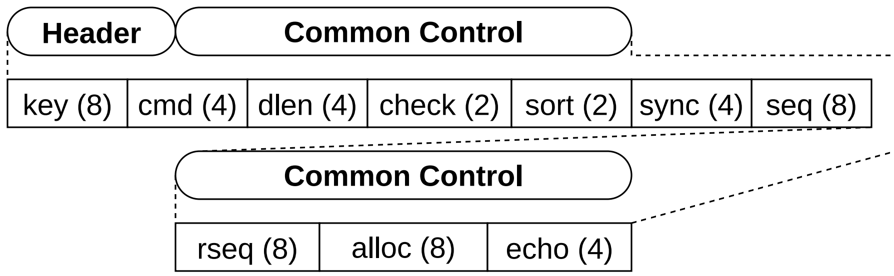

Figure 3-3 — CNTL Packet Syntax

data received without gaps. It therefore represents the lower edge of the flow control window. If the `NOERR` bit is set, gaps are ignored, so the value in the *rseq* field is the highest sequence number yet seen on the incoming data stream.

The *alloc* field represents the upper edge of the flow control window. The value it holds is one past the highest sequence number acceptable on the incoming data stream.

The *echo* field carries the highest *sync* value yet seen in any incoming packets. The act of echoing this monotonically increasing number back to its originator gives the originator some idea of how synchronized the endpoints are. The details of the synchronizing handshake procedure are discussed in detail in Sections 4.6.3 and 5.4.2.

A CNTL packet is not used if an error in the data stream is encountered; ECNTL packets are used under these conditions.

A CNTL packet may be generated at any time, but there are several specific conditions that cause a CNTL packet to be generated. While the data stream is error free, a CNTL packet is generated in response to the following conditions:

- the `SREQ` bit is set in any incoming packet, except during traffic specification negotiation (a TCNTL is used)
- all of the data received prior to the arrival of a set `DREQ` bit have been delivered to the user
- the `EDGE` bit value in this packet is different from the one in the most recently received packet

If more than one of the above conditions exist, a single CNTL packet can be sent.

3.4 ECNTL Packet

The ECNTL packet is used to convey error control information as well as common control information. The syntax for an ECNTL packet is given in Figure 3-4. The *rseq*,

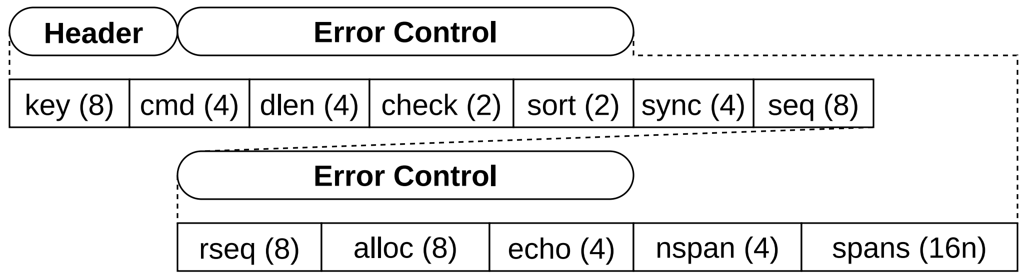

Figure 3-4 — ECNTL Packet Syntax

alloc, and *echo* fields are interpreted just as in a CNTL packet. The additional fields, *nspan* and *spans*, provide the sequence numbers of any non-contiguous data received on the incoming data stream of the sender of this packet.

The *nspan* field indicates the number of pairs of sequence numbers held in the *spans* field. A pair of sequence numbers, or span, indicate the data that have been correctly received. The gap information, which can be determined from the *spans* field, allows this packet's receiver (the data stream's transmitter) to selectively retransmit only the missing bytes of data. The first gap is from *rseq* to the first sequence number in the first span in the *spans* field. This spans information can be ignored, in which case go-back-*N* retransmission is done by retransmitting from *rseq*. The procedures for use of the ECNTL packet for error control are given in Section 4.6.2, "Acknowledgements and Retransmission."

An ECNTL packet is sent under the following conditions:

- when a CNTL or TCNTL packet should be sent but errors exist in the incoming data stream
- if the `FASTNAK` bit is set for the incoming data stream and a gap is encountered

If both of the above conditions exist, a single ECNTL packet can be sent to satisfy both conditions.

3.5 TCNTL Packet

The TCNTL packet is used to convey and negotiate traffic shaping information as well as common control information. The syntax for a TCNTL packet is given in Figure 3-5. The *rseq*, *alloc*, and *echo* fields are interpreted just as in a CNTL packet. The additional fields of the Traffic Control segment provide a mechanism for the key exchange algorithm (*xkey* field) and parameters for traffic shaping (the Traffic Specifier). The traffic shaping information is held in the *traffic* field, whose format is determined by the value of the *tformat* field, and whose length is given by the *tlen* field. The formats for the Traffic Specifier segment are described fully in Section 2.3.4

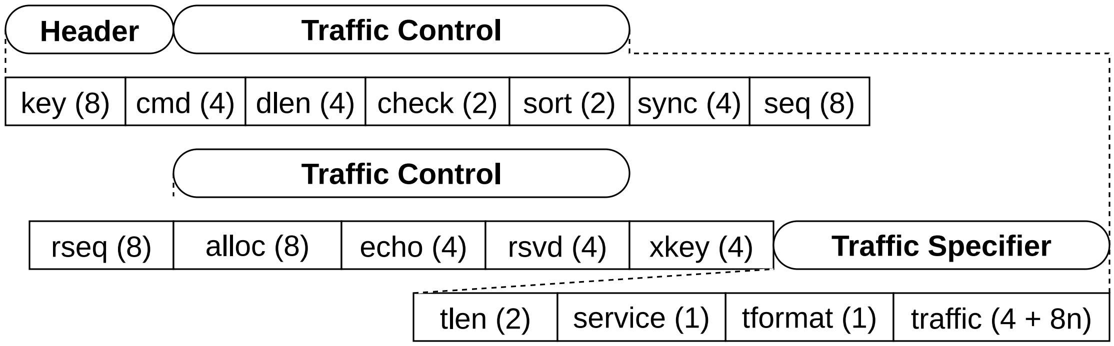

Figure 3-5 — TCNTL Packet Syntax

A TCNTL packet may be generated during an active association when one of the following conditions exists:

- an endpoint wants to modify the current traffic specification
- an endpoint wants to respond to a traffic specification modification request
- an endpoint wants to perform a key exchange

The first and second condition follow the rules for traffic negotiation set forth in Section 4.2.5, “Traffic Specification Negotiation.” The first condition occurs when the context gets new traffic information from the network or new traffic requirements from the user. A TCNTL packet is generated with the new traffic specification, and sent to the other endpoint, opening a traffic specification negotiation. This TCNTL must have the `SREQ` bit set. A traffic specification negotiation must be delayed if a synchronizing handshake is in progress, until after the completion of the synchronizing handshake.

The second condition is a response to a suggestion for a (possibly new) setting for the traffic parameters. When a FIRST packet or TCNTL packet is received, the receiver checks the suitability of the parameters. If the FIRST or TCNTL packet carries a set `SREQ` bit, the receiver responds with a TCNTL (if there are no errors in the data stream), possibly modifying the parameters (or the receiver may reject a suggested traffic specification outright with a DIAG packet). A TCNTL packet generated under this second condition must not carry a set `SREQ` bit.

The third condition arises from the key exchange procedure (Section 4.2.4). Unless the receiver of the FIRST packet tells the originator what return key to use, the abbreviated context lookup procedure (Section 4.2.3) cannot be used for packets sent from the originator. The *xkey* field of the TCNTL packet is used to convey this return key value back to the originator. A key exchange can happen only once during the lifetime of the association; prior to and after this exchange, the *xkey* field must contain a value of zero.

When a TCNTL packet with the `END` bit set is to be sent, a CNTL packet with the `END` bit set may be sent instead.

3.6 JOIN Packet

A JOIN packet has the syntax shown in Figure 3-6. JOIN packets are used to join an in-progress multicast conversation. See Section 5.3.4, “Joining an In-Progress Multicast Association” for details. The JOIN packet does not consume sequence number space.

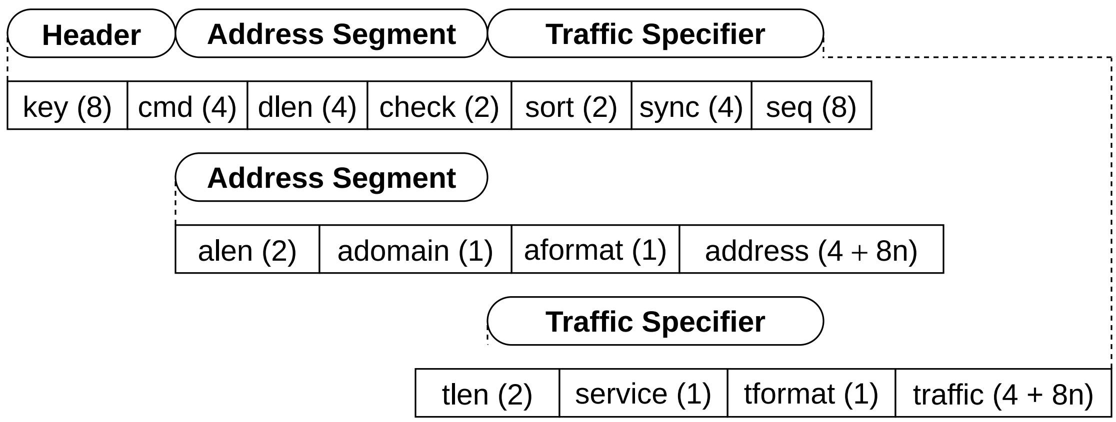

Figure 3-6 — JOIN Packet Syntax

The JOIN packet consists of an Address Segment and a Traffic Specifier segment. The Address Segment specifies the address of the multicast group that the JOIN packet’s sender wishes to join. The Traffic Specifier indicates the shape of the traffic that the sender requests or is willing to offer.

A JOIN packet is generated under one of the following conditions:

- a context wishes to join an in-progress multicast association as a receiver
- a multicast transmitter accepts a receiver into the multicast group

When a JOIN packet is used to attempt to join an in-progress multicast association, the *key* field must carry the value zero, and the *seq* field must contain the requesting context’s return key value. This special use of the *seq* field allows the requesting context to convey its identity to the multicast transmitter. The Address Segment contains the requestor’s source host identifier as the source address, and the multicast group address as the destination address. The Traffic Specifier contains the traffic shaping parameters that the requesting context is willing to accept (multicast associations are simplex, so there will be no traffic shaping information necessary for the data stream from the requesting context to the multicast transmitter). This packet is multicast, via the underlying data delivery service, to the multicast group address. Only the multicast transmitter is allowed to respond to a JOIN packet having a value of zero in its *key* field.

The multicast transmitter, upon accepting the request to join the in-progress multicast association, issues a JOIN packet back to the context making the request. (The method for rejecting the request is given in Section 5.3.4, “Joining an In-Progress Multicast Association.”) The multicast transmitter places its *key* value in the *key* field (note that this is not a return key), and the current sequence number for the association in the *seq* field. The multicast transmitter also fills in the Traffic Specifier with the values it will use on the multicast data stream. The requesting context, upon receipt of this JOIN packet, maps the source address and the *key* value to the requesting context for use in the full context lookup procedure (see Section 5.3.6). Upon completion of this JOIN packet handshake, the requesting context becomes a member of the multicast receiver set.

3.7 DIAG Packet

DIAG packets are used to report pathological conditions that are either fatal or which require corrective action. The format for a DIAG packet is given in Figure 3-7. The *code* field indicates the major error category, and the *val* field modifies that category with more specific information. The *message* field is not parsed by XTP, but may be written to a log file or given to the user. The values for the *code* and *val* fields are given in Table 4-1 and Table 4-2. The circumstances governing DIAG packet generation are given in Section 4.6.4, “Error Notification.”

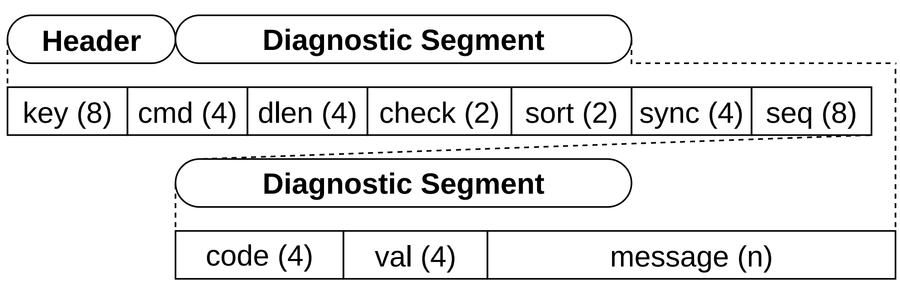

Figure 3-7 — DIAG Packet Syntax

A DIAG packet’s *key* field identifies the context to receive the DIAG packet. If a DIAG packet cannot be delivered to a context, the packet is dropped. This is to avoid a “storm” of DIAG packets sent back and forth, each reporting an error condition the previous DIAG created. Only the `NOCHECK` bit is meaningful in the *options* field of the DIAG. The *dlen* and *check* fields must contain meaningful values. All other fields the header must reflect the values from the header of the packet that caused the DIAG to be generated, if one did. Otherwise, the other fields must contain zero.

DIAG packets are generated when some error condition exists. In general, the error conditions are caused by:

- the failure to deliver a packet to the destination context
- a change in the maximum allowable packet size imposed by the underlying data delivery service
- notification of demise of a host

A DIAG packet can never have a set `SREQ` bit. Consequently, it is not protected by the WTIMER, so a lost DIAG packet cannot readily be detected. For this reason, the protocol is designed to recover from any of the errors signalled by the DIAG packet by relying on timers to expire.

4 Unicast Functional Specification

XTP unicast provides a high degree of functionality through orthogonal protocol mechanisms. These mechanisms are in the form of fields and bitflags used during packet exchanges over the lifetime of an association. Association management procedures define how these fields and bitflags are used during the lifecycle of the association. The major protocol procedures — flow control, rate control, and error control — are independent and configurable.

This section first defines the fundamental concepts in XTP, including the context and association state machines. Next, the procedures for unicast association management are discussed, complete with unicast association establishment and termination. There are several termination semantics, and these are given in detail. The last three subsections describe the particulars of the flow, rate, and error control procedures. A similar discussion for multicast functional specification is given in Section 5.

4.1 Protocol Fundamentals

Processes. Conceptually, an XTP implementation consists of four processes: a *receiver* process, a *sender* process, a *reader* process, and a *writer* process. The receiver and sender processes provide XTP's interface to the underlying data delivery service. The reader and writer processes provide XTP's interface to the users (applications). Their relationship to one another and to associated structures is shown in Figure 4-1. The following description of these processes is not intended to imply implementation requirements; these processes are used in the protocol description to compartmentalize the responsibilities of the protocol procedures.

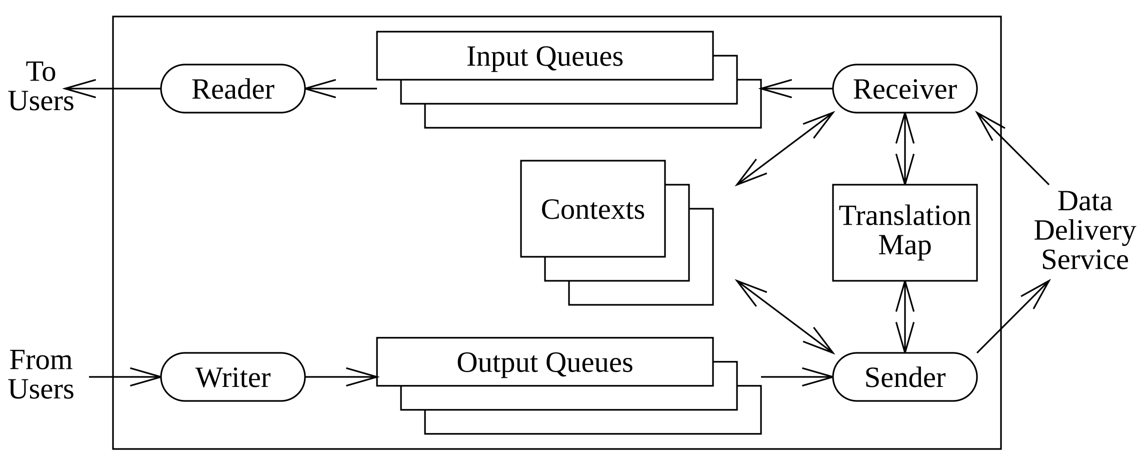

Figure 4-1 — XTP Host Architecture

The receiver process acts on packets received from XTP's underlying data delivery service, directing the packets to the appropriate contexts. A packet is matched to its context by using some packet header information to index into the *translation map*. Once found, the context parses the packet's control information. If the packet is a data-bearing packet, the data are placed directly into input queues pending commitment from the context. When the context is assured that the data are valid, it commits the data to the reader process. The users then access this data through the reader process.

The writer process acts upon output commands from the users, placing data into the output queues and informing the appropriate contexts of the data's presence. The contexts construct packets and give them to the sender process. The sender process pulls data from the output queues into the outgoing data-bearing packets, and gives the data and control packets to the underlying data delivery system.

Context State Machine. Figure 4-2 shows the state diagram for a context. All contexts in an implementation exist in a quiescent state until they are activated. When a user submits an input command, a quiescent context moves to the listening state. This must occur some time before another user at a remote host issues an output command. The output command causes a quiescent context at this remote host to move directly into the active state, and a FIRST packet to be generated and sent. When this FIRST packet is received by the context in the listening state (and all conditions are met for accepting this FIRST packet), the listening context moves into the active state. The contexts at both endpoints are now active, and data transfer may occur in both direction for an arbitrary length of time. When the contexts are finished sending and receiving data, they each move into the inactive state. When one of the contexts issues a packet with the `END` bit set, the association is terminated and both contexts return to quiescent states.

Association State Machine. The state diagram for an association is shown in Figure 4-3. This diagram is from the point of view of one of the contexts in the association. The association state machine begins with both the incoming and outgoing data streams in the

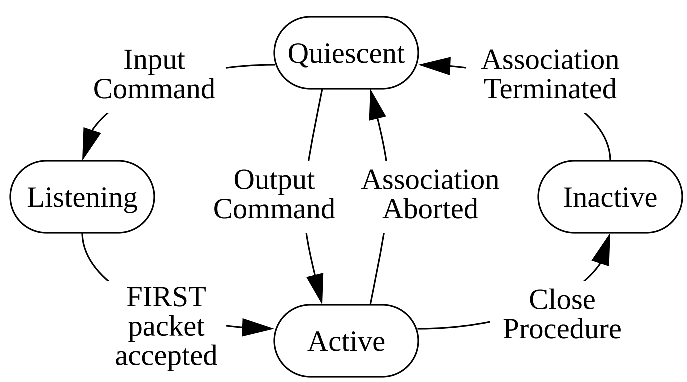

Figure 4-2 — Context State Machine

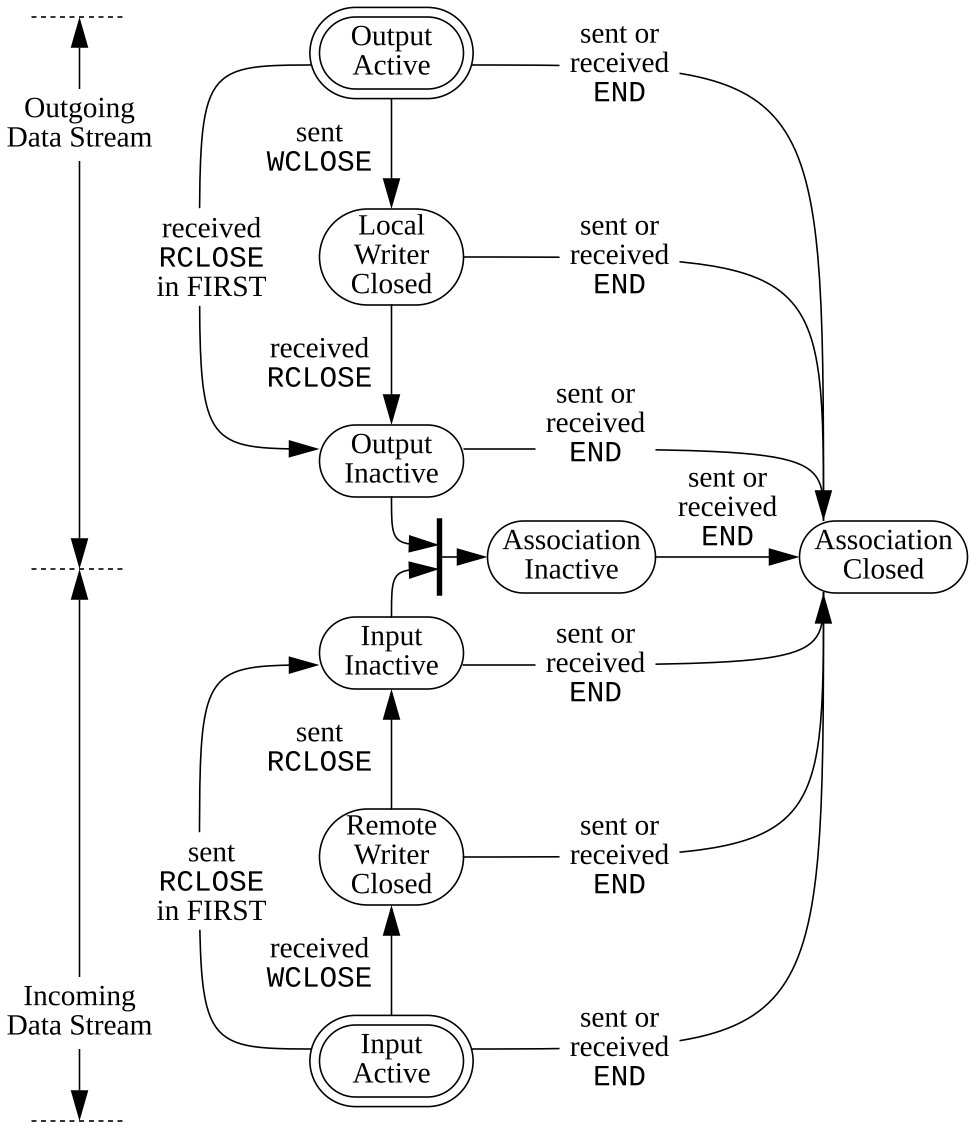

Figure 4-3 — Association State Machine

active state. The dual nature of this state machine reflects the fact that an association controls two data streams. As data flows in both directions, both data streams remain in active states.

The outgoing data stream side moves from output active to local writer closed when a packet is sent with the `WCLOSE` bit set. The local writer process may not transmit any new data (no sequence space can be consumed from this point on), but previously transmitted data may be retransmitted. When a packet is received with its `RCLOSE` bit set, the outgoing data stream moves from local writer closed to output inactive. Receiving an `RCLOSE` bit implies that all data have been received to the satisfaction of the receiver, so no further DATA packets whatsoever can be sent on the outgoing stream.

Independently, the incoming data stream moves from input active to remote writer closed when a packet with the `WCLOSE` bit set is received. This tells the context that no new data will be arriving on this data stream. When all data have been received, according to the error control parameters, a packet with the `RCLOSE` bit set is sent. This moves the incoming data stream into the input inactive state.

Either data stream may go inactive first, but once both data streams have gone inactive, they both move into the association inactive state. This is equivalent to the context inactive state. When one of the endpoints sends a packet with an `END` bit set, the association moves from inactive to closed, which is the condition for the context state machine to move from inactive back to quiescent.

Data Streams. A data stream is an abstraction for an arbitrary length string of sequenced bytes, where each byte is associated with a *sequence number*. The *sequence space* for a data stream starts at zero and continues indefinitely. Only specific fields in XTP packets consume sequence space. Sequence space is conceptually monotonically increasing, but due to finite sequence number representation, "rollover" may occur such that one byte may be further into the sequence space than another but have a smaller sequence number. Implementations must ensure that sequence numbers always represent relative positioning within the sequence space.

Sequence Numbers. Sequence numbers provide the basis for flow control and error control between XTP endpoints. 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 pairs of sequence number, called *spans*, that delineate portions of the sequence space.

Buffers. 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 noncontiguous 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.

Packets. A *packet* is the data transfer unit defined by XTP. A buffer may fill several packets since there is no restriction on buffer size. A *message* consists of one or more buffers. An XTP packet is contained, or encapsulated, within one or more lower layer frames. This lower layer is the *underlying data delivery service* for XTP. XTP is designed to be independent of lower layer influences, but it does depend on lower layer addresses and expects a validity check from the lower layer.

Timers. There are several timers that are used or maintained during the lifetime of an association. The WTIMER is used to bound the amount of time a context will wait on a response to a status request (a set `SREQ` bit in any sent packet). The CTIMER is a long-duration timer used to generate keep-alive packet exchanges. The CTIMEOUT timer bounds the amount of time an endpoint will try to reestablish the association before giving up. The RTIMER is the rate control timer, governing the length of time between bursts of data.

4.2 Association Management

An association is established when a listening context receives a FIRST packet, and both this and the initiating context have moved into the active state. A received FIRST packet is matched against all listening contexts to find one that will accept the incoming data stream. This is described in the section called "FIRST Packet Matching." The "Full Context Lookup" and "Abbreviated Context Lookup" sections describe mapping incoming packets to the appropriate contexts. The "Key Exchange" section describes an optional optimization that eliminates the need for full context lookups for incoming packets. When there is no longer a need for the association, the association is closed using procedures described in "Association Termination."

4.2.1 FIRST Packet Matching

A received FIRST packet, if it is not discovered to be a duplicate for an already active context, is subjected to a matching algorithm to determine which listening context, if any, should get the FIRST packet. Each listening context submits an *address filter* that represents the values of an address that the context is willing to accept, as well as acceptable traffic shaping parameters and options bits. The FIRST packet's contents are compared against each listening contexts' criterion for acceptance until either a match is made or all listening contexts have been examined.

Upon receipt of a FIRST packet, the receiving host takes the following steps:

1. If a full context lookup (see Section 4.2.2) on this FIRST packet finds an active context, the FIRST packet is a duplicate; the context to which this packet belongs should respond to the `SREQ` and `DREQ` bits, if set, and accept any additional data carried within, but no new context becomes active.
2. If the FIRST packet is not a duplicate, the receiving host performs a series of comparisons to find an appropriate listening context:
   - The *address* field within the Address Segment of the FIRST packet is matched against a set of *filters*.

- The *service* field in the Traffic Specifier segment is matched against the context's *service* value: if the context's *service* value is 0, any *service* field value will match; otherwise the values must match exactly.
- The traffic specification values in the Traffic Specifier segment are matched against the traffic specification acceptable by the context.
- The settings of the option bits in the *options* field of the header of the FIRST packet are examined for acceptability.

3. If the FIRST packet's address, traffic specification, and options are acceptable, the FIRST packet is given to this listening context.
4. Otherwise, the matching algorithm moves on to the next listening context, and the comparisons in Step 2 are repeated.

If all of the listening contexts are examined but no acceptable match is made, the FIRST packet is rejected by using a DIAG packet whose *code* value is 1, "Context Refused." There is a hierarchy of rejection reasons, as indicated by the *val* value:

1. failure due to the Address Segment
   - *val* 1: "No listener"
   - *val* 3: "Address format not supported"
   - *val* 4: "Malformed address format"
2. failure due to the *service* field
   - *val* 8: "No provider for service"
3. failure due to the Traffic Segment
   - *val* 5: "Traffic format not supported"
   - *val* 6: "Traffic specification refused"
   - *val* 7: "Malformed traffic format"
4. failure due to the options bits
   - *val* 2: "Options refused"

If none of these reasons apply, the rejection should be done using a DIAG packet with *code* value 1 and *val* value 0, "Unspecified."

The listening context is allowed to reject FIRST packets if the options are not acceptable. In particular, `NOCHECK`, `NOERR`, `MULTI`, `RES`, `SORT`, `NOFLOW`, `FASTNAK`, and `RCLOSE` are options whose values, set or cleared, may be grounds for rejecting the FIRST packet.

Implementation Note

How the user specifies which *options* bits are acceptable and which are not is strictly an issue for the API. However, a useful concept is a *yes_mask* and a *no_mask*. The *yes_mask* specifies the set of bits that must be set. The *no_mask* specifies the set of bits that must not be set. The following logic formula will indicate if the packet should be rejected:

(((yes_mask | options) – options) || (no_mask & options))

If the *address* and *service* fields match a context, and the Traffic Specifier and *options* values are acceptable, the FIRST packet is accepted. An entry for this context is made in the translation map that will map to this context any incoming packets whose *key* field is the same as this FIRST packet's *key* field, and whose source host's address (obtained from the underlying data delivery service) is the same as this FIRST packet's source host's address. This entry is fundamental in the full context lookup, described next. The packet is given to the found context, which has now moved from the listening state to the active state.

Implementation Note

If addresses from the underlying data delivery service are not unique within a local host, there may be problems with implementing the translation map. This situation is common for hosts which are connected to multiple physical networks, since unique address are usually not required across separate networks. One way to avoid this ambiguity is to construct a host-unique prefix for each underlying data delivery service used by the host, and add this prefix to all addresses obtained from the underlying data delivery service before using those addresses internally.

4.2.2 Full Context Lookup

The full context lookup procedure maps an incoming packet to the appropriate active context if the *key* value in the incoming packet is not a return key (that is, the `RTN` bit is not set in the *key* field of the packet). The translation map structure holds the mapping from packet information to appropriate context. The packet information is used as an index into the translation map to retrieve the handle of the context.

The *key* field of an incoming packet is checked to determine if it is a return key or a normal *key*. If it is a return key, the abbreviated context lookup procedure, described next, is used. If the *key* value is not a return key, that *key* value and the packet's source host's address (obtained from the underlying data delivery service) are used as a pair to index into the translation map. If a FIRST packet with this *key* value has been received from this packet's source host, a mapping will exist in the translation map. If not, the lookup will fail, and a DIAG packet (containing the *code* for “Invalid Context”) should be returned to this packet's sender.

4.2.3 Abbreviated Context Lookup

The abbreviated context lookup procedure is an optimized method for mapping an incoming packet to the appropriate context without using the translation map. By definition, a *key* value is generated by a host to be unique within that host. Then the *key* value is placed into the *key* field of the FIRST packet. When the FIRST packet is received and given to the matching listening context, that context notes the FIRST packet's *key* value, sets its `RTN` bit, and uses this value as the return key. The return key value is placed in any packets sent in the return direction to the host that sent the FIRST packet.

If the *key* field of a received packet holds a return key, the context manager knows immediately that the *key* value from which this return key is derived was generated at this host. Since the *key* value is unique within this host, there is a direct mapping from this *key* value to the context for whom the *key* value was generated (e.g., it could be an index into an array of contexts). The abbreviated context lookup procedure is the implementation-dependent method for making this direct mapping.

4.2.4 Key Exchange

Packets sent in the return direction are matched with their intended context via the abbreviated context lookup procedure, but packets sent in the forward direction cannot be matched this way because these packets do not carry return keys. The key exchange procedure is the method by which the context that received the FIRST packet can tell the context that sent the FIRST packet what return key value to use, so that both sides can use the abbreviated context lookup procedure.

At some point during the association, but usually in response to the first `SREQ` received from packets travelling in the forward direction, a key exchange can occur. The key exchange procedure uses the *xkey* field in a TCNTL packet. The context that received the FIRST packet places into the TCNTL packet that context's *key* value with the `RTN` bit set. This is that context's return key value. The context that sent the FIRST packet, upon receipt of this TCNTL packet, must note the value of the *xkey* field and must use this value as the *key* value in all outgoing packets from that point onward.

A key exchange procedure is optional; if it is not performed, a full context lookup will be required for all packets without a return key in its *key* field. If a key exchange is done, however, packets in both directions of the association must carry the appropriate return key in the *key* field. These packets, upon receipt, can be mapped to their proper contexts via the abbreviated context lookup procedure.

4.2.5 Traffic Specification Negotiation

A traffic specification is a contract between the application and the service provider regarding the shape of the data being transferred. There are potentially many parameters to the shape of traffic, including the throughput, maximum size of a burst of data, frequency of individual packets, interpacket spacing, and jitter. There is no canonical list of traffic parameters. Instead, XTP uses a segment, the Traffic Specifier, where a set of traffic specification formats is defined for various situations (see Section 2.3.4, “Traffic Specifier Segment”).

XTP provides a mechanism for negotiating traffic shaping parameter values. The FIRST packet carries a Traffic Specifier suggesting the parameters for the traffic (these values may include what the sender would like to use in the forward direction, and specifying the values that should be used for traffic in the return direction). A TCNTL packet must be generated any time the traffic specification changes during the association.

The traffic specification negotiation is a two-way handshake. The initial packet in the handshake may be either a FIRST packet (the beginning of an association) or a TCNTL packet (subsequent to the establishment of the association). A traffic specification negotiation is delayed if a synchronizing handshake is currently in progress. The first packet in the handshake (FIRST or TCNTL) must have the `SREQ` bit set. When the initiator of the negotiation sends a packet with the `SREQ` bit set, the negotiation is said to be open; the initiator must save the *sync* value sent in this initial packet. The receiver of this initial packet has three choices:

1. reject the traffic specification outright
2. accept the traffic specification outright
3. accept the traffic specification with modifications

If the receiver of the packet chooses to reject the traffic specification, the receiver returns an appropriate DIAG packet (“Request Refused, Traffic specification refused” or “Request Refused, Options refused”). If the receiver chooses to accept or modify the traffic specification, it will respond with a TCNTL packet with the `SREQ` bit cleared.

The traffic specification negotiation is terminated when the initiator of the negotiation receives a TCNTL packet whose *echo* field matches that of the *sync* field from the initial packet of the negotiation. If the WTIMER expires while there is an open negotiation, a synchronizing handshake (Section 4.6.3) using TCNTL packets occurs.

The initiator of the traffic specification negotiation performs the following steps:

1. The Traffic Specifier of a FIRST packet (if this is the initial packet of the association) or a TCNTL packet (otherwise) is loaded with the traffic specification request, the *options* field is loaded with the appropriate options, and the `SREQ` bit is set.
2. The value from the *sync* field from this packet is saved in a variable **saved_sync**.
3. A state variable **tspec_neg_open** is set to true.
4. If a TCNTL packet is received whose *echo* value equals the **saved_sync** value, the negotiation is complete; set **tspec_neg_open** to false and examine the Traffic Specifier and *options* field values for acceptability.
5. If an ECNTL packet is received whose *echo* value equals the **saved_sync** value, and **tspec_neg_open** is true, first satisfy the retransmission requests, then send the Traffic Specifier again in a TCNTL packet, with the `SREQ` bit set; go to Step 2.

6. If the WTIMER expires and **tspec_neg_open** is true, a synchronizing handshake using TCNTL packet is begun; on successful termination of the synchronizing handshake, go to Step 4.

Likewise, the receiver of a traffic negotiation request performs the following steps:

1. Any received packet with the `SREQ` bit set, including a FIRST or TCNTL packet, can cause an ECNTL packet to be generated if gaps in the data stream have been noticed (see Section 3.4); generating an ECNTL under error conditions takes precedence over responding to the negotiation request.
2. If no gaps exist in the data stream, the Traffic Specifier and the *options* field from a received FIRST or TCNTL packet with the `SREQ` bit are examined for acceptability.
3. If acceptable, a TCNTL packet is sent with the `SREQ` bit cleared, *echo* field containing the **rcvd_sync** value, the Traffic Segment loaded with the traffic specification response, and the *options* field set with the appropriate options (if the `END` bit is to be sent in this packet, a CTNL packet may be used instead of a TCNTL packet).
4. If unacceptable, an appropriate DIAG packet is sent, if allowed.

If either the receiver of the request or the receiver of the response does not accept the traffic parameters or the options bits, the traffic specification negotiation is rejected with a DIAG packet.

DATA and CNTL packets may be exchanged during a traffic specification negotiation, but any data sent during the negotiation may be rejected if the data do not meet the negotiated traffic specification.

4.2.6 Changing Modes

Changing the mode bits of the options field changes the characteristics of the association. These bits include `NOERR`, `RES`, `SORT`, and `NOFLOW` (`MULTI` cannot be changed). Either endpoint may reject the change of the mode bits.

Since packet reordering may occur, leading to possibly unpredictable data transfer policies, a conservative method for changing mode bits during an association is to use a traffic specification negotiation, and restrict data transfer until the traffic negotiation is complete.

4.2.7 Association Termination

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. An association is closed by completing the packet exchanges with `WCLOSE`, `RCLOSE`, and `END` bits (see Figure 4-3). Closure is achieved with various degrees of gracefulness, depending on the bits used and the order in which they are set.

The `WCLOSE` bit is set when the sender has completed the transmission of all data in its outgoing data stream. Once the `WCLOSE` bit is set in an outgoing packet, all subsequent outgoing packets, except retransmitted DATA packets, must carry the set `WCLOSE` bit.

There are two cases when the `RCLOSE` bit may be sent in an outgoing packet: (1) the `RCLOSE` bit is sent in the FIRST packet, indicating that the sender's incoming data stream is closed and this association will be simplex, and (2) the `RCLOSE` bit is sent after the receipt of a packet with the `WCLOSE` bit set, and all data on the incoming data stream have been received in accordance with the error control parameters for that data stream. The receipt of an `RCLOSE` bit prior to sending a `WCLOSE` bit is a protocol error. Once set, the `RCLOSE` bit must be carried in all subsequent outgoing packets.

After a `WCLOSE` is received, the error control procedures must ensure that all data on the incoming data stream have been received, according to the error control parameters. Once the error control procedures have been satisfied, the `RCLOSE` bit must be sent in the next outgoing packet.

Design Note

The only two ways one endpoint can cause another endpoint to generate a packet is by setting the `SREQ` or `DREQ` bit, or both, in an outgoing packet. As a consequence, when a `WCLOSE` bit is sent, it is the user's responsibility to ensure that either some packet will be generated at the other endpoint to return the `RCLOSE`, or an error report in the form of an ECNTL packet will be generated. Typically, an `SREQ` will accompany the `WCLOSE` for these reasons.

If an `SREQ` is sent with a `WCLOSE`, and retransmissions were needed, an `SREQ` should be sent with the last retransmitted DATA packet (see Section 4.6.2, "Acknowledgements and Retransmission," for complete details on retransmitting an `SREQ` bit).

The `END` bit indicates the end of the association and is only sent once. The context sending the `END` bit goes quiescent immediately, subject to remaining in a zombie state for connection hazard reasons. The context receiving the `END` bit immediately goes quiescent as well, but is not subject to remaining in a zombie state.

The sender of the `END` bit must wait in a zombie state for a duration of CTIMEOUT. This is to eliminate connection hazards resulting from the case where the packet carrying the `END` bit is lost. Otherwise, if the intended receiver of the `END` bit requests a retransmission, the host sending the packet with the `END` bit set would respond with a DIAG, and the receiver would not know if the connection was lost or finished gracefully. This is avoided by maintaining a zombie context with at least enough state information to respond to a retransmission request. For situations where the hazard will not arise, or the users do not care, the CTIMEOUT can be reset to zero, effectively eliminating the zombie state. If, while in the zombie state, a context receives a packet with the END bit set, the context can immediately become quiescent.

Fully Graceful Independent Close

While XTP does not define specific close semantics, several close handshakes are particularly useful. A fully graceful independent close is shown in Figure 4-4. Context A initiates the close of its outgoing data stream with a packet with WCLOSE set. When all data are accounted for according to the error control parameters for the A-to-B data stream, Context B responds with a packet with RCLOSE set. At this point, the A-to-B data stream is gracefully closed, but the B-to-A data stream remains open. Later, Context B initiates the close of its outgoing data stream by setting the WCLOSE in an outgoing packet. Context A responds accordingly with an RCLOSE. At this point, Context B sends a packet with the END bit set, and the association is terminated.

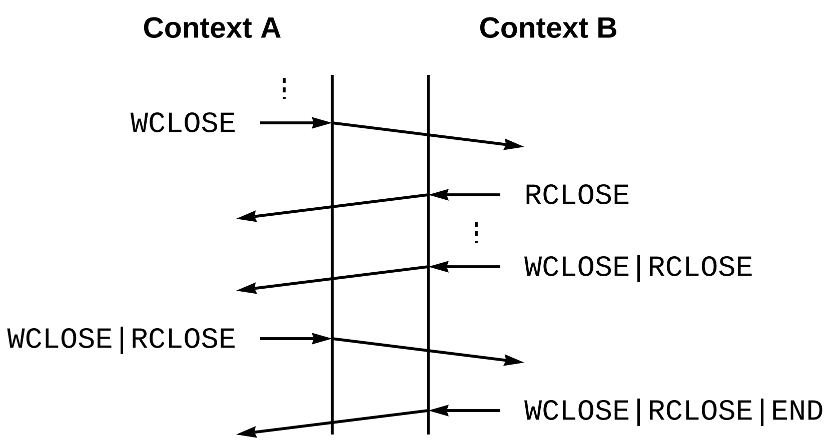

Figure 4-4 — Fully Graceful Independent Close

Abbreviated Graceful Close

Although closing each of the two data streams of an association is an independent activity, the exchange of the packets can be abbreviated by “piggybacking” some of the close bits on the same packet. Figure 4-5 shows how a fully graceful close can be achieved using only three packets. Context A initiates the close by setting the WCLOSE bit in an outgoing packet. The packet carrying the RCLOSE from B to A also carries the WCLOSE for the B-to-A data stream. The third packet carries the RCLOSE for the B-to-A data stream, and the END bit to end the association. Note that this is only graceful if Context A enters the zombie state after sending the END bit.

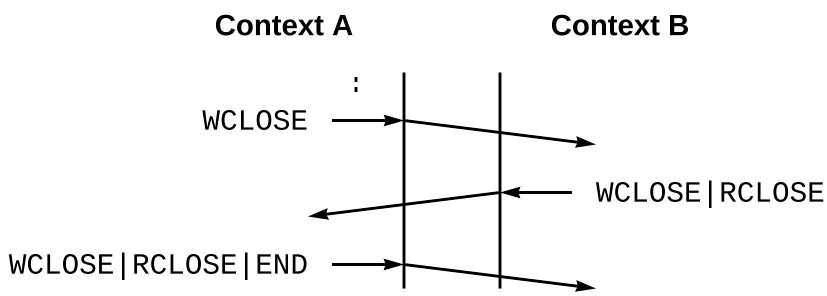

Figure 4-5 — Abbreviated Graceful Close

Forced Close

Figure 4-6 shows how Context A can gracefully close the A-to-B data stream but cause the B-to-A data stream to close ungracefully. The `WCLOSE` bit is set in a packet from Context A to Context B. This initiates the graceful close for A-to-B. The `RCLOSE` in the return packet acknowledges the A-to-B `WCLOSE`. The last packet carries a set `END` bit, ending the association.

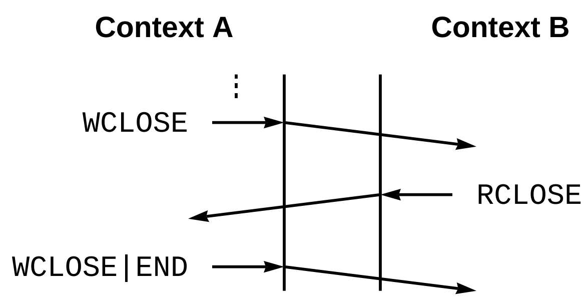

Figure 4-6 — Forced Close

Abortive Close

Sending the `END` bit at any time during the life of the association aborts the association, regardless of the state of each of its data streams. Figure 4-7 shows a single packet carrying the `END` bit.

Close Due To Inactivity

An association is also terminated if no packets are received at one of the contexts for some period of time. The connection timer CTIMER is used to allow XTP to recover from system and network failure by measuring inactivity. The CTIMER is enabled when the association first becomes active. The length of the CTIMER interval must meet the criterion

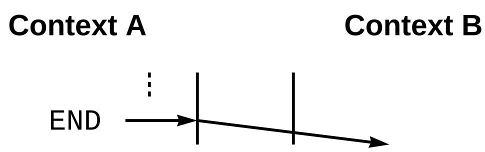

Figure 4-7 — Abortive Close

put forth in the Implementation Note in Section 2.2.1, “Key Field.” The context counts the number of packets that have arrived during the CTIMER interval. If the packet count is greater than zero when the CTIMER expires, the CTIMER is restarted. If the packet count is zero, the CTIMER is restarted, and the context initiates a synchronizing handshake (Section 4.6.3) to verify that the other endpoint is still alive. If the synchronizing handshake fails, the context is aborted.

4.3 Timers

There are four timers that facilitate the protocol’s control procedures. Lost packets are discovered using the WTIMER. A lost association is discovered using the CTIMER. The CTIMEOUT timer monitors a synchronizing handshake, limiting the length of time the handshake is attempted. The RTIMER is the rate control timer used to space bursts of data.

The WTIMER is the timer that guards against the loss of a packet with the `SREQ` bit set. Whenever a packet is sent with the `SREQ` bit set, the transmitter increments the **saved_sync** value by one and places it into the *sync* field of the packet, as described in Section 2.2.6. The context sending the packet with the `SREQ` bit set also starts the WTIMER, loading it with a smoothed round-trip time estimate (see the Implementation Note in Section 2.3.1.3). The WTIMER is the amount of time the transmitter will wait for the arrival of the control packet requested with the `SREQ` bit.

Implementation Note

It is an implementation decision whether or not multiple WTIMERs are kept if multiple status requests are outstanding. If only one WTIMER is kept, the WTIMER must be restarted for each packet sent with the `SREQ` bit set, even if the WTIMER were already running.

If a control packet arrives at the transmitter before the WTIMER expires, the value in the *echo* field is compared with the **saved_sync** value. If they are equal, the WTIMER is stopped. If the WTIMER expires, the context starts the synchronizing handshake, described in Section 4.6.3

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

The user should be able to set the length of the CTIMER interval, but the user must not be allowed to disable the CTIMER. The CTIMER duration must be bounded so that the key anti-aliasing properties can be asserted (see the Implementation Note in Section 2.2.1, “Key Field,” for details about why CTIMER has a maximum value).

The CTIMEOUT timer limits the amount of time a synchronizing handshake can continue before the context aborts the association, as described in Section 4.6.3, “Synchronizing Handshake.” The CTIMEOUT timer is also used when a context goes into the zombie state after sending a packet with the `END` bit set. The CTIMEOUT timer can be disabled by setting the CTIMEOUT interval to zero, but if this is done, the initial value of the **retry_count** for the synchronizing handshake must not also be zero.

The RTIMER is the rate control timer used to govern the frequency of sending bursts of data. Its use is described in Section 4.5, “Rate Control.”

4.4 Flow Control

The volume of XTP output is regulated by an end-to-end windowing flow control mechanism. (The *rate* at which XTP sends packets into the network is regulated by an independent, timer-based mechanism described in Section 4.5, “Rate Control”). XTP’s flow control is based on a sliding window of sequence numbers. A sequence number is assigned to each output byte of the data stream, starting with the initialized sequence value.

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

Two option bitflags modify the way flow control is handled. The `RES` bit sent by the transmitter indicates to the receiver that the receiver should advertise conservative flow control values, specifically, the *alloc* value should reflect only as much buffer space as the user has allocated for the association. This is called *reservation mode*. The `NOFLOW` bit indicates to the receiver that the transmitter does not wish to adhere to flow control constraints, so flow control in the forward direction will be disabled. If the receiver does not wish to abide by these modes, the receiver can reject the association with a DIAG packet.

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.

4.5 Rate Control

Rate control governs the producer-consumer relationship between XTP endpoints. Rate control is concerned with how fast packets and their contents can be processed, or consumed, at the receiver. Parameters throttling the rate of production can be fed back to the sender through the rate control parameters of the Traffic Specifier. The Traffic Specifier *tformat* `0x01` (see Section 2.3.4.3) has explicit rate control parameters. If explicit rate control parameters are available, default *rate* and *burst* parameters must be used.

The output packet rate is regulated by two context variables, **credit** and **burst**, and by a refresh timer called RTIMER. (The **credit** and **burst** variables are not required by XTP but are useful for explaining rate control.) The values for **credit** and **burst** are calculated from the *rate* and *burst.* The *rate* value specifies the maximum data rate in bytes per second. The *burst* value specifies the maximum number of bytes to be sent in a burst of packets. The *rate* value divided by the *burst* value gives the number of **burst**-size transmissions per second, or the rate at which the **credit** variable is refreshed. The *burst* value divided by the *rate* value gives the time period for RTIMER.

The **credit** and **burst** variables are initialized with the *burst* value. With each outgoing data-bearing packet, **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 value of **credit**
2. if **credit** is positive, then replace it with **burst**

A value of zero for *burst* means that this context is not constrained, and can transmit at will. A value of zero for *rate* halts data transmission completely, although control 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.

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

4.6 Error Control

Error control in XTP is based on the exchange of information regarding lost or damaged data and the retransmission of these data. Each packet is examined for damage by performing a checksum, either over the header only or over the whole packet. Lost data are detected and recovered using an acknowledgement and retransmission procedure. The loss of a status request is detected by a timer; recovery in this case starts an exchange of packets designed to synchronize the endpoints of the association. Notification of other error conditions, such as unexpected packets or protocol errors, is made using DIAG packets. This section describes these facets of XTP's error control procedures.

4.6.1 Checksum

The XTP checksum function is the same function used for Internet protocols. The algorithm is a 16-bit one's complement sum over all octet pairs concerned (if the number of octets is odd, the last is data is zero padded and added in). The checksum is performed over the whole packet if the `NOCHECK` bit is cleared, and over the header only if the `NOCHECK` bit is set.

The *check* field (Section 2.2.4) of the header segment of XTP packets carries the checksum value. When the checksum is to be calculated, the *check* field is cleared, and the one's complement sum is formed over pairs of octets. Carries are folded back into the sum on two's complement machines (i.e., the overflows from the most significant end are added to the least significant bit). The result of the sum is placed into the *check* field.

To verify the checksum, the one's complement sum is again calculated over the octets concerned, including the *check* field. The checksum should be zero in the one's complement sense if the check succeeds. A C function for this checksum is given in Appendix A.

A received packet that fails the checksum is dropped, and no further action is taken.

4.6.2 Acknowledgements and Retransmission

A receiver detects missing packets by checking its incoming packet stream for gaps in the sequence space. The receiver records the missing data by keeping the sequence number of the first missing byte, and optionally by keeping a list of spans of correctly received data. When a control packet is to be sent, the context checks to see if any data are missing. If there are no data missing, then a CNTL packet is used. This packet acknowledges the receipt of all data whose sequence numbers are less than the value in the *rseq* field (Section

2.3.1.1). If missing data have been detected, an ECNTL packet is sent. In addition, using the *rseq* field in this packet to acknowledge data in the same manner as in the CNTL packet, the *nspan* and *spans* fields (Section 2.3.2.1 and Section 2.3.2.2) are used to selectively acknowledge spans of data received.

Receipt of an ECNTL packet implies that some data have been lost. The *rseq*, *nspan*, and *spans* fields specify what data are lost. The transmitter may retransmit data whose sequence numbers start at *rseq* and continue to the highest sequence number sent by the transmitter. This is go-back-*N* retransmission. Alternatively, the transmitter may selectively retransmit only the data specified as missing. In this case, the transmitter retransmits data starting at *rseq* and continuing up to, but not including, the first value of first *spans* pair. The next piece of retransmitted data is from the second value of the first *spans* pair to the first value of the second *spans* pair. This is illustrated by the example in Section 2.3.2.2.

Design and Implementation Note

Maintaining selective acknowledgement queues and tracking spans and gaps can lead to considerable complexity and overhead. This overhead can be justified when there are high packet loss rates (whether from buffer overflow or 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 packet loss networks, there may be no detectable performance difference between using selective retransmission or using simple go-back-*N*.

When the cost of maintaining selective acknowledgement queues is not justified, an XTP receiver may maintain just *rseq* and **hseq**, the value of the highest sequence number seen. Since the ECNTL packet design contains the fields 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.

A transmitter requests the status of the receiver's incoming data stream by setting the `SREQ` or `DREQ` bits in an outgoing packet's header. The `SREQ` bit indicates to the receiver that the status must be reported immediately. The `DREQ` bit indicates to the receiver that the status request should be satisfied only after the delivery to the user of all data whose sequence numbers are less than the value in the *seq* field, if this packet is a control packet, or the value of the *seq* field plus the value of the *dlen* field, if this packet is a data-bearing packet.

If the transmitter has set the `FASTNAK` bit in outgoing packets, the receiver is instructed to send ECNTL packets without first being asked to do so via the `SREQ` or `DREQ` bit. The `FASTNAK` bit indicates "fast negative acknowledgement mode"; an ECNTL packet is generated when the receiver discovers a gap in the data stream, that is, when the *seq* field value is greater than the next sequence number expected. The receiver generates an ECNTL packet reporting this gap. The receiver may not generate another ECNTL packet in response to a gap until it receives at least one packet for which the *seq* field value is the next sequence number expected. This is to avoid a storm of retransmissions.

Implementation Note

An implementation may limit the transmitter to retransmit only once any missing data indicated by an ECNTL packet until it can be determined that the retransmitted data have also been lost. This can be accomplished by keeping two additional sequence variables, **kseq** and **kseq_sync**, in the transmitter. The variable **kseq** is set to the maximum of the received *rseq* value and the highest retransmitted sequence number. The variable **kseq_sync** is set to the *sync* value of the last retransmitted packet. If the *echo* field of a subsequently received ECNTL packet is greater than or equal to the value in **kseq_sync**, the **kseq** variable is reset to *rseq*. Only data whose sequence numbers are greater than or equal to **kseq** are retransmitted.

Control of errors in the data stream can be turned off by the transmitter by setting the `NOERR` bit in all outgoing packets. This is called “no-error mode.” In no-error mode, the receiver must set the *rseq* field of control packets to the highest sequence number seen on its incoming data stream. An ECNTL packet is, therefore, never used by the receiver when in no-error mode.

4.6.3 Synchronizing Handshake

A synchronizing handshake is a control packet exchange where the *sync* field in the outgoing control packet must match the *echo* field in the incoming control packet. After this procedure, the transmitter is certain of the receiver’s state. The procedure is intended to eliminate spurious decisions that might be caused by network timing anomalies and packet loss. The procedure uses the *sync* field from the header, the *echo* field from the Control Segment, the WTIMER, the CTIMEOUT, and a **retry_count** variable. The variable **retry_count** governs the number of times the request control packet is sent. The CTIMEOUT timer bounds the total amount of time expended on the synchronizing handshake. If either CTIMEOUT expires or the number of retries exceeds a limit**,** XTP aborts the association.

When a packet is sent with the `SREQ` bit set, the packet’s sync value is saved in the variable **saved_sync**, and WTIMER is started, as described in Section 4.3. If the WTIMER expires before a control packet arrives whose *echo* field value is equal to the value in **saved_sync**, the context enters into the synchronizing handshake procedure. The objective is to probe the receiver with control packets at exponentially increasing time intervals until there is a successful handshake, or the CTIMEOUT or **retry_count** safeguards abort the context. No data-bearing packets are allowed to be sent during a synchronizing handshake, including retransmitted data; retransmission may proceed once the handshake has completed. The transmitter takes the following steps:

1. Load the CTIMEOUT timer with its initial value, reset the **retry_count** to its initial value, and set an exponential backoff variable **K** to 1.

2. Send a control (CNTL, ECNTL, or TCNTL, as appropriate) packet with the `SREQ` bit set, and save the *sync* value from that packet in the variable **saved_sync**.
3. Load the WTIMER with **K** times a smoothed round-trip time estimate (see the Implementation Note in Section 2.3.1.3).
4. If any received control packet contains an *echo* field that matches the value of **saved_sync**, then the synchronizing handshake is complete; stop the CTIMEOUT and WTIMER timers.
5. If the WTIMER expires before the conditions in Step 4 are satisfied, decrement the **retry_count** by 1, multiply **K** by 2, and go to Step 2.
6. If either **retry_count** equals 0 or the CTIMEOUT timer expires, abort the association.

4.6.4 Error Notification

A DIAG packet is used to notify the endpoints of an association when an error occurs. DIAG packets are generated either by a context or by the XTP context manager. In general, the error conditions are caused by (1) failure to deliver a packet, (2) a change in the maximum allowable packet size imposed by the underlying data delivery service, (3) the impending demise of the host, or (4) unacceptable traffic specification requests. A DIAG packet can never have a set `SREQ` bit.

The Diagnostic Segment of DIAG packets has three parts: the *code* field, the *val* field, and the *message* field. The *code* field specifies a general type of error condition. The *val* field gives more specific information about the nature of the error. Values for the *code* field that are defined for these situations are given in Table 4-1.

code	*Meaning*	*Receiver Action*
1	Context Refused	Abort context
2	Context Abandoned	Abort context
3	Invalid Context	Abort context
4	Request Refused	Implementor’s discretion
5	Join Refused	Abort context
6	Protocol Error	Implementor’s discretion
7	Maximum Packet Size Error	Fix PDU size or abort context

Table 4-1 — DIAG Code Values

The *Receiver Action* column specifies how the receiver of a DIAG packet with that row’s *code* value is supposed to react. A “Context Refused” DIAG packet is sent in response to a FIRST packet that was not accepted. The “Context Abandoned” DIAG packet is used to indicate that the sender’s context is being aborted, usually because the host is going down. The “Invalid Context” DIAG packet is used when an incoming non-FIRST packet cannot be matched to a context. The “Request Refused” *code* for DIAG packets is used for refusing a change in the current service, either by changing the mode bits in the *options* field or by sending a TCNTL packet with unacceptable traffic parameters. The “Join Refused” *code* is used by the multicast transmitter to deny a request to join the in-progress multicast association. A “Protocol Error” *code* indicates that the protocol has been violated in some way, such as trying to send fresh data on a closed data stream. The “Maximum Packet Size Error” indicates either that a sent packet was larger than the receiver’s underlying data delivery service could handle, or that the receiver’s underlying data delivery service is changing the maximum size of data it can handle. The maximum size of the Information Segment that the sender expects to transmit during the lifetime of the association is the value of the *val* field when the *code* is 7 (see the discussion on *maxdata* in Section 2.3.4.3, “Traffic Field”).

The *val* values specify more accurately what caused the DIAG packet to be generated. Table 4-2 shows the meanings of the *val* values and which *val* values are appropriate for the various *code* values.

val	*Meaning*	*codes*					
		1	2	3	4	5	6
0	Unspecified	✓	✓	✓	✓	✓	✓
1	No listener	✓					
2	Options refused	✓			✓	✓	✓
3	Address format not supported	✓					
4	Malformed address format	✓			✓		
5	Traffic format not supported	✓			✓	✓	
6	Traffic specification refused	✓			✓	✓	
7	Malformed traffic format	✓			✓	✓	
8	No provider for service	✓			✓	✓	
9	No resource	✓			✓	✓	
10	Host going down		✓				
11	Invalid retransmission request						✓
12	Context in improper state						✓
13	Join request denied					✓	

Table 4-2 — Appropriate Code/Val Combinations

5 Multicast Functional Specification

XTP multicast provides a powerful mechanism for group communication that supports a data transfer service for a one-to-many data flow. A multicast transmitter may send to an arbitrarily large receiver group. Since this is a transport layer multicast — rather than a datalink multicast or broadcast — flow, rate, and error control procedures are applied to the transmission of arbitrary-size messages to arbitrary-size groups.

Reliable transmission requires transmitter knowledge of the state of the receivers. In unicast, the transmitter knows that there is only one receiver, so simple techniques can be used to ascertain the receiver's state. In multicast, however, the state of all of the receivers must be tracked and resolved into information that the transmitter can use in its control algorithms. This implies that the transmitter must maintain some knowledge about the group of receivers.

XTP multicast provides the same control algorithms and mechanisms as XTP unicast. The fundamental difference between multicast and unicast is that there is no notion of duplex data transfer. This is because XTP does not dictate how data from multiple transmitters should be fused into one data stream. As a consequence, XTP multicast is a simplex data flow from one transmitter to an arbitrary number of receivers. (Appendix E shows how XTP multicast can be extended to include many-to-one and many-to-many data flows when assumptions are made about the data flows.)

Association management in multicast is closely related to the management of the group of receivers. As in unicast, the establishment of a multicast association is the process of discovering receivers. Unlike unicast, receiver set size may grow or shrink during the lifetime of the association. XTP does not impose policies for managing the group of receivers, since these are application and interface specific, but rather XTP provides the mechanisms for admitting, rejecting, and ejecting members whenever a group management policy so dictates.

This section first defines the fundamental concepts in XTP multicast, including the context and association state machines. Next, the procedures for association management are discussed, complete with association establishment and termination. There are several termination semantics, and these are given in detail. The particulars of the flow, rate, and error control procedures, as they apply to a multicast environment, are discussed.

5.1 Multicast Fundamentals

XTP multicast is intended for media that provide a broadcast or multicast facility. This service can be also extended to non-multicast media via a data delivery service that provides output replication to a set of destinations. Such a replication layer would be considered part of the media-specific encapsulation, and would be defined as part of the XTP encapsulation.

XTP multicast refers to a single transmitter with multiple receivers. It provides procedures to transmit a data stream in sequence order and free of duplicate data from a single XTP context to a set of XTP receiver contexts. XTP multicast is not by nature duplex. Whereas both endpoints of an XTP unicast conversation have a sending and a receiving side for duplex communication, in the multicast case endpoints are one-sided; receiving endpoints in the one-to-many case cannot send data in the reverse direction. Therefore, there is a distinction between the multicast transmitter (only the sending side is active) and the multicast receivers (only the receiving side is active).

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

1. All packets in a multicast association have the `MULTI` bit set, while packets in a unicast association always have the `MULTI` bit cleared.
2. Packets sent by the multicast transmitter may utilize the group address or an individual receiver's unicast address.
3. In an established multicast association, control packets sent to the multicast transmitter by the multicast receivers must use the transmitter's unicast address.
4. All packets sent from the multicast transmitter have the `RCLOSE` bit set.

All other fields and bitflags defined for unicast transmission are defined for multicast. A receiver should respond to the various field and bit values are the same way as in unicast, except where explicitly described in the rest of this section.

Multicast receivers join a multicast association in one of two ways. The first method is when a context listening on a multicast address receives a multicast FIRST packet. The second occurs when a context sends a JOIN packet to the multicast transmitter for an in-progress multicast association, and the transmitter responds with a JOIN packet, admitting the context to the group as a multicast receiver.

As contexts are admitted to the receiver group, the multicast transmitter maintains information about all active multicast receivers in the association. An *active receiver* in a multicast group is a receiver whose control information is used by the multicast transmitter when the transmitter runs its control algorithms. The active receiver set may be a subset of the all of the receivers; the group management policy in use at the transmitter determines the set of active receivers. Control packets sent by receivers who are not in the active receiver set will be ignored by the transmitter.

Control packets are issued by multicast receivers in response to receiving either SREQs or DREQs in the multicast transmitter's outgoing data stream. These control packets must be sent using the multicast transmitter's unicast address. For reliable data transmission in multicast mode, a transmitter must positively associate incoming control packets with past events if continuous output streaming is desired. This can be accomplished by matching returned *echo* values with local *sync* values. When a multicast receiver receives a control packet from the multicast transmitter, the same procedures are applied to it as defined for unicast associations.

Multicast receivers can leave the group by sending a CNTL packet with the END bit set. The multicast transmitter removes the receiver from the group but does not have to close the association, even if that is the last active receiver in the group. The multicast association is terminated only when the multicast transmitter sends a packet with the END bit set.

5.2 Multicast Addressing

The addressing information in the Address Segment is parametric and independent of the underlying data delivery service, as described in Section 2.4.1, "Address Segment." In order to use XTP multicast, the underlying data delivery service must provide XTP with a multicast or broadcast service. XTP does not define how to assign a multicast group address for this service; it is assumed that address assignments are accomplished by an outside mechanism, such as agreement between peers or an address management protocol.

The values placed in the Address Segment of FIRST or JOIN packets are also not assigned by XTP but must reflect the distinction between multicast addresses and unicast addresses. For the FIRST and JOIN packets, the source host address, when placed into the appropriate field of the Address Segment, must be a unicast address, and the destination host address must be a multicast address. These values are used to match a FIRST packet to listening contexts, and the JOIN packet to the appropriate multicast transmitter.

Example

Internet Class D addresses, as defined in RFC 1112, are multicast addresses used by IP. 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 MAC addresses, the low-order 23 bits of the IP address are placed in the low-order 23 bits of the Ethernet MAC address 01-00-5E-00-00-00. This means that many Class D addresses map onto the same MAC address. For FDDI the mapping is the same.

If IP multicast is used, the Class D address would appear in the *dsthost* field in the Address Segment, and the RFC 1112 mapping of that address would appear as the destination MAC address. In this way, a 48-bit IEEE-compatible group address can be uniquely mapped back into an Internet Class D address.

5.3 Multicast Association Management

Management of a multicast association is similar to management of a unicast association except for two important distinctions: the multicast association is not symmetric with respect to data transfer, and gathering receiver information, while trivial in the unicast case, must be done for an arbitrary number of receivers in a receiver group whose membership may change during the association. This second distinction is referred to as multicast group management.

5.3.1 Multicast Group Management

Multicast group management is concerned with the reliability of a multicast transmission from the viewpoint of dynamic group membership. The reliability of the multicast association depends on the control algorithms used in the association and on the group management policy. Conceptually, the XTP multicast transmitter maintains a table with the state information of the active multicast receivers as derived from control packets. The control algorithms running at the multicast transmitter use the contents of this table while the group management policy determines when and how changes in membership are handled.

The group of active receivers may be a subset of the actual group of receivers. These active receivers are the ones whose control information is used to drive the control algorithms, while control information from all other receivers is not used.

XTP multicast is a collection of mechanisms that support group communication. A group management policy encapsulates how these protocol mechanisms are used. There are three aspects of group communication where a policy is required:

- group membership admission
  - how the multicast transmitter discovers the initial set of receivers
  - how receivers are admitted to the active receiver group
  - how to admit receivers to the receiver set but not to the active receiver group
- group membership pruning
  - when to remove a receiver from the active receiver group
  - when to drop a receiver from the receiver set
- group reliability
  - when is the active receiver group insufficient

The mechanisms used for admission to the multicast association are described in Sections 5.3.3 and 5.3.4. The user must specify how the initial group of active receivers is compiled, and the criterion for admission to this active receiver group while the association is in progress. The user must also specify the policy for admission to the receiver set even if the receiver is not part of the active receiver group.

The user must specify under what conditions a receiver set is to be removed from the active receiver group, and when a receiver is to be ejected from the receiver set entirely. The user must define what is meant by a receiver "falling too far behind" the other receivers. Group pruning is discussed in Section 5.3.10

The term "reliable" must be defined with respect to the integrity of the active receiver set. Group reliability semantics are discussed in Section 5.3.5.

5.3.2 Multicast Association Establishment

An association is established when one or more listening multicast contexts receives a FIRST packet, and all participating contexts (one transmitting and one or more receiving) have moved into the active state. There can be multiple multicast contexts on the same host listening on the same multicast address. An incoming FIRST packet is matched against *all* listening contexts to find those that will accept the association.

A received FIRST packet, if it is not discovered to be a duplicate for an already active context, is subjected to a matching algorithm to determine if any listening contexts should get a copy of the FIRST packet. Each listening context submits an *address filter* that represents the values of an address that the context is willing to accept, as well as acceptable traffic shaping parameters and options bits. The FIRST packet's contents are compared against each listening context's criterion for acceptance until all listening contexts have been examined.

Upon receipt of a FIRST packet, the receiving host takes the following steps:

1. If a full context lookup (see Section 5.3.6) on this FIRST packet finds an active context, the FIRST packet is a duplicate; a copy of the FIRST packet should be given to each of the contexts to which this packet belongs, and each of them should respond to the `SREQ` and `DREQ` bits, if set, and accept any additional data carried within, but no new contexts become active.
2. If the FIRST packet is not a duplicate, the receiving host performs a series of comparisons to find all appropriate listening contexts:
   - The *address* field within the Address Segment of the FIRST packet is matched against a set of *filters*.
   - The *service* field in the Traffic Specifier segment is matched against the context's *service* value: if the context's *service* value is 0, any *service* field value will match; otherwise the values must match exactly.
   - The traffic specification values in the Traffic Specifier segment are matched against the traffic specification acceptable by the context.
   - The settings of the option bits in the *options* field of the header of the FIRST packet are examined for acceptability.

3. A copy of the FIRST packet is given to each listening context for which the FIRST packet's address, traffic specification, and options are acceptable.
4. Otherwise, the FIRST packet is discarded without reply.

For each listening context that accepts the FIRST packet, an entry for the context is made in the translation map that will map to this context any incoming packets whose *key* field is the same as this FIRST packet's *key* field, and whose source host's address (obtained from the underlying data delivery service) is the same as this FIRST packet's source host's address. This entry is fundamental in the full context lookup, described next. The packet is given to the found context(s), which has (have) now moved from the listening state to the active state.

Unlike unicast, listening contexts are not allowed to reject a FIRST packet with a DIAG packet. If a multicast FIRST packet arrives but fails to meet the criteria set out by the listening context, the context simply ignores the FIRST packet.

Example

A multicast transmitter with unicast host address A elects to initiate a multicast association. For purposes of this example, three processes (two on host B and one on host C) anticipate the formation of the multicast group and have already posted a listen on the multicast address. Call these contexts B_1 and B_2 on host B and C_1 on host C. Knowledge of the multicast address on which to listen has been provided by some outside agent.

The multicast transmitter sends a FIRST packet with its source address set to be A's unicast address and with the destination address set to be the multicast address. Hosts B and C each see the FIRST packet, compare it against the address, service, traffic, and options specified by contexts B_1, B_2, and C_1, and, assuming acceptability, pass the FIRST packet to the three awaiting contexts; the translation map in each host associates the *key*, and unicast source address in the FIRST packet with these three contexts so that all future packets with the same *key* and source address will likewise be passed to these receiving contexts.

At this moment, the three contexts become members of the multicast association (because their state has moved from listening to active), although the transmitter does not yet know about them. So at this point a multicast association has been successfully established, but the multicast transmitter does not know how many active receivers there are, nor can it identify any of them uniquely.

5.3.3 Discovering the Receiver Set

The establishment of a multicast association does not provide the multicast transmitter information about the receiver set. To form an active receiver group, the multicast transmitter must request the status of the receivers by sending a packet to the receivers with the `SREQ` bit set. This packet should be the FIRST packet. The receivers must respond to the first `SREQ` seen on the multicast address by sending a TCNTL packet to the transmitter (since the multicast transmitter ignores control packets from receivers not in the receiver set, an ECNTL packet, if errors were detected, is not useful here until after the TCNTL is sent). Each receiver must fill in the *xkey* field with its return key.

From these TCNTL responses, the multicast transmitter can build a data structure that identifies the receivers and captures their current state. While the form and content of this data structure are not specified by XTP, useful information to record for each receiver would be its source address, its return key, its current value of *alloc*, its current value of *rseq*, and the *echo* value associated with these values; all of this information is available from the TCNTL responses. Of course, other information may be captured as well, but this information allows the user to implement some fundamental reliability management services.

If the user's group management policy does not require knowledge of the receiver set, the multicast transmitter does not need to gather receiver information.

5.3.4 Joining an In-Progress Multicast Association

When a multicast receiver wants to join (or rejoin) a multicast association that is already in progress, the following procedure must be used. Joining an in-progress multicast association involves sending a JOIN packet to the multicast group. This JOIN packet is sent periodically until either a reply is received or until a user-defined number of attempts have been made. The Address Segment of the JOIN packet must have the multicast group address in the appropriate destination host address field, and the requesting host's unicast host address in the source host address field. The *key* field of the JOIN packet must be zero, the *seq* field must carry the requesting context's return key value, and the `MUTLI` bit must be set.

An established multicast receiver must never respond to a JOIN packet. The multicast sender alone responds by sending a JOIN packet. This JOIN packet contains the *key* value for the multicast context, the next sequence number for the data stream in the *seq* field, and the usual values for all other fields back to the requestor. Upon receiving a JOIN packet in which the *key* field is not zero, the requesting context moves to the active state and parses the JOIN packet as though it were a FIRST packet. The new receiver begins receiving data starting at the sequence number carried in the *seq* field.

When a new receiver joins an in-progress association, the multicast transmitter updates the set of active receivers according to its group management policy. A new receiver does not have to be included in the active receiver set; if it is not, the control information sent by the receiver will not be used in the multicast transmitter's control procedures.

The multicast transmitter may reject the JOIN packet by sending a DIAG packet back to the join requester using the requester's unicast address and its return key, and the `MULTI` bit must be set. DIAG packets sent in a multicast association must follow the rules outlined in Section 5.4.3, "Error Notification." The JOIN packet is rejected by using a

DIAG packet whose *code* value is 5, “Join Refused.” There is a hierarchy of rejection reasons, as indicated by the *val* value:

1. failure due to the *service* field
   - *val* 8: “No provider for service”
2. failure due to the Traffic Segment
   - *val* 5: “Traffic format not supported”
   - *val* 6: “Traffic specification refused”
   - *val* 7: “Malformed traffic format”
3. failure due to the options bits
   - *val* 2: “Options refused”
4. failure due to group management policy
   - *val* 13: “Join request denied”

If none of these reasons apply, the rejection should be done using a DIAG packet with *code* value 5 and *val* value 0, “Unspecified.”

5.3.5 Group Reliability Semantics

In general, the data structure maintains a set of useful information about each active receiver. The multicast transmitter sets `SREQ` in any outgoing packet to force an update of the information.

To determine whether all receivers are synchronized with each other with respect to the data stream, the multicast transmitter would observe the *rseq* values in the data structure to verify that all values are within some user-defined threshold of the multicast transmitter’s last *seq* value. If an *rseq* value is identified as being significantly lower than the multicast transmitter’s *seq* value, this may indicate a receiver that is significantly slower than the other members of the group, or a receiver that has exited the group without notification.

Suppose a receiver has been identified that has fallen behind in the sequence space. Is the receiver dead, or just slow? Setting `SREQ` as described above forces an update of the data structure. If all receivers respond (as can be determined by matching the returned *echo* value against sent *sync* values), then the offending receiver is just slow. If one or more receivers fails to respond, a synchronizing handshake (Section 5.4.2) will assure that all receivers have had adequate time to respond. A receiver that has not responded within the bounds of the synchronizing handshake is assumed to be dead.

If the user’s reliability semantics require that all members of the group must remain alive and current, then the detection of a failed receiver dooms the group’s integrity; the multicast association is then aborted by the multicast transmitter by sending an `END` bit to the group (consistent with the requirement to maintain a zombie state in the multicast transmitter if it is required that all data up through and including the packet containing the `END` bit be delivered reliably).

If the user's reliability semantics require that only still-functioning members of the group remain current, then there is an alternative to aborting the group when one receiver fails or falls significantly behind in the sequence space. The multicast transmitter can, upon detection of a failed or slow receiver, send an `END` bit to the unicast address of that particular receiver and remove that receiver's state information from the data structure; this effectively removes a receiver from the group, and now the remaining group members may continue.

The group reliability data structure allows the implementation of *k*-reliability semantics. Suppose that the user requires that at least *k* receivers be operational for the group to continue; by monitoring the cardinality of the group, the user can assure that this requirement is met. Note that this example of reliability does not specify the identity of the reliable group members, only their number, so under this definition the group is intact as long as their are at least *k* functioning members, regardless of who they are. A more rigorous reliability semantic could be enforced by requiring that not only must *k* group members be present, but they must be *k* specific members.

The user can even impose hybrid reliability requirements on the group. For example, it can require that group members X, Y, and Z be fully reliable, but impose no such restraint on other group members. By tracking the group membership data structure, the user can assure that X, Y, and Z are both active and current, and can abort the group if any of those three receivers fail. Meanwhile, other members can join and leave, but their presence or absence will have no effect on the total group reliability semantics.

The amount and type of information recorded in the group membership data structure, and the degree to which that information is exposed to the user, affects the breadth of group reliability semantics that can be imposed by the user.

5.3.6 Full Context Lookup

In the unicast case, the full context lookup procedure maps an incoming packet to the *single* appropriate active context if the *key* value in the incoming packet is not a return key, as explained in Section 4.2.2, "Full Context Lookup." In the multicast case, this lookup procedure will be used to find *all* of the contexts to which the incoming packet should be delivered. The translation map structure holds the mapping from packet information to all appropriate multicast contexts.

5.3.7 Abbreviated Context Lookup

Any packet whose *key* field holds a return key value, whether in a multicast association or not, will be mapped to the single context indicated by the return key, as explained in Section 4.2.3, “Abbreviated Context Lookup.”

5.3.8 Key Exchange

A return key cannot be used for packets sent from the multicast transmitter to the whole multicast receiver group, since each multicast receiver may have a different return key. Consequently, data-bearing and control packets sent on the multicast address must always carry the multicast transmitter’s *key* value, so each receiver will be found using the full context lookup described above. However, there are situations where a packet can be sent to a specific multicast receiver. The only way to uniquely identify a single receiving context is for the multicast transmitter to have the receiver’s unicast address and its return key.

There are two ways that a multicast receiver can inform the multicast transmitter of its return key: by using a TCNTL packet in response to a (possibly FIRST) packet with the `SREQ` bit set, or by sending the return key value in the *seq* field of the JOIN packet used to request admission to the group.

5.3.9 Traffic Specification Negotiation

As with the unicast association, the traffic specification can be negotiated. There are several differences in the way the negotiation takes place in multicast associations because of the unidirectional flow of data, the fact that the traffic shape will necessarily be the same for all of the multicast receivers, and the fact that multicast receivers are not allowed to reject a traffic specification with a DIAG packet.

When the FIRST packet is sent to the initial group of receivers, the FIRST packet carries a Traffic Specifier indicating the parameters to be used for the outgoing data stream. These receivers filter on this specification. If the FIRST packet has the `SREQ` bit set, those receivers that accept the specification as is or with modification respond with a TCNTL packet. Those receivers that do not accept the specification drop out of the group, either silently, or by sending a CNTL packet with the `END` bit set. DIAG packets can not be used to drop out of the multicast group.

During an attempt to join an in-progress association, the JOIN packet is sent on the multicast address with a Traffic Specifier filled in with the parameters the receiver would like to see from the transmitter. If the transmitter finds these parameters acceptable, or if the transmitter modifies them, the multicast transmitter will respond with another JOIN packet as prescribed above. The transmitter rejects this attempt to join by using a DIAG packet.

At any time during the multicast association, the multicast transmitter or any one of the multicast receivers can renegotiate the traffic specification by using the rules for traffic specification negotiation described in Section 4.2.5. If the change is initiated by the multicast transmitter, those receivers that do not accept this change drop out of the multicast group (described next).

5.3.10 Leaving the Multicast Association

There are three ways that a multicast receiver can depart a multicast association: voluntary exit, forced exit, and silent exit. When a receiver is no longer interested in the multicast association, the receiver voluntarily leaves the multicast association by sending a CNTL packet with the `END` bit set to the multicast transmitter. The multicast receiver then follows the same rules as any context sending an `END` bit: the context goes into a zombie state for CTIMEOUT seconds. The multicast receiver in the zombie state responds only to packets from the multicast transmitter sent to it on the receiver's unicast host address, where the packet contains a return key in the *key* field. The only valid response is for the zombie receiver to retransmit a CNTL packet with the `END` bit set. The multicast transmitter updates the active receiver group as receivers leave the group.

A multicast receiver can be forced to exit the multicast association. Any time a multicast receiver receives a packet with the `END` bit set, the receiver must immediately abandon the multicast association and go quiescent, as per the rules for receiving an `END` bit. The multicast transmitter updates the active receiver group. If the packet with the END bit is not received by the receiver, that receiver may still send control packets back to the transmitter. The multicast transmitter can ignore these control packets or send an additional control packet with the `END` bit set.

If a multicast receiver in the active group fails to respond to status requests, the multicast transmitter will eventually enter a synchronizing handshake as described in Section 5.4.2. If the multicast receiver fails to respond to this handshake, the multicast transmitter removes the receiver from the active receiver group.

5.3.11 Multicast Association Termination

In order for a multicast association to be terminated, all of the active multicast receivers must be released, and the multicast transmitter must send a packet with the `END` bit set. A context in a healthy association is closed by completing the packet exchanges with `WCLOSE`, `RCLOSE`, and `END` bits. Just as with unicast, closure is achieved with various degrees of gracefulness, depending on the bits used and the order in which they are set.

The `WCLOSE` bit is set when the multicast transmitter has completed the transmission of all data in its outgoing data stream. Once the `WCLOSE` bit is set in an outgoing packet, all subsequent outgoing packets, except retransmitted DATA packets, must carry the set `WCLOSE` bit.

The `RCLOSE` bit has to be set in each outgoing packet from the sender, starting with the FIRST packet. This indicates that the sender's incoming data stream is closed, and the multicast association will be inherently simplex. As a consequence, all packets sent from multicast receivers will have the `WCLOSE` bit set.

The multicast receivers set their `RCLOSE` bit only after the receipt of a packet with the `WCLOSE` bit set. After a `WCLOSE` is received, the error control procedures must ensure that all data on the incoming data stream have been received, according to the error control parameters. Once the error control procedures have been satisfied, the `RCLOSE` bit must be sent in the next and all subsequent outgoing packets.

The `END` bit sent from the multicast transmitter indicates the end of the association. The packet carrying the `END` bit can be multicast to the whole group, or sent via a unicast address to a specific receiver (as discussed above). The multicast association is terminated only when the multicast transmitter multicasts a packet with an `END` bit set.

A multicast transmitter, after sending the `END` bit, must wait in a zombie state for a duration of CTIMEOUT. This is to eliminate connection hazards resulting from the case where the packet carrying the `END` bit is lost; if any of the multicast receivers ask for a retransmission, the host with the now-quiescent multicast transmitter would respond with a DIAG, and the receiver would not know if the connection was lost or finished gracefully. This is avoided by maintaining a zombie context with at least enough state information to respond to a retransmission request. For situations where the hazard will not arise, or the users do not care, the CTIMEOUT can be reset to zero, effectively eliminating the zombie state.

Abbreviated Graceful Close

This is the standard closing procedure for multicast. As depicted in Figure 5-1, Context A is the multicast transmitter, and Context B, Context C, and Context D are the multicast receivers. All packets originating at Context A have the `RCLOSE` bit set, and all packets originating at receivers B, C, and D have bit `WCLOSE` set. Context A initiates the close by setting the `WCLOSE` bit in an outgoing packet. This packet is received at Contexts B, C, and D. Each receiver will respond with a packet that has bit `RCLOSE` set. The multicast transmitter has to collect responses from all active multicast receivers B, C, and D carrying the `RCLOSE` bit. After that, Context A sends a packet with all three bits `WCLOSE`, `RCLOSE`, and `END` set to end the association.

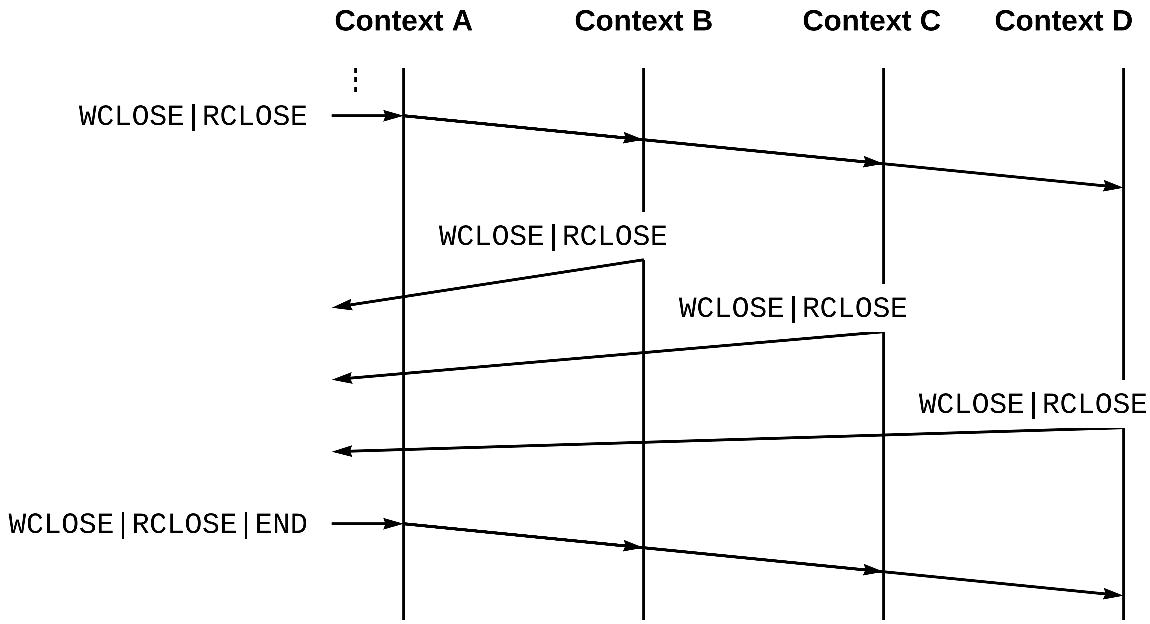

Figure 5-1 — Abbreviated Graceful Close

Abortive Close

Sending the `END` bit from a multicast transmitter at any time during the life of the association aborts the association, regardless of the state of its multicast data stream. Sending the `END` bit from a multicast receiver at any time during the life of the association only serves to remove the receiver from the list of active receivers.

5.4 Flow Control, Rate Control, and Error Control

The multicast transmitter obeys the same rules for flow control, rate control, and error control as a unicast sender. To summarize: the multicast transmitter begins with default or inherited values for allocation, rate control, and the wait interval. As the sender adds `SREQ` or `DREQ` to outgoing DATA or control packets, receivers respond with control packets, which contain new values for the flow, rate, and error control algorithms. The sender also periodically updates the round-trip time estimate by observation. Thus, the packet exchanges carrying flow control and rate control parameters between the sender and receivers are identical to those between a sender and a single receiver.

The only consideration specific to multicast, however, is that the multicast transmitter must gather control information from the group of multicast receivers. Each receiver responds to status requests with control packets. The values for the flow, rate, and error control algorithms must be resolved such that the values for *rseq* and *alloc* (from any control packet), the effective *rate* and *burst* (possibly from a TCNTL packet), and *nspan* and *spans* (for ECNTL packets) are aggregated from the control packets received from the known group of receivers. How these values are aggregated from the receiver group is implementation, and possibly application, specific, and is not defined by XTP.

As in the unicast case, multicast error control is based on the exchange of information regarding lost or damaged data and the retransmission of this data. Each packet is examined for damage by performing a checksum, either over the header only or over the whole packet, depending on whether `NOCHECK` is set. Lost data are detected and recovered using an acknowledgement and retransmission procedure. The loss of a status request is detected by a timer; recovery in this case starts an exchange of packets designed to resynchronize the endpoints of the association. Notification of other error conditions using DIAG packets, however, is allowed only if the DIAG packet can be sent using a unicast host address as the destination and a return key in the *key* field.

5.4.1 Acknowledgement and Retransmission

The same error control procedures defined for unicast associations are available in multicast mode. If the `NOERR` bit is set, the multicast receivers apply the standard algorithm: all sender `SREQ` and `DREQ` requests are acknowledged with CNTL packets with the *rseq* value set to the highest sequence number received. If `NOERR` is not set, a receiver reports *rseq*, *nspan*, and *spans* in ECNTL packets according to the normal unicast rules and reports its state to the sender in control packets.

If error recovery is enabled, the values for the error control algorithm must be resolved such that the worst case values for retransmission are taken from the set of received control packets from the receivers. This means that the multicast transmitter retransmits all of the lost data reported in the set of control packets from the receivers.

The rules for use of the `FASTNAK` bit in the multicast cast are the same as for unicast except that the multicast association has one data stream, so only the multicast transmitter can set the `FASTNAK` bit for the multicast association. This causes the receivers to generate ECNTL packets as soon as a gap in the data stream is noticed.

5.4.2 Synchronizing Handshake

A synchronizing handshake in multicast, as in unicast, serves to establish a point in the association when each participant has up-to-date status information. If the synchronizing handshake is initiated by the multicast transmitter, the transmitter must ensure that all of the receivers have completed the synchronizing handshake before the transmitter can quit the handshake. Those receivers who do not complete the synchronizing handshake when the retry count is exceeded or the CTIMEOUT timer expires are removed from the transmitter's active receiver group. The exponential back-off algorithm reflects this fact as follows:

1. Load the CTIMEOUT timer with its initial value, reset the **retry_count** to its initial value, and set an exponential backoff variable **K** to 1.
2. Send a control (CNTL, ECNTL, or TCNTL, as appropriate) packet with the `SREQ` bit set, and save the *sync* value from that packet in the variable **saved_sync**.
3. Load the WTIMER with **K** times a smoothed round-trip time estimate (see the Implementation Note in Section 2.3.1.3).
4. If a control packet is received from each active receiver containing *echo* fields that match the value of **saved_sync** for all received control packets, then the synchronizing handshake is complete; stop the CTIMEOUT and WTIMER timers.
5. If the WTIMER expires before the conditions in Step 4 are satisfied, decrement the **retry_count** by 1, multiply **K** by 2, and go to Step 2.
6. If either **retry_count** equals 0 or the CTIMEOUT timer expires, either remove these receivers who failed to respond or abort the multicast association.

Implementation Note

"Slotting" is a technique that imposes a random delay at the receivers 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.

With slotting, the number of control messages is distributed over time rather than concentrated at once. This enhances performance because it avoids an instantaneous increase in network offered load (also called a "control packet implosion").

A multicast receiver conducts a synchronizing handshake with the multicast transmitter in exactly the same way that a unicast synchronizing handshake is conducted. When a multicast transmitter receives a packet with the `SREQ` or `DREQ` bit set, the control packet sent in return may be multicast to the group or simply sent to the receiver requesting the status via its unicast address.

5.4.3 Error Notification

Error notification in the form of DIAG packets is allowed in within multicast associations only if the DIAG packet is generated by a context. Also, a DIAG packet must never be generated in response to packets sent on the multicast address. These two rules specifically disallow rejections of a multicast FIRST packet.

A DIAG intended for a specific context must be sent using a unicast address and a return key in the *key* field. DIAG packets, as with all packets in a multicast association, carry the `MULTI` bit set. The DIAG packet "Context Abandoned, Host going down," may be sent on the multicast address only if the host going down is the multicast transmitter.

6 Encapsulation

An encapsulation procedure specifies how to incorporate a protocol data unit (PDU) from one layer within the PDU of a lower layer. An XTP packet is contained, or encapsulated, within a lower layer frame, and that lower layer becomes the data delivery service for the XTP packet. The underlying data delivery service employed by XTP, and into whose frames XTP packets will be encapsulated, must provide three things:

1. end-to-end delivery of the XTP packet
2. some validity assurance over the encapsulating frame
3. the source host addresses used by the service

The service must provide end-to-end delivery since XTP is a transport layer protocol without routing capabilities. If XTP is used in an internetworking environment, XTP's underlying data delivery service must be a network layer protocol. However, XTP may be used in an environment where routing services are not needed; in this case XTP can be interfaced directly over a MAC or AAL layer protocol, and XTP packets encapsulated directly into those frames. Encapsulations for both of these environments are given in this section.

The source host addresses are used by XTP's full context lookup procedure (see Section 4.2.2). The source address must uniquely identify the sending host for the full context lookup procedure to work.

Implementation Note

An XTP packet and its internal segments are defined on 8-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 an encapsulation frame in contiguous memory, the frame should be offset in memory to account for various misaligned frame header sizes.

6.1 Ethernet Encapsulation

The packet format for XTP implementations that are interfaced directly to an Ethernet is also shown in Figure 6-1. The *DestAddr* and *SrcAddr* fields are the 6-byte physical address of the Ethernet device. The *type* field is used to determine the client protocol for which this frame is destined, and consequently provides a means for demultiplexing Ethernet frames to various higher layer protocols. (IEEE 802.3 does not have the *type* field, so those frames need LLC information to demultiplex them.) The Ethernet type for XTP packets is `0x817D`.

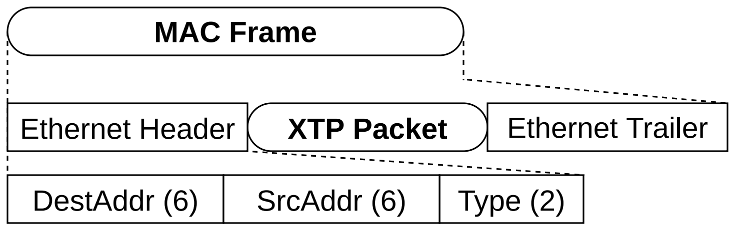

Figure 6-1 — Ethernet Encapsulations

6.2 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. The format for this encapsulation is shown in Figure 6-2.

In this encapsulation, the destination and source service access point fields, *DSAP* and *SSAP*, are both assigned the decimal value 170 to indicate that the SNAP header is present. The *control* field value is 3, indicating "unnumbered information." The organiza-

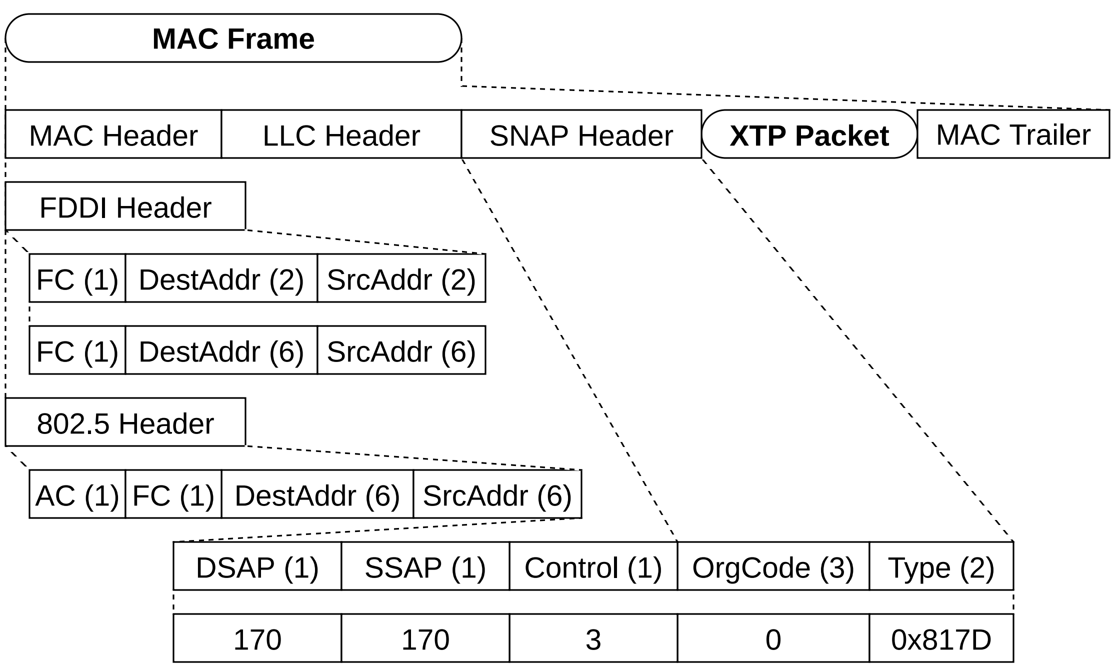

Figure 6-2 — LLC Layer Encapsulations

tion code field *OrgCode* is assigned the value 0, which is the code assigned to Xerox Corporation. The *type* field is the same as for Ethernet (`0x817D`).

6.2.1 FDDI Encapsulation

In general, encapsulating an XTP packet into an FDDI frame follows the guidelines established in RFC 1103 “A Proposed Standard for the Transmission of IP Datagrams over FDDI networks.” The FDDI header has three fields, a frame control field (*FC*), the destination address (*DestAddr*) and the source address (*SrcAddr*). The *FC* field contains information about this MAC frame, including rudimentary priority information, the address length, and what kind of data is present in this frame (LLC data or MAC control). The address fields can be either 16 or 48 bits long, and represent the physical addresses of the FDDI devices in the destination and source hosts.

6.2.2 IEEE 802.5 Token Ring

Encapsulation for IEEE 802.5 Token Ring is similar to the encapsulation for FDDI frames in that it uses LLC and SNAP headers in the same manner as FDDI. The access control field (*AC*) contains the priority and reservation bits that are used in Token Ring’s priority reservation access mechanism. The frame control field (*FC*) indicates whether the data contained within is LLC data or a MAC control frame. The *DestAddr* and *SrcAddr* fields are the 6-byte physical address of the Token Ring device.

6.2.3 ATM Adaptation Layer 5

Encapsulation for ATM AAL5 follows the guidelines put forth in RFC 1483, “Multiprotocol Encapsulation over ATM Adaptation Layer 5.” Section 4.1 describes the LLC encapsulations for routed protocols. When XTP is being multiplexed for a single VC, it uses LLC 802.1/SNAP encapsulation using XTP’s assigned *EtherType* `0x817d`.

XTP packets can also be encapsulated directly into AAL 5 frames without using an LLC encapsulation.

6.3 IP Encapsulation

The IP encapsulation for an XTP packet is shown in Figure 6-3; the IP encapsulation follows the rules specified in RFC 791. (LLC and SNAP headers will be included as specified in RFC-1103 when IP datagrams are transmitted on FDDI networks, in accord with RFC 1103.) The 1-byte *protocol* field in the IP header is loaded with the decimal value 36; this value has been assigned to XTP to specify that XTP is the next higher-layer protocol.

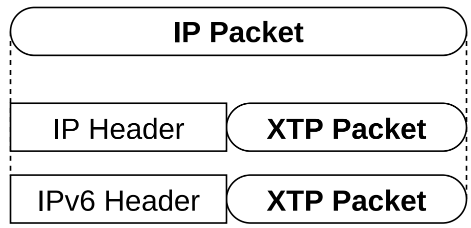

Figure 6-3 — IP Encapsulation

An IP version 6 encapsulation will be defined when the IP version 6 frame structure is fully defined.

6.4 Security Encapsulation

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.

Appendix A Check Function

This is a C function for this checksum. This is the Braden, Borman, and Partridge algorithm as related by Stevens in his book, “UNIX network programming.”

```
unsigned short xsum(int len, unsigned short* ptr) {
  unsigned int sum = 0;
  unsigned short oddbyte = 0;
  unsigned short answer;

  /*
   * Algorithm: use a 32-bit accumulator (sum) and add sequential
   * 16-bit words into it; at the end, add the two halves of sum together
   * to fold back the carries. Return the one’s complement of this.
   */

  while (len > 1) {
    sum += *ptr++;
    len -= 2;
  }

  /* Mop up an odd byte, if necessary. */
  if (len == 1) {
    *((unsigned char*)&oddbyte) = *(unsigned char*)ptr;
    sum += oddbyte;
  }

  /* Add back carry outs from top 16 bits to low 16 bits */
  sum = (sum >> 16) + (sum & 0xFFFF);     /* Add high-16 to low-16 */
  sum += (sum >> 16);                     /* Add carry */
  answer = ~sum;                          /* One’s complement */

  /* Return the answer */
  return(answer);
}
```

Appendix B Additional Traffic Specifier Formats

The traffic format `0x02`, shown in Figure B-1 has been assigned to the quality of service parameters defined by the Technical University of Berlin.

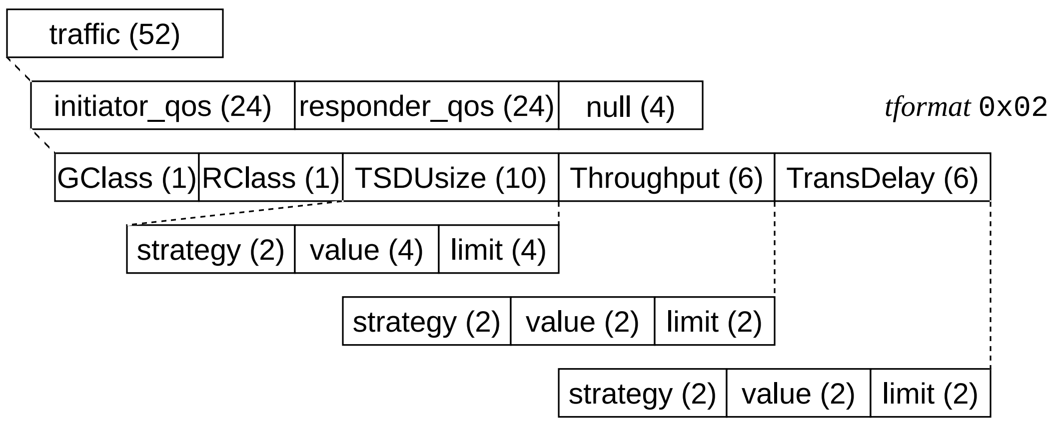

Figure B-1 — Traffic Field Structure for Format 0x02

This traffic specification has two identical fields for specifying quality of service parameters. The *initiator_qos* specifies the parameters for the data flow from the context initiating the connection to the responder(s) of this connection. The *responder_qos* field similarly holds the parameters describing the user data flow from the responding context(s) of the connection to the initiating context (reverse data stream). Within both of these fields are structures to specify message size, throughput, delay characteristics and reliability of the data transfer. The transport system (XTP and its underlying service) has to guarantee these parameters as described in an additional parameter, the Guarantee Class (*GClass*).

The traffic specifier is negotiated during the establishment of the connection. Furthermore, renegotiations may appear during the lifetime of the connection as described in Section 4.2.5, “Traffic Specification Negotiation.” This negotiation takes place between all involved communicating entities (initiating user, initiating XTP-context, network provider, responding XTP-context(s), responding user(s)). Specific functions may map this traffic specifier onto QoS-parameters of underlying providers (e.g., the traffic descriptor of an ATM connection).

To make the negotiation of the parameters more effective, both desired and acceptable (threshold) values for all negotiable parameters are given. As an example, the transport user could specify a desired transit delay of 25 ms and an acceptable delay of 35 ms. The connection will be established only if the desired delay does not exceed the acceptable 35 ms after the complete negotiation. This implies that the acceptable value cannot be negotiated and a QoS negotiation may become unsuccessful before the complete negotiation handshake, if the desired value exceeds the acceptable value during the negotiation process.

The assignment of the *strategy* parameters is not yet defined; they must be zero. The meaning of the other parameters is as follows:

Guarantee Class (*GClass*) describes the probability that the negotiated QoS is actually provided by the transport system and mechanisms if the QoS is violated. The following Guarantee Classes are defined:

GClass	*Meaning*
`0`	best-effort provision of QoS, no indication of QoS violation to the user
`1`	best-effort provision of QoS, indication of QoS violation to the user
`2`	guaranteed provision of QoS, no indication of QoS violation to the user
`3`	guaranteed provision of QoS, indication of QoS violation to the user

Reliability Class (*RClass*) indicates the degree to which the protocol supports error detection and error recovery. The following Reliability Classes are defined:

RClass	*Meaning*
`0`	unreliable; duplex transport of user data without failure indication
`1`	unreliable; duplex transport of user data with indication of lost data
`2`	unreliable; duplex transport of user data with indication of lost and corrupt data
`3`	partially reliable; duplex transport of user data with correction of lost data, but with no correction of corrupted data
`4`	reliable, duplex transport of user data

Maximum TSDU (message) size (*TSDUsize*) is the maximum allowed size of one message to transmit, given in bytes.

- *strategy*: MBZ
- *value*: negotiable parameter describing the maximum message size
- *limit*: lowest acceptable maximum message size

Throughput (*Throughput*) is the number of sent/received messages per second

- *strategy*: MBZ
- *value*: negotiable parameter describing the throughput
- *limit*: lowest acceptable throughput

Transit Delay (*TransDelay*) is the elapsed time between sending a message and its delivery to the peer user. The delay is measured in milliseconds.

- *strategy*: MBZ
- *value*: negotiable parameter describing the delay
- *limit*: highest acceptable delay

Appendix C Address Resolution

This appendix describes a family independent mechanism for address resolution within XTP. The packet formats and structures allow for compatibility among XTP implementations that do not support this appendix. The principles for this approach are similar to those described in RFC 826.

This mechanism is not intended as a general address resolution mechanism. It is intended primarily for those environments where network level encapsulations are not needed. If network level encapsulations are required, address resolution is normally accomplished through the use of the network level address resolution mechanism.

Packet Format

The packet format used for address resolution is a DIAG packet. The *code* for this type of DIAG is 8 (Address Resolution). Currently there are two *val* values defined for this mechanism:

code	*Meaning*	*val*	*Meaning*
8	Address Resolution	30	Host address resolution request
		31	Host address resolution response

Table C-2 — Address Resolution DIAG code and val Values

The *message* value for the packet is a full XTP Address Segment. The following fields in the XTP header should be zero: *key, sort, sync,* and *seq.* The remaining header fields must contain valid values.

Host Address Resolution Request

An address resolution request packet is a DIAG packet with *code* 8, *val* 30. It is used to determine an unknown lower layer address of a desired host. The destination in the address segment describes the desired host for which the address is being sought. The *source* field in the address segment describe the sender. When an end-system receives an address resolution request packet with the destination host address matching its host address it must send an address resolution response packet to the *source* host. Address resolution request packets use the broadcast or multicast facilities of the underlying service provider. All end-systems can cache the (*source*, *MAC id*) pair from address resolution request packets regardless of the destination host being targeted.

Host Address Resolution Response

An address resolution response packet is a DIAG packet with *code* 8, *val* 31. It is used to notify a system of a host's lower layer address. The destination in the address segment describes the host that sent the address resolution request. The *source* field in the address segment describes the sender. The (*source*, *MAC id*) pair of the sender is the information being sought. When an end-system receives an address resolution response packet with the destination host address matching its own host address, the host caches the (*source*, *MAC id*) pair and continues any activities that were blocked awaiting this information. Address resolution response packets are sent directly to the target host and do not use the broadcast facilities of the underlying service provider.

Appendix D Service Profile Definitions

The *service* field of the Traffic Specifier indicates the type of service that is expected on this association. In general, XTP is a universal receiver protocol in the sense that the transmitter's actions cause the receiver to react. The service values do not weaken that statement. Rather, the profiles defined below for the *service* values give the expected settings for the options bits in XTP packets, as a guideline to implementors.

Table D-3 shows the six service types currently defined for XTP. These service types dictate the values for options bits used during the packet exchanges. Options bits that are not listed are to be used as appropriate; those that are listed must use the value assigned.

service		*Type of Service*
Decimal	*Hex*	
1	`0x01`	Traditional Unacknowledged Datagram Service
2	`0x02`	Acknowledged Datagram Service
3	`0x03`	Transaction Service
4	`0x04`	Traditional Reliable Unicast Stream Service
5	`0x05`	Unacknowledged Multicast Stream Service
6	`0x06`	Reliable Multicast Stream Service

Table D-3 — Service Type Values

The packet exchanges described below assume an error-free data transmission. For profiles where error detection and correction are necessary, the ECNTL packets will be used as defined by the protocol.

Traditional Unacknowledged Datagram Service

The traditional unacknowledged datagram service, *service* value `0x01`, is modelled after the UDP service. This data is not error controlled, and the closing semantics require that the receiver not respond. There is no indication of delivery. This service applies to either unicast or multicast.

Initiating Endpoint:

- Set the CTIMEOUT to 0 to disable the zombie state
- FIRST (and DATA, if necessary) packet *options*:
  - `EDGE` — off
  - `NOERR` — on
  - `RES` — off

- `NOFLOW` — off
- `FASTNAK` — off
- `SREQ` — off
- `DREQ` — off
- `RCLOSE` — on
- `WCLOSE` — on iff the end of the datagram is included
- `EOM` — off
- `END` — on iff the end of the datagram is included

Acknowledged Datagram Service

The acknowledged datagram service, *service* value `0x02`, uses a single packet to transmit data, but requests acknowledgement of the data's receipt. This data is error controlled, so the closing semantics require that the receiver respond. This service applies to both multicast and unicast.

Initiating Endpoint:

- Set the CTIMEOUT as appropriate.
- FIRST (and DATA, if necessary) packet *options*:
  - `EDGE` — off
  - `NOERR` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — on iff the end of the datagram is included
  - `DREQ` — off
  - `RCLOSE` — on
  - `WCLOSE` — on iff the end of the datagram is included
  - `EOM` — off
  - `END` — off

Corresponding Endpoint:

- Set the CTIMEOUT as appropriate. Zombie state is useful here.
- CNTL packet (in response to `SREQ` if no errors) *options*:
  - `EDGE` — off
  - `NOERR` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — off
  - `DREQ` — off

- `RCLOSE` — on
- `WCLOSE` — on
- `EOM` — off
- `END` — on

The corresponding endpoint goes into a zombie state for CTIMEOUT seconds.

Transaction Service

The transaction service, *service* value `0x03`, is only specified for unicast since there is no notion of a return data stream in the XTP multicast. A multicast transaction can be built, however, using the techniques described in Appendix E, "Multicast Extensions."

A transaction is a three-way exchange, as shown in Figure D-1: one or more packets are sent as a request, one or more packets are sent as a response only after the request has been completely received, and a final CTNL packet acknowledges the response and closes the association.

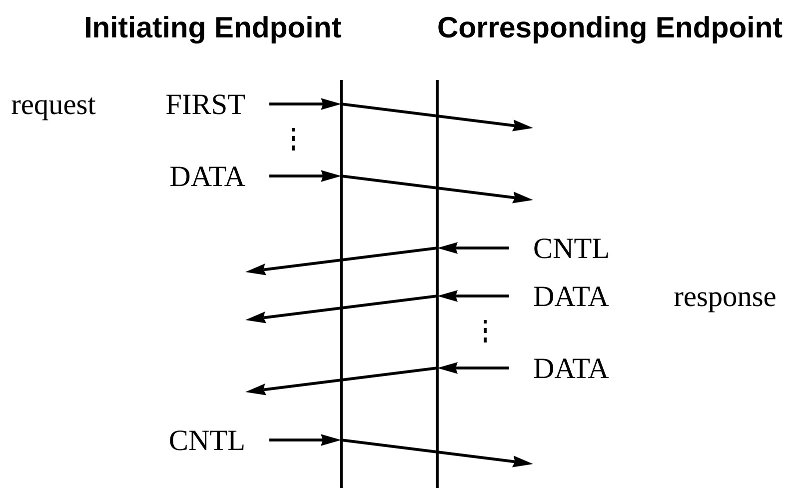

Figure D-1 — Transaction Service Profile

Initiating Endpoint:

- Set the CTIMEOUT as appropriate.
- FIRST (and DATA, if necessary) packet (for sending request) *options*:
  - `NOCHECK` — off
  - `EDGE` — off
  - `NOERR` — off
  - `MULTI` — off
  - `RES` — off

  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — on iff the end of the transaction request is included
  - `DREQ` — off
  - `RCLOSE` — off
  - `WCLOSE` — on iff the end of the transaction request is included
  - `EOM` — on iff the end of the transaction request is included
  - `END` — off

Corresponding Endpoint:

- Set the CTIMEOUT as appropriate.
- CNTL packet (in response to `SREQ` if no errors) *options*:
  - `NOCHECK` — off
  - `EDGE` — off
  - `NOERR` — off
  - `MULTI` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — off
  - `DREQ` — off
  - `RCLOSE` — on
  - `WCLOSE` — off
  - `EOM` — off
  - `END` — off
- DATA packets (for sending response) *options*:
  - `NOCHECK` — off
  - `EDGE` — off
  - `NOERR` — off
  - `MULTI` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — on iff the end of the transaction response is included
  - `DREQ` — off
  - `RCLOSE` — on
  - `WCLOSE` — on iff the end of the transaction response is included
  - `EOM` — on iff the end of the transaction response is included
  - `END` — off

Initiating Endpoint:

- CNTL packet (in response to `SREQ` if no errors) *options*:
  - `NOCHECK` — off
  - `EDGE` — off
  - `NOERR` — off
  - `MULTI` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — off
  - `DREQ` — off
  - `RCLOSE` — on
  - `WCLOSE` — on
  - `EOM` — off
  - `END` — on

The initiating endpoint goes into a zombie state for CTIMEOUT seconds.

Traditional Reliable Unicast Stream Service

The traditional reliable unicast stream service provides reliable full-duplex data transfer as provided by TCP, as shown in Figure D-2. (Note that the DATA packets may be sent from either endpoint at any time after the establishment of the association, and either side may send a `WCLOSE` bit first.)

Initiating Endpoint:

- Set the CTIMEOUT to 0 to avoid the zombie state
- FIRST packet *options*:
  - `NOCHECK` — off
  - `EDGE` — off
  - `NOERR` — off
  - `MULTI` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — on
  - `DREQ` — off
  - `RCLOSE` — off
  - `WCLOSE` — on iff last of data is included
  - `EOM` — off
  - `END` — off

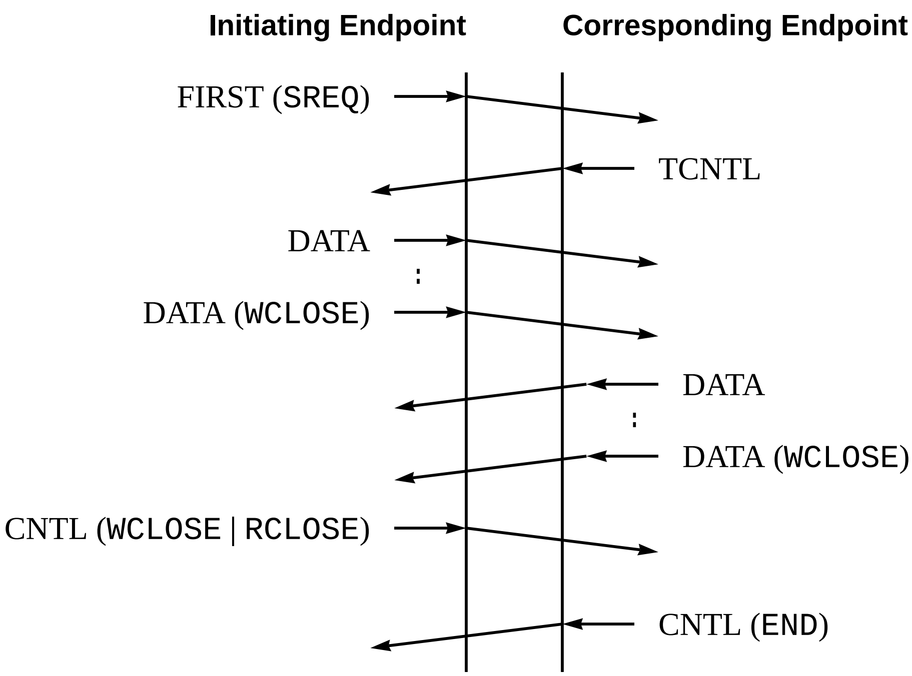

Figure D-2 — Traditional Reliable Streams Service

- DATA packet *options*:
  - `NOCHECK` — off
  - `EDGE` — off
  - `NOERR` — off
  - `MULTI` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — on iff last of data included, else as appropriate
  - `DREQ` — off
  - `RCLOSE` — on iff `WCLOSE` seen and all data received
  - `WCLOSE` — on iff last of data is included
  - `EOM` — off
  - `END` — off
- CNTL packet *options*:
  - `NOCHECK` — off
  - `EDGE` — off
  - `NOERR` — off
  - `MULTI` — off
  - `RES` — off
  - `NOFLOW` — off
  - `FASTNAK` — off

    - `SREQ` — as appropriate
    - `DREQ` — off
    - `RCLOSE` — on iff `WCLOSE` seen and all data received
    - `WCLOSE` — on iff last of data sent
    - `EOM` — off
    - `END` — off

Corresponding Endpoint:

- Set the CTIMEOUT as appropriate
- TCNTL packet (in response to `SREQ` in FIRST packet) *options*:
    - `NOCHECK` — off
    - `EDGE` — off
    - `NOERR` — off
    - `MULTI` — off
    - `RES` — off
    - `NOFLOW` — off
    - `FASTNAK` — off
    - `SREQ` — off
    - `DREQ` — off
    - `RCLOSE` — on iff `WCLOSE` seen and all data received
    - `WCLOSE` — on iff no data to be sent
    - `EOM` — off
    - `END` — on iff `WCLOSE` and `RCLOSE` have been seen
- CNTL packet *options*:
    - `NOCHECK` — off
    - `EDGE` — off
    - `NOERR` — off
    - `MULTI` — off
    - `RES` — off
    - `NOFLOW` — off
    - `FASTNAK` — off
    - `SREQ` — as appropriate
    - `DREQ` — off
    - `RCLOSE` — on iff `WCLOSE` seen and all data received
    - `WCLOSE` — on iff last of data sent
    - `EOM` — off
    - `END` — on iff `WCLOSE` and `RCLOSE` have been seen
- DATA packet *options*:
    - `NOCHECK` — off
    - `EDGE` — off
    - `NOERR` — off
    - `MULTI` — off

- `RES` — off
- `NOFLOW` — off
- `FASTNAK` — off
- `SREQ` — on iff last of data included, else as appropriate
- `DREQ` — off
- `RCLOSE` — on iff `WCLOSE` seen and all data received
- `WCLOSE` — on iff last of data is included
- `EOM` — off
- `END` — off

The corresponding endpoint goes into a zombie state for CTIMEOUT seconds.

Unacknowledged Multicast Stream Service

The unacknowledged multicast stream provides transmission of data with no data transfer reliability and no group membership reliability. The multicast transmitter will emit the data, but will be unaware of whether any receivers receive it. This scenario parallels the situation with broadcast radio and television; the transmitter transmits, but does not know if any of its transmissions are received, or by whom.

Initiating Endpoint:

- FIRST (and subsequent DATA) packet *options*:
  - `NOERR` — on
  - `MULTI` — on
  - `NOFLOW` — on
  - `FASTNAK` — off
  - `SREQ` — off
  - `DREQ` — off
  - `RCLOSE` — on
  - `WCLOSE` — off
  - `EOM` — off
  - `END` — on iff the association is to be terminated

Reliable Multicast Stream Service

The reliable data multicast transfer requires that the group membership be stable and that all data be transferred reliably. This scenario is achieved by having the multicast transmitter emit FIRST and subsequent DATA packets with `MULTI` and `RCLOSE` both set, but with `NOFLOW` and `NOERR` both clear. These settings assure that the data flow itself will be fully error-controlled and thus fully reliable.

To ensure group membership reliability, the multicast transmitter must set `SREQ` on the FIRST packet. This allows XTP to build the data structure identifying group members. Whenever `SREQ` is set in an outgoing packet, XTP will solicit a status report from each receiver (including a synchronizing handshake, if necessary). From these responses the multicast transmitter can determine the current state of each group member (both identity and progress). If any of the group members do not reply, then the multicast association can be aborted by having the multicast transmitter send an `END` bit to the multicast group. If the reliability semantics permit the group to continue after a group member dies, this can be effected by having the multicast transmitter send the non-responsive member an `END` bit on the receiver's unicast address, thus removing it from the group. By setting CTIMEOUT to an appropriate value, this allows the designer to bound the amount of time (equal to CTIMEOUT interval) that the total group will stall, waiting for a receiver to respond, before deleting that member from the group and continuing without it.

Initiating Endpoint:

- Set the CTIMEOUT as appropriate.
- FIRST (and subsequent DATA) packet *options*:
  - `NOCHECK` — off
  - `NOERR` — off
  - `MULTI` — on
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — on, and as appropriate
  - `RCLOSE` — on
  - `WCLOSE` — on iff the end of the data is included
  - `END` — off

Corresponding Endpoint:

- Control packet (in response to `SREQ`) *options*:
  - `NOCHECK` — off
  - `NOERR` — off
  - `MULTI` — on
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — off
  - `RCLOSE` — on iff `WCLOSE` seen and all data received
  - `WCLOSE` — on
  - `EOM` — off
  - `END` — off

Initiating Endpoint:

- CNTL packet (to end multicast association) *options*:
  - `NOCHECK` — off
  - `NOERR` — off
  - `MULTI` — on
  - `NOFLOW` — off
  - `FASTNAK` — off
  - `SREQ` — off
  - `RCLOSE` — on
  - `WCLOSE` — on (unless this is an abort)
  - `EOM` — off
  - `END` — on iff `RCLOSE` seen from all receivers, or if abort

Appendix E Multicast Extensions

XTP multicast mechanisms support reliable 1-to-*N* communication, yet there are many multi-party communication paradigms. Unfortunately, collecting a set of appropriate abstractions can cause a protocol specification to become overly complex. In addition, it is not the place of a protocol specification to determine the set of paradigms; rather, the protocol should provide tools upon which communication services can be built. In this respect, the XTP multicast mechanisms can be used to build interesting extensions to the 1-to-*N* multicast service. Examples of such extensions are given here.

Concentration and Cloning

Concentration and cloning are optional extensions to XTP multicast that can be built using various multicast and unicast mechanisms in XTP. Concentration is a reliable transmission of arbitrary messages from a set of hosts in the multicast group to a single host. This is the inverse of multicast, where multiple data streams are concentrated into one receiving host. If data from *N* hosts are 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 unicast XTP association.

The technique of cloning, described below, can be used to improve efficiency of the concentration. If a large number of concentration channels are needed, there may be an advantage to creating 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 packets.

N-by-*M* Communication

XTP multicast mechanisms support reliable 1-to-*N* communication. Application data flows involving *N* data sources sending to *M* data sinks, or *N*-by-*M* communication, must be constructed using multiple XTP associations. The following discussion is an illustration of how an *N*-by-*M* service can be derived from the basic XTP multicast facilities. In particular, note the use of XTP cloning and of a *transport layer bridge* to multiplex *N* data streams onto a single output channel.

The *N*-by-*M* service described here is a message-based, reliable, ordered, multi-peer service. A set of *N* communicants concurrently transmit messages to each other (symmetric group communication), and the messages are reliably delivered to each member in the group with a mutually consistent ordering at *M* receiving sites. This scheme for atomic multicasting between peers in a distributed application exhibits strong reliability and a simple client-server structure.

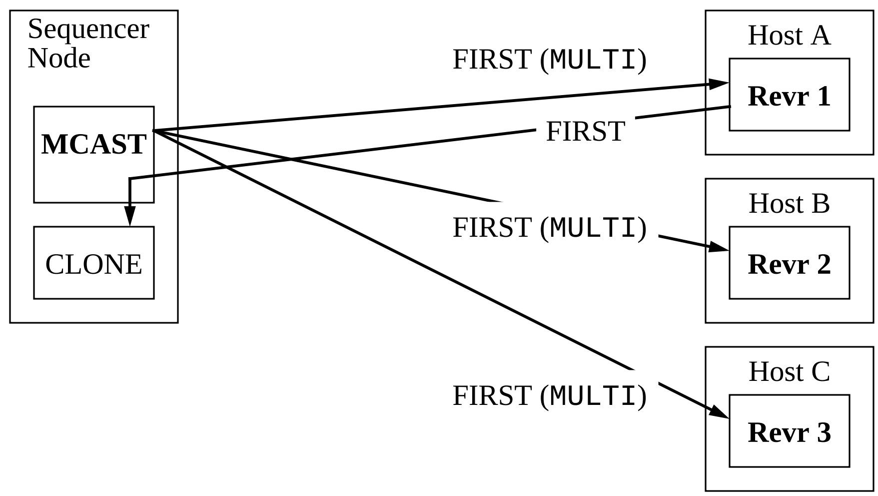

Figure E-1 — *N*-by-*M* Connection Setup

Communication set-up for the *N*-by-*M* service is shown in Figure E-1. An application entity at the *sequencer node* for the group sets up a reliable multicast connection with the receivers in the multicast group. When a receiver accepts the FIRST packet sent to establish the reliable multicast association, the receiver sends a unicast FIRST packet back to the sequencer node. Using cloning, the multicast transmitting context at the sequencer transparently establishes a reverse channel with each group member. That is, each group member now has a reliable XTP unicast association to the sequencer node.

Figure E-2 illustrates message transfer in the atomic *N*-by-*M* multicast service. Group members inject their messages into the network asynchronously (1). A message is first sent to the sequencer (2) where the message is sent out on the reliable multicast connection (3 and 4). This relaying of messages is performed by a transport layer bridge (3), that is, a mechanism that controls the multiplexing of data from a set of receiving contexts into a single (multicast) transmitting context. In this case, the transport layer bridge must preserve message boundaries when forwarding data from the back-channels into the outgoing multicast channel. Flow and rate control on the incoming back-channels can be used to throttle members. The XTP associations ensure reliable delivery and a mutually consistent message ordering of the global message stream at each group member. The sequencer node detects receiver failure using its knowledge of the active group membership through the XTP group management facility.

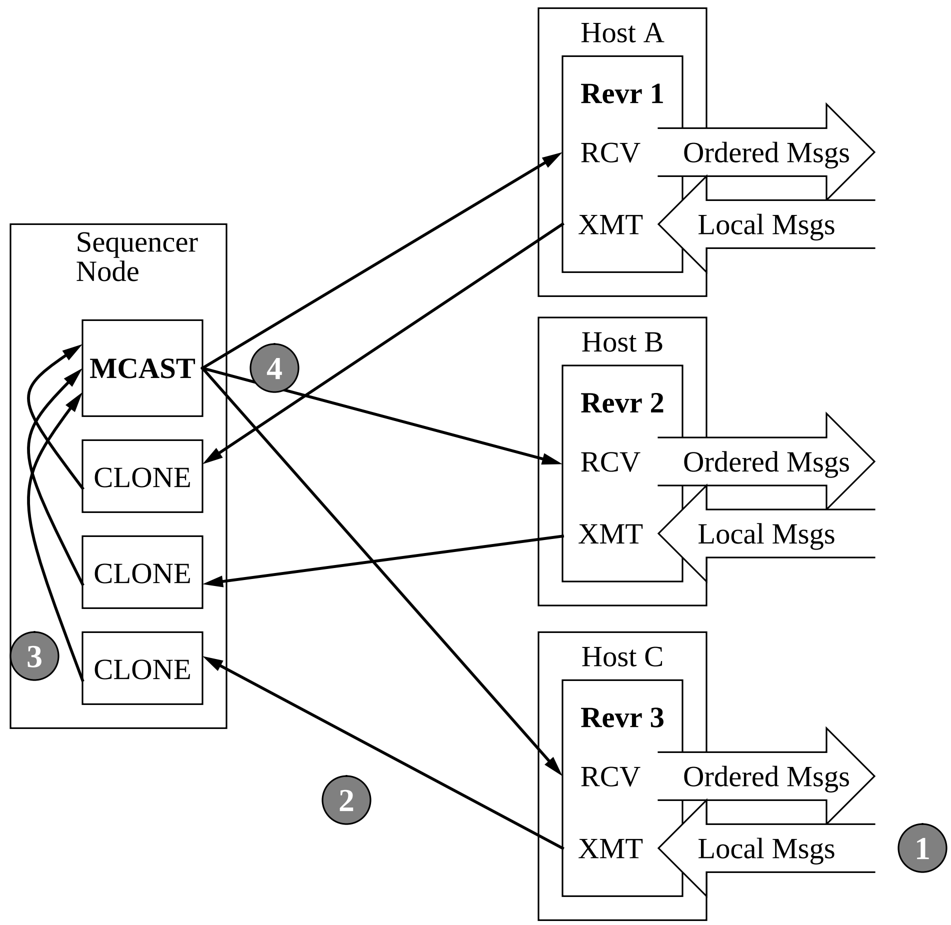

Figure E-2 — *N*-by-*M* Reliable Ordered Multicast