MASTER

UCID-19293

DELTA-T PROTOCOL SPECIFICATION

small tee

Richard W. Watson

December 4, 1981

Lawrence Livermore Laboratory

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under Contract W-7405-Eng-48.

UCID--19293

DE82 010601

Delta-t PROTOCOL SPECIFICATION
(Working Draft)

R. W. Watson
Lawrence Livermore National Laboratory
December 4, 1981

Table of Contents

1. Introduction
2. Delta-t Services and Mechanisms
    2.1 Introduction
    2.2 Addressing
    2.3 Delta-t Assotiation
    2.4 Protection
    2.5 Assurance
        2.5.1 Introduction
        2.5.2 Lost Packets
        2.5.3 Duplicate Packets
        2.5.4 Missequenced Packets
    2.6 Synchronization Including Connection Management
    2.7 Resource Management
        2.7.1 Segmentation
        2.7.2 Flow Control
        2.7.3 Window Management
    2.8 Diagnostics and Measurement
    2.9 Services Not in Delta-t
    2.10 Future Services
3. Services Required of the Network Layer
    3.1 Data Objects and Addressing
    3.2 Protection
    3.3 Assurance
    3.4 Resource Management
    3.5 Synchronization Including Connection Management
    3.6 Control Information
4. Model of the Delta-t Environment
    4.1 Environment Model
    4.2 Event Handling
5. Delta-t Use of DeltaGram Header Fields
    5.1 Introduction
    5.2 Data Packets
    5.3 Ack Packets
    5.4 Direct Control Packets
    5.5 Nak Packets
6. Delta-t Operation
    6.1 Introduction
    6.2 Connection Record Definition and Management
        6.2.1 Introduction
        6.2.2 Sender Initialization
        6.2.3 Receiver Initialization
        6.2.4 Allocation and Deallocation of State
        6.2.5 Connection Record Definition
    6.3 The Delta-t Module Global Declaration Environment
    6.4 Timer Event Handling and Retransmission Procedures
        6.4.1 DtTimeout
        6.4.2 Handling Retransmission
        6.4.3 Send Timer Expiration
    6.5 User Interface Events
        6.5.1 Receive or Ack Generation Events
        6.5.2 Data or Rendezvous Packet Send

6.5.2.1 DtStartData and DtFinishData
6.5.2.2 Sending a Data Packet
6.5.2.3 Sending a Rendezvous Packet
6.6 Packet Received Event
6.6.1 DtPktRcvd
6.6.2 Receipt of a Data Packet
6.6.3 Receipt of an Ack Packet
6.6.4 Receipt of a Rendezvous Packet
6.6.5 Receipt of a Nak Packet
6.6.6 Sending a Nak Packet
Acknowledgment
References
Appendix A - Notes on Timer Values and Rules
Appendix B - EIM Interface State Record Definition and IPC User Interface Operations

1. Introduction

This document is one of a series describing protocols associated with the Livermore Interactive Network Communication System (LINCS) hierarchical architecture [4,15,18]. At the heart of LINCS is its basic interprocess communication (LINCS-IPC) service [21]. LINCS-IPC defines a reliable, flow controlled, full duplex, uninterpreted, labeled bit stream communication service. LINCS-IPC is level 4 in the LINCS architecture. Level 3 of LINCS is the Network layer defining an internetwork datagram type service [19]. LINCS-IPC interfaces to User processes that utilize higher level syntactic and semantic conventions for process interaction [20]. The transport service provided by the Delta-t protocol can be considered a sublayer of the LINCS-IPC layer. Delta-t augments the Network level service as required to support LINCS-IPC. This document specifies the services provided by the Delta-t protocol to support LINCS-IPC, the operation of Delta-t, and the services Delta-t requires of the Network level.

This document was written to be self-contained but the reader will find it useful to have available for reference the LINCS-IPC and LINCS DeltaGram Network layer protocol specifications [19,21].

Implementations are underway in Pascal for the PDP-11 running under RT11 and RX11, in BLISS for the VAX running under VMS, in MODEL for the CRAY-1 and CDC 7600 running under NLTSS and LTSS, and for the SEL 32/75 running under PORT.

2. Delta-t Services and Mechanisms

2.1 Introduction

Delta-t logically supports a permanent, reliable, flow controlled, full duplex, labeled bit stream connection between two ports. There is no extra packet exchange overhead to reliably manage connection state as in other stream oriented protocols [3,10,12]. Therefore Delta-t can support an efficient, low delay, minimum packet exchange, reliable transaction oriented service as well as high stream throughput.

The Delta-t protocol, as defined here, assumes the services of a datagram protocol, the DeltaGram protocol [19]. The decomposition of services between Delta-t and DeltaGram was made by determining which services required intermediate routing node support and those that must be performed end-to-end or could be most efficiently handled at these points.

Below we outline and discuss both the user services visible at the next higher level interface to Delta-t and the internal protocol mechanisms used to support these. Appendix B outlines the logical functionality of an interface to the LINCS-IPC service supported by Delta-t [21]. The next higher level interface used in this specification is a lower level interface internal to the LINCS-IPC layer.

2.2 Addressing

Communication within the LINCS architecture takes place between ports. Ports are identified by 64 bit LINCS addresses. Ports are bound to processes. Port to process binding is a higher level issue of no concern to Delta-t. Actual data movement between ports is supported by the Network layer (DeltaGram) protocol. Therefore no additional addressing structure is provided by Delta-t.

2.3 Delta-t Association

An unordered port pair defines a full-duplex data channel called an association. Delta-t detects and recovers from lost, duplicated, and missequenced data. Damaged data is detected and discarded (lost) by the Network layer. Delta-t labels data bits with a protection level and optionally also with B and/or E synchronization marks (see Section 2.6). Internally Delta-t also labels bits with a sequence number, version, lifetime, and other control information. Data transfer on an association is flow controlled.

The state information at each end necessary to provide these services logically always exists for all possible associations (permanent connections). After appropriate timeouts, the state information is reset to default values. When this state information has a default value it can be deallocated and does not need to be maintained by the implementation. Management of this state (connection management) is under timer control and does not require user interface primitives or special opening and closing packet exchanges [14,16]. The state at each end is kept in connection records (CR).

Delta-t's assurance, flow control, and connection management mechanisms are outlined in the appropriate following sections.

2.4 Protection

The protection level of Delta-t data is passed through to the Network layer which enforces an appropriate protection policy [5,19]. Encryption, if required to convert untrusted links (or subnetworks) connecting trusted nodes into trusted links, is assumed to take place at the Link level of the LINCS architecture. Receiver buffer space is protected by association identifier outside Delta-t within LINCS-IPC (see Appendix B). Additional access control services are defined at higher levels of the architecture.

2.5 Assurance

2.5.1 Introduction

Delta-t guarantees data will not be lost, duplicated or missequenced. The Network layer provides optional damage detection and discard on a per packet basis. If packet segmentation is not required, then this protection is end-to-end. Whenever segmentation occurs, the Network layer provides hop-by-hop protection with no unprotected gaps. It is assumed that the next higher level interface will maintain (or allow the user to determine) the sequence of data sent and received on a given association.

Delta-t provides for data assurance through data sequence numbers (SNs) on bits, a positive-acknowledgment/retransmission mechanism, and bounds on packet lifetime. The mechanisms used to detect and recover from lost, missequenced, and duplicate packets are identical to those used in many other transport protocols [11,13]. Delta-t's timer based connection management is, however, unique [6,16]. A negative acknowledgment (Nak) is also provided as an efficiency and diagnostic aid, although it is not essential to correct protocol operation. The Delta-t assurance mechanisms are now outlined.

2.5.2 Lost Packets

Delta-t detects and recovers from lost packets by positive acknowledgment and retransmission. The origin transmits a packet and then waits an interval for a positive acknowledgment (Ack). This interval is usually slightly longer than the average round trip time for a packet and its Ack to be generated and traverse the network. If an Ack is not forthcoming in that interval, the unacknowledged packet is retransmitted. If no positive acknowledgment is received after attempting some number of retransmissions (giveup time), an error is reported to the user with an indication of the successfully Acked data and of data transmitted but not Acked. A giveup timeout can result through failure of data to be delivered or failure of Acks to be returned. Delta-t level information cannot determine which case occurred. Either case could occur from an end-node computer crash or serious network problem such as a partition. A higher-level recovery mechanism, using conventions on the B/E marks or higher-level delimiters, is required if the ambiguity needs to be removed. Delta-t has been designed to limit the cases where this ambiguity can occur to situations such as end-node system crashes and serious network faults which are outside of its ability to detect and recover.

The choice of retransmission interval is an important factor affecting average packet delay and network efficiency. If the interval is too long, large average delays can result. If the interval is too short, average delay may be less, but network efficiency is decreased due to the possibility that packets may be retransmitted unnecessarily. This choice is complicated in an environment where average delay is quite route dependent.

If a packet is detected as damaged, if its lifetime expires, or if another delivery problem exists within the routing network or at the destination, a Nak packet is returned to the origin. This information may be used to trigger retransmission and may be recorded as a hint for diagnostic purposes.

An acknowledgment mechanism is based on being able to identify the units acknowledged. In Delta-t bits are numbered sequentially with a sequence number. An Ack indicates the SN of the next bit the receiver expects to receive. The acknowledged SN (ASN) implies acknowledgment of all previous SNs. Therefore, if an Ack is lost, Acks of succeeding bits acknowledge preceding bits. Similarly, duplication of Acks will cause no difficulty because they just confirm what is already known.

The size of the field chosen to represent SNs is finite and therefore SN arithmetic is performed modulo 2^n, where n is the number of bits in the SN field. In Delta-t n = 32. Because SNs wrap around, care must be taken to avoid having two different bits or their Acks with the same SN in the network at once. Because Naks are used strictly as an efficiency or diagnostic hint Nak ambiguity is not an assurance problem. If we define the term MPL to stand for either the longest time a packet can exist, or is estimated to exist or is desired to exist in the network (maximum-packet-lifetime), R as the maximum time a sender will keep retransmitting a packet, A as the maximum time a receiver will wait before sending an Ack, and T as the maximum new SN generation rate (often maximum transmission rate), then, assuming new bits are transmitted at the maximum rate even while retransmission takes place, the following inequality must be satisfied to meet the above unique SN condition:

$$2^n > (2*MPL + R + A)T.$$

This inequality assures that a sender generating SNs at the maximum rate will not reuse an SN until it is guaranteed that an SN and any Acks of it have arrived or no longer exist in the network.

2.5.3 Duplicate Packets

SNs are also used for duplicate detection. At any point in time the receiver knows what SN it is expecting next. We call this SN the left-window-edge (LWE), because at any point in time, for assurance and flow control reasons, the receiver is only willing to accept bits with SNs within a particular range called the acceptance window. SNs less than the LWE are duplicates [17]. Duplicates are discarded and become lost. The mechanism of the previous section is used for recovery.

2.5.4 Missequenced (out-of-order) Packets

A missequenced bit is one with an SN not equal to the LWE but within the acceptance window. Two implementation choices exist for handling a missequenced bit:

(1) it can be held (its lifetime continues counting down) until its predecessors arrive, on the assumption they will follow shortly and all can be Acked before the sender's retransmission interval

elapses, thus increasing efficiency, or

(2) it can be discarded, with retransmissions providing for correct ordering thus simplifying the implementation.

The model of Section 6 assumes the latter.

2.6 Synchronization Including Connection Management

Delta-t's synchronization services support bits being labeled with B and E marks (B-bit and E-bit respectively) and a guarantee of sequenced data delivery. Use of B- and E-bits is determined by higher level convention. The purpose of the B-bit is to label the beginning of a higher level data unit, such as a message [20]. It provides a synchronization mark in the data stream where parsing or other operation can safely begin. This function is provided in other transport protocols by explicit connection opening packet exchanges. The purpose of the E-bit is twofold, to label the end of a higher level data unit and to indicate a required higher level wakeup point.

Internally Delta-t supports sequenced data delivery using SNs. Delta-t provides reliable management and synchronization of the state at each end by a timer mechanism. Reliable connection management is a subtle area discussed in detail in references [1,7,14,16] and Appendix A. Here we briefly outline the simple timer mechanism used by Delta-t for connection management.

Conceptually, there are three main phases in connection management (explicit phase separation is not required in Delta-t): (1) initializing (opening) the connection records at each end to nondefault values, (2) evolving the state during ongoing data transfer, and (3) resetting or terminating (closing) state information when no further data needs to be transferred. During the reliable opening of a transport protocol assurance connection, the main problem is establishing initial SNs meeting the following opening conditions:

O1: If no connection state exists or it is in the default state, (connection closed) and the receiver is willing to receive, then no packets from a previous connection should cause a connection to be initialized and duplicate data to be accepted.

O2: If a connection exists, then no packets from a previous connection should be acceptable within the current connection.

In order to avoid ambiguity about the state of data sent, connections should be closed in a way allowing each side to know that the other side has received any data sent (a graceful close). This implies two closing conditions:

C1: A receiving side must not close until it has received all of a sender's possible retransmissions and can unambiguously respond to them, and

C2: A sending side must not close until it has received an Ack for all its transmitted data or allowed time for an Ack of its final retransmission to return before reporting a giveup failure.

Delta-t's timer-based approach meets the connection management conditions above by having both sender and receiver maintain connection records long enough to guarantee that all duplicates have died out, information flow is smooth (all bits sent that could be acceptable are accepted), and all transmissions, retransmissions, and Acks have arrived at their destination, if they are ever going to arrive. The connection records at each end of an association are under control of a receive-timer (Rtimer) and send-timer (Stimer) respectively. No synchronization between timers is required, other than that provided by the sending and receiving of packets, but it is assumed that the timers at each end run approximately at the same rate; that is, over an interval of $3\Delta t$ (see below for Δt definition) there is no significant drift. For reasonable Δt intervals (less than 1 to 2 minutes) this assumption is easily satisfied with current clock specifications. The Rtimer interval guarantees that the receiver maintains its connection record long enough to (1) detect all duplicates and (2) guarantee that acceptable SN's will reach the receiver. While Rtimer > 0 the receiver will only accept packets with SNs in its acceptance window. The Stimer interval must be such that (1) the sender's connection record be maintained as long or longer than the receiver's, in order for the sender to be sure to generate acceptable SNs, (2) it is long enough to recognize all Acks that it may receive and (3) it will not reuse a SN until all previous data packets and their Acks using that SN have died.

The rules for timer intervals, control of the timers, setting of packet header control flags, SN selection, and packet acceptance are developed in [6] and Appendix A. They are quite simple. We define the quantity,

> $\Delta t = MPL + R + A$, where MPL is a worst case estimate of the time for traversing the network and R and A are as defined earlier.

Safe values for use in initializing the timers are:

$$\text{receive-time} = 2*\Delta t$$
$$\text{send-time} = 3*\Delta t.$$

R.1) Stimer is refreshed whenever a new SN (i.e. a new data bit or Rendezvous packet) or reliable-Ack is sent (see Section 2.7.3 for discussion of rendezvous and reliable-Acks).

R.2) Once a bit b_i has had its maximum retransmission time (or equivalently maximum number of retransmissions) no new bits can be transmitted until b_i has been Acked; bits b_{i+k} which had previously been transmitted can continue being retransmitted until their maximum retransmission time.

R.3) Rtimer is refreshed whenever a new SN is accepted or data overflow occurs.

R.4) When Rtimer expires, the receive state is reset to its default values.

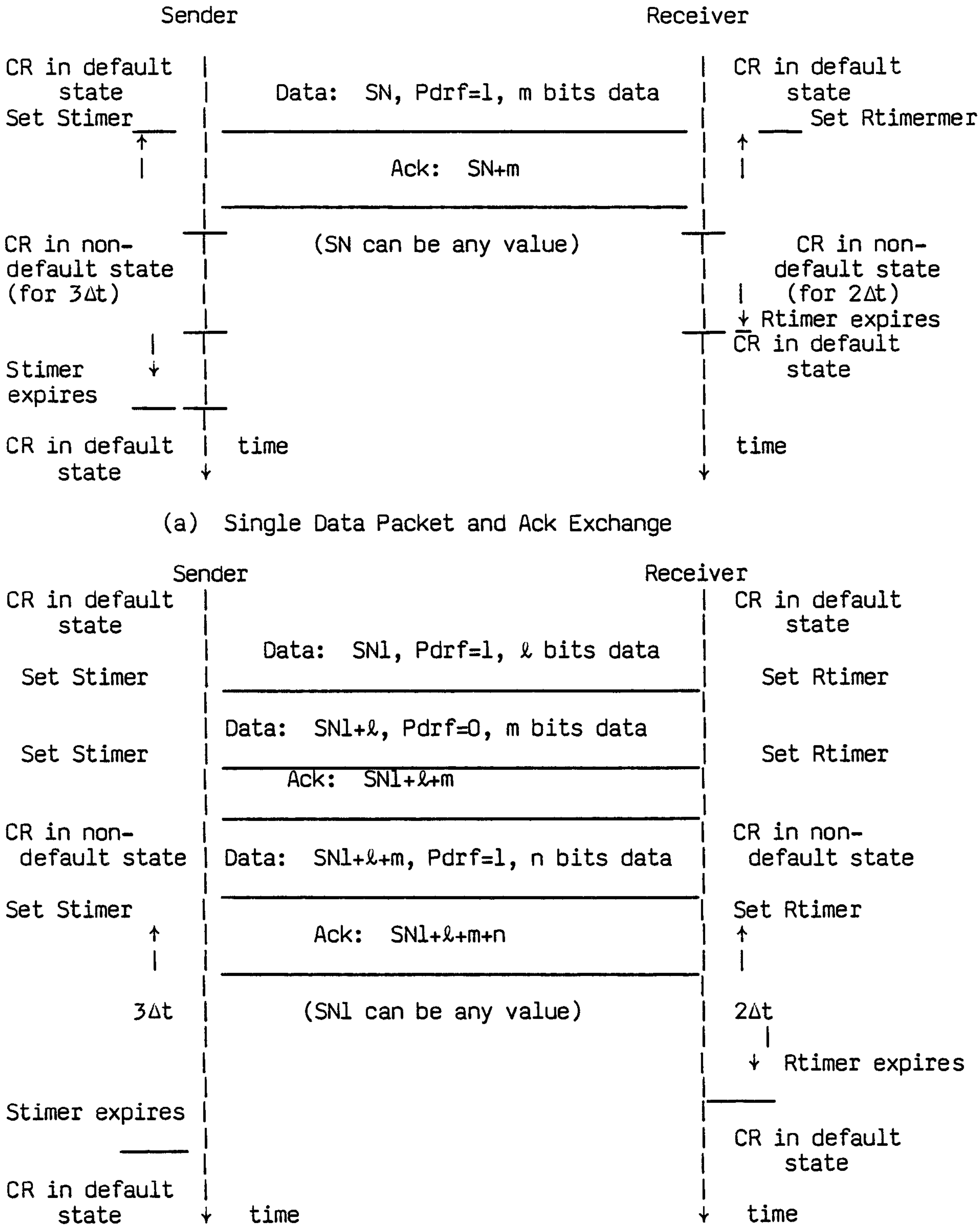
R.5) Once a bit or Rendezvous or reliable-Ack is initially transmitted its lifetime is set equal to Δt and starts counting down.

R.6) At the point an SN is tested for acceptance, the lifetime of any Ack packet generated is set equal to Δt and begins counting down.

R.7) When Stimer expires (giveup timeout) the send state is reset to its default values, any initial SN can be used when new data needs sending, and if unAcked SNs exist a giveup error is reported.

Delta-t packet headers label their first bits with a Data-Run-Flag (Pdrf), set 1 in packets sent when all previously sent SNs have been acknowledged, allowing receivers to detect missequenced packets before it has initialized its state [6]. When the Rtimer is nonzero only packets with SNs in the acceptance window can be accepted and the Pdrf value can be 0 or 1. When the Rtimer is zero only a packet with Pdrf=1 is acceptable. If the Stimer is nonzero, then the next contiguous SN to that contained in the connection record must be used when a new bit is to be sent. If the Stimer is zero, then any initial SN can be used because no packets for the association exist in the network.

With the above mechanism no exchanges of packets are required to reliably open or close connections. A sender's connectionrecord is "opened" automatically, i.e., holds nondefault state, when SNs are sent. A receiver's connection record is "opened" automatically when acceptable SNs are received. Each record is returned to its default state when sending and receiving activity cease or pause because Stimer and Rtimer go to zero. Therefore, connection records are automatically maintained only when needed. Also no problems exist when both ends of an association simultaneously begin sending. Figure 2.1 illustrates two common cases of packet exchange and CR management.



(b) Example Multiple Data Packet Exchange

Figure 2.1. Example Packet Exchanges and CR State (for simplicity data exchange in only one direction shown.)

2.7 Resource Management

2.7.1 Segmentation

Delta-t supports a bit stream service. The bit stream at the user interface may be segmented into buffers in an actual implementation. Segmentation of the bit stream into packets is an internal Delta-t implementation issue. If packets need further segmentation during packet transport that is handled by the DeltaGram protocol.

2.7.2 Flow Control

There are still many questions needing answers in the flow control area, particularly related to how flow control interacts with buffer management, retransmission, and other protocol and implementation mechanisms [8]. Throughput is dependent on the interaction of these issues. Flow control mechanism design problems arise from the desire for a mechanism and choice of identifiable flow control unit(s) that reflect the nature of the several resources being protected (e.g., user and system buffers, CPU cycles, interface access) and yet allows efficient transmission on an association, independent of the widely varying implementation choices possible. Until we feel we understand the issues better we have chosen for this version of Delta-t the simple window or credit flow control mechanism. It works as follows.

Each Ack packet contains a window (credit) field indicating the additional number of bits of data, relative to the ASN, that the receiver can accept. In effect, the quantity (ASN + window - 1) indicates the highest SN the receiver is willing to accept. In Delta-t this information is advisory only. Receivers may renege on window promises, or senders can send more. Overflow of the receiver's window will result in the overflow data being discarded. Sender or receiver strategies that result in frequent overflow will cause inefficient use of resources. Therefore, sending and receiving strategies should be such that this is an infrequent event.

2.7.3. Window Management

The receiver must implement a policy for determing what window to advertise. The policy chosen can be a function of user or system buffer space available for an association, based on statistical management of a buffer pool, etc. Similarly a sender must implement a transmission policy relative to the receiver's advertised window, and as information is sent, adjust its estimate of the receiver's window. A range of policies are possible in each of these areas. The optimum policy is dependent on receiver operation and buffer management strategy, normally unknown to the sender.

Choice of these policies as well as protocol mechanisms supporting reliable window exchange is called window management. While choice of these policies and their interaction can significantly affect performance our level of understanding is such that this specification cannot provide much in the way of explicit guidance, except as follows. First sending policy.

The sender must update its estimate (output window) of the receiver's available input window according to the following rule: As each bit is sent decrement by one the output window, unless the bit is labeled with an E, delimiting a higher level data unit. In the latter case the sender should assume the available output window goes to zero. The returning Ack of the E-bit will update this appropriately. (Note: The send window remains zero until the E-bit is Acked.) The motivation for this rule is to cover the case where the receiver may be implementing a block buffer strategy (first bit address and count) and complete a buffer once an E-bit is placed in it, thus invalidating any previously advertised window.

Higher level LINCS conventions restrict use of message boundaries to only define wakeup points in the data stream. In a LINCS control stream this is a point where an action, and normally a reply, is expected and, therefore, pipelining of control messages is not required. Data is transferred as specified in a control message, in a single data message. Data message pipelining is not expected. Therefore, the pause in data sending resulting from assuming a zero window when an E-bit is sent will not cause a performance degradation.

The discussion to follow contains more motivation than that for other mechanisms because the issues are not documented elsewhere. A question that sending policy must answer is the following. When the state record at the sender indicates that the receiver has less space than it has data to send (particularly a zero length output window) and all data sent has been Acked, how long should it wait before attempting to send? The answer must consider the problem resulting from the possibility of missequenced or lost Ack packets [7] and that the receiver may be using small buffers and the window may remain small. If a receiver sends an Ack packet advertising a window, then sends another packet advertising a larger window, and the latter arrives first, the sender's state will indicate that a window renege has taken place and thus the sender may not send while it awaits a new larger window indication. A similar problem results if the Ack packet indicating an increased window gets lost. The receiver may also have a long delay before a window increases.

We consider two cases sending into a small but positive window and sending into a zero window; first the positive window case. We assume that receiver action is indicated by an E-bit, therefore, a sender must always send as much data as the output window allows when there is an E-bit to send or a maximum size packet can be filled. Once data has been sent it will be retransmitted until an Ack is received, thus providing for reliable window update. The sender might also have a timer (not modeled in this specification) to force sending into a smaller positive window than desired for packet handling efficiency.

Now consider the problem of a zero window. When the sender receives an Ack indicating a zero window there are two cases, either the sender has more data to send or it does not. In the latter case, expected to be common in a distributed operating system or transaction environment, no more packet exchange need take place. Each end's state records will timeout and be discarded. When the sender again has more to send, it will do so. In the former case the sender wants to wait for the receiver to reliably indicate that the window has opened. What is desired is a mechanism to assure a reliable window opening without the inefficiency of the sender polling the receiver [12] or the receiver constantly sending Acks on inactive connections [3,10].The sender must indicate once to the receiver that it should reliably signal window opening when it occurs.

The mechanism is the following and is illustrated in Figure 2.2. When the sender's state indicates that all data sent have been Acked, there is data to send, and a zero window exists, it sends what is called a Rendezvous packet indicating that it wants to be informed (rendezvous-at-the-sender) when the window goes positive (which might be immediately). The Rendezvous packet contains a field that consumes SN space protecting it against duplication or missequencing. Since it is only sent when all previous data have been Acked, none of the usual difficulties that can result from including control information in SN space exist [7]. The Rendezvous packet is retransmitted until Acked (or its retransmission interval expires), thus protecting it against loss. When the receiver's input window opens and it is in the rendezvous-at-sender state, it will send a specially labeled Ack packet and at retry intervals retransmit this packet until it receives an acceptable Data packet (which in effect "Acks" it), thus protecting the "window opening" Ack (reliable-Ack) against loss. Duplication or missequencing of these Acks at most cause extra packet exchanges and are not assurance hazards. We now discuss issues associated with window overrun.

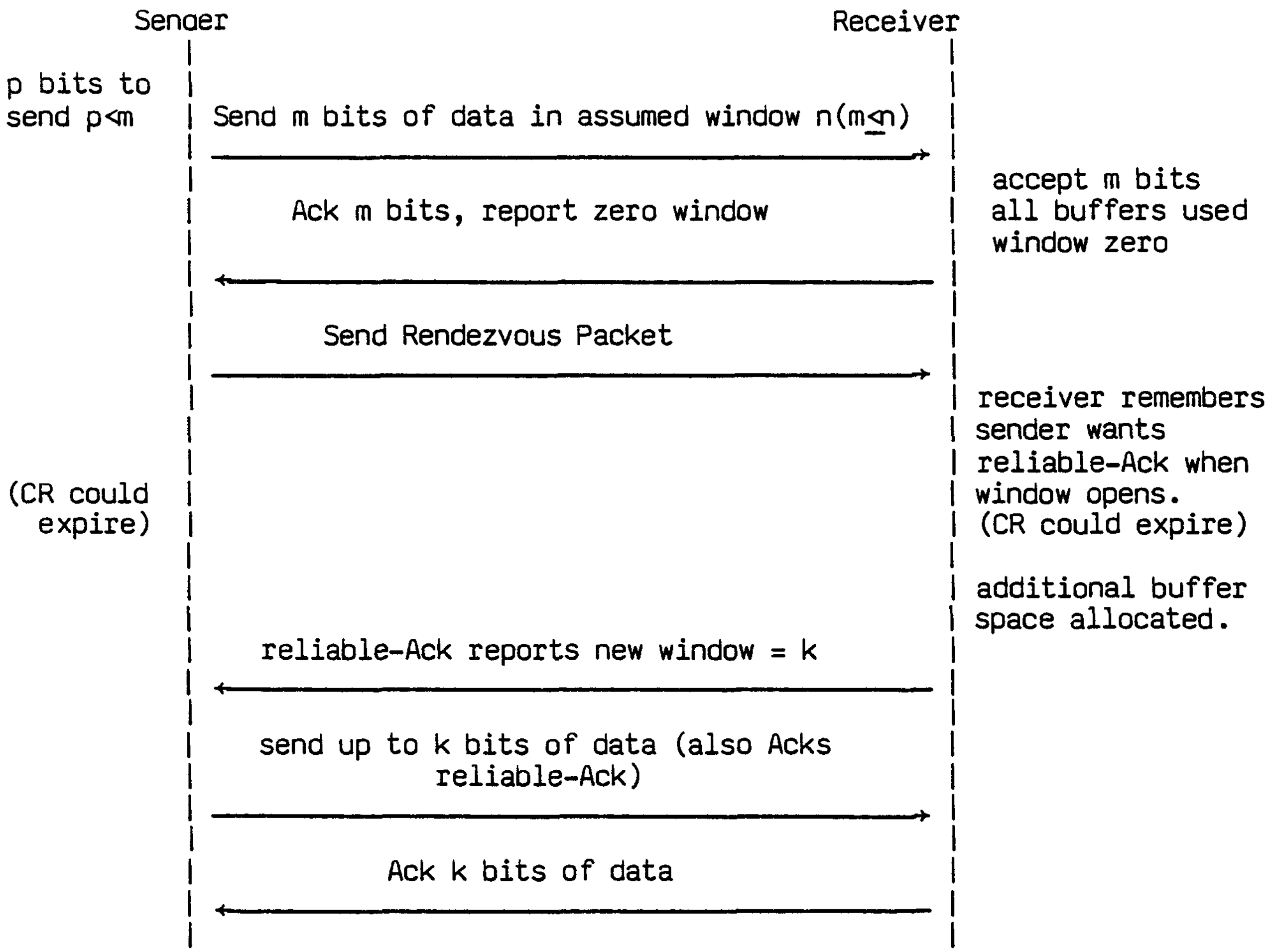


Figure 2.2. Rendezvous-at-sender Packet Exchange without Overflow.

Window overrun can occur because (1) the receiver reneged on an advertised window due, for example, to buffer withdrawal or because it was advertising windows based on a statistical buffer management strategy, or (2) the sender sent more than the advertised window. The sender might be able to detect window overflow if it receives an Ack with (ASN + window) less than the highest SN sent. Then it could stop transmitting new and retransmitting data outside the window. If the sender just blindly kept transmitting and retransmitting before the receiver allocated buffer space, there would be the possibility both of unnecessary traffic and that the giveup interval on some data might expire. If unAcked bits were outstanding at giveup time (because they were simply discarded by the receiver due to window overflow) then an unnecessary ambiguity exists for the user when the giveup is reported. The sending user does not know whether or not these bits were delivered, when in fact the receiving protocol module knew they were not. The user may then unnecessarily enter an expensive higher level error recovery procedure to resolve the ambiguity. Because the input window opening delay could be much longer than Δt if the window advertised is based on user space and the user is subject to long scheduling delays, unnecessary ambiguous situations could be frequent.

One Delta-t design goal, as stated earlier, has been to minimize the number of these ambiguous situations to those outside its control (network partitions and end node crashes). To deal reliably with the above problem two approaches are possible: (1) to make it illegal to renege or overrrun an advertised window (common in many protocols) or (2) to provide mechanism to reliably allow renege or overrun. Delta-t supports the latter. If overrun actually occurs, the receiver explicitly reports this fact in an Ack packet with a window-overflow-flag (Pwof) set. The outstanding unAcked overflow bits are then logically treated by the sender as if they were never sent, in effect extending their lifetime. Extending the lifetime of an overflow bit does not introduce any duplication hazard in a timer-based protocol if the rendezvous-at-the-sender procedure described above is used. It could introduce a hazard if polling were used, but the mechanism below would remove it also.

Duplicates of the overflow bits can cause a hazard, however, with the rendezvous-at-sender procedure described above if they are accepted by the receiver just after the window opens because they will "Ack" any reliable-Ack that may have been sent (stopping retransmission) and if that reliable-Ack and the Ack of the duplicate data just accepted both are lost, the sender will never learn the window has opened. To avoid this problem, Delta-t uses the following mechanism illustrated in Figure 2.3 to assure that duplicates of overflow bits are unacceptable.

When the receiver detects overflow it generates an Ack packet indicating overflow and a zero window and enters a state where it will not accept further Data packets until it receives an acceptable Rendezvous packet. When the sender receives an Ack indicating overflow has occured it (1) resets the state of the overflow bits as if they were never sent and (2) generates a Rendezvous packet (since all data sent has now been Acked) that contains the ASN in the Ack indicating overflow (so the receiver will accept it) and an SN offset to be added to it that will yield an SN larger than any overflow SN sent. The receiver then translates its input window SNs by the offset and reenters the Data packet acceptance state.

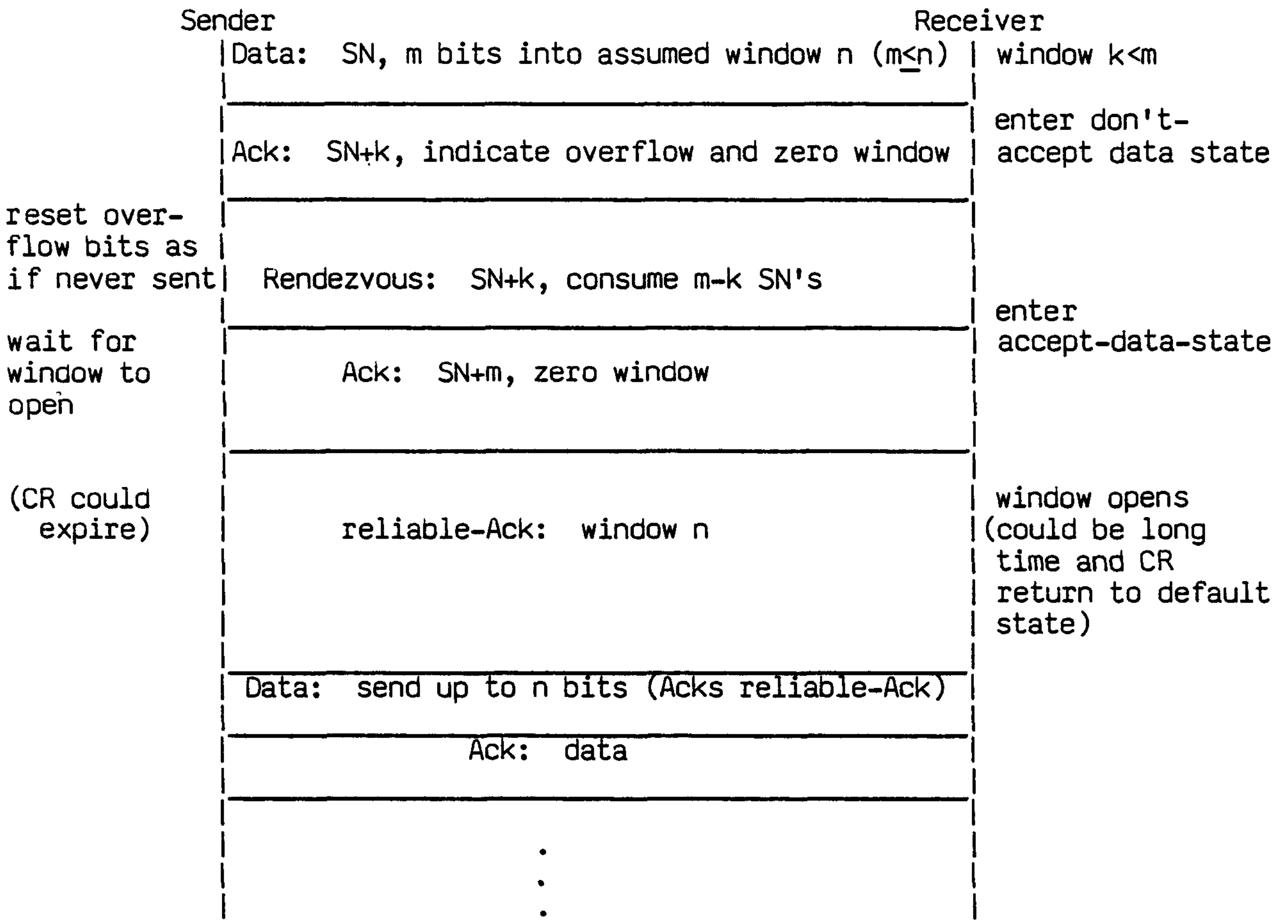


Figure 2.3. Rendezvous-at-sender with Overflow.

The question yet remains of what strategy the receiver should use in deciding what size input window to advertise. This is a very implementation dependent issue. Some suggestions are given in Appendix B.

2.8. Diagnostics and Measurement

The only diagnostic and measurement service offered by this version of Delta-t is the generation of Nak packets when a packet's lifetime has expired and optionally when out-of-sequence packets are rejected. Trace and timestamp routing services are offered by DeltaGram.

2.9 Services Not in Delta-t

Many transport protocols support two channels per association, a normal data channel and a second channel called variously an expedited, out-of-band, or interrupt channel. When a need for the latter type of channel is required by a LINCS application, a separate association is used.

Delta-t requires no Reset, Purge, or Clear type packets, nor are user interface primitives required to assist Delta-t in management of its connection records.

No priority field for data is provided.

The above features are not required because of the advantages of timer based connection management [16].

2.10 Future Services

The Delta-t bit stream is labeled with a Delta-t version number to provide for future evolution.

3. Services Required of the Network Layer

Delta-t as defined here is assumed to operate on top of a Network layer providing the services below.

3.1 Data Objects and Addressing

Delta-t assumes that the Network layer provides a full duplex uninterpreted data channel between two ports, each identified by 64 bit addresses.

3.2 Protection

Delta-t labels bits with a protection level passed on to the Network layer where a routing level protection policy is assumed enforced [5,19].

3.3 Assurance

Delta-t assumes that the Network layer is detecting and discarding damaged packets with a mechanism leaving no gaps in the protection.

Delta-t assumes that packet lifetime is bounded, that it can specify this bound, and that the receiving Delta-t end can obtain the assumed initial bound set by the sending end.

3.4 Resource Management

Flow Control
Network layer flow control service is not required.

Segmentation
Delta-t assumes that the Network protocol will segment packets containing Delta-t SNs used as packet identifiers and maintain correct bit labeling.

3.5 Synchronization

The only synchronization service required is to know where Delta-t packets begin and end and the ability to support the carrying of Delta-t B and E bit labels.

3.6. Control Information

Certain control information used by Delta-t such as initial packet lifetime, may also be used by the Network Layer. It is assumed that this information can be conveyed in either direction across the interface.

4. Model of the Delta-t Environment

4.1 Environment Model

Delta-t supports the LINCS-IPC or related services [21]. All IPC services are on an association basis.

Figure 4 illustrates the flow of information between remote user processes on an association. If the communicating processes were local then layers 1-4a would be replaced with a local transport mechanism.

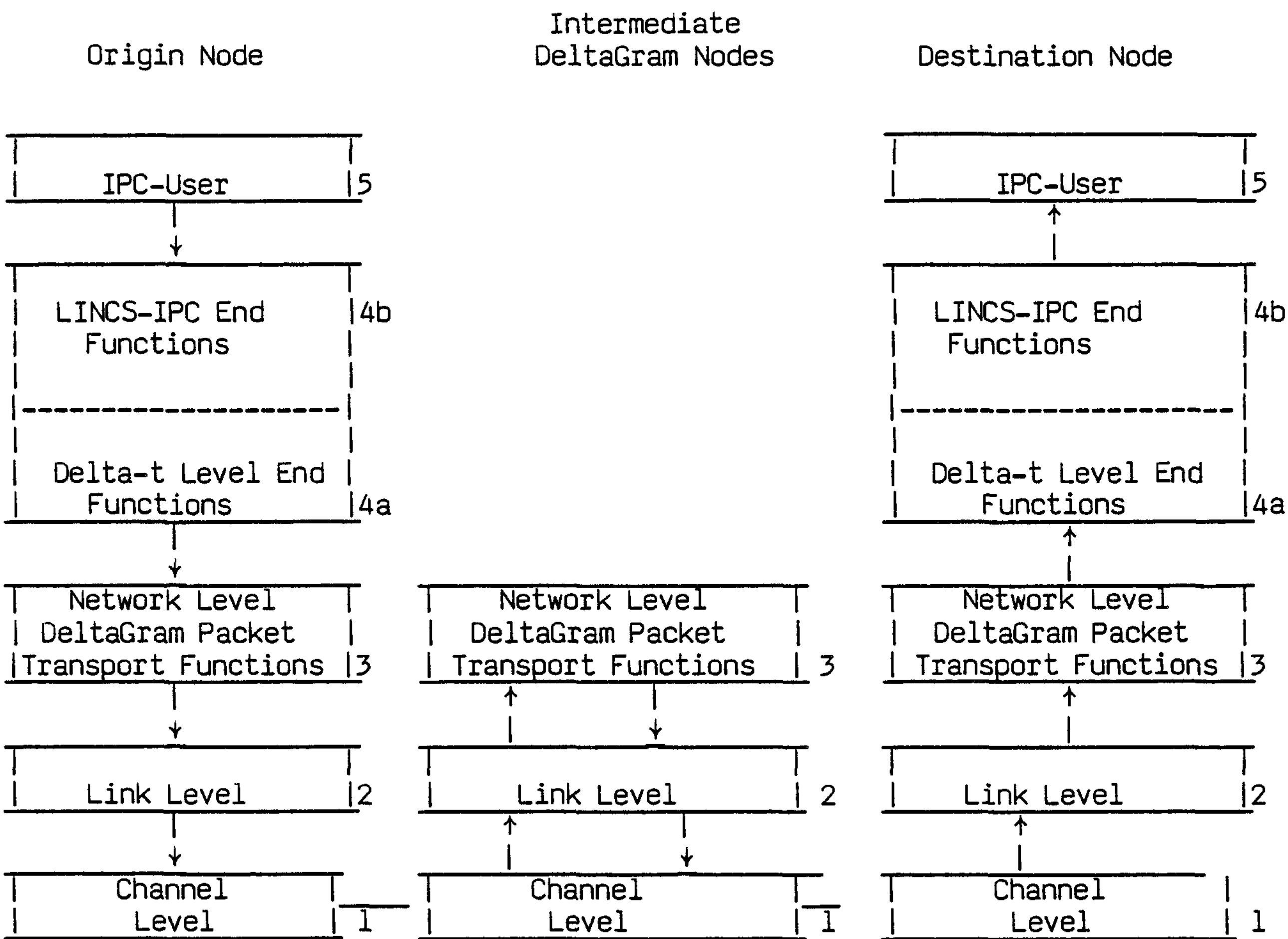


Figure 4-1. Information Flow Between Remote User Processes

A specification requires a model of the environment in which the protocol is to operate and of the structure of the protocol module itself. This specification is based on a programming language procedure model. The procedures embody the desired response to external events (next higher level interface, timer, next lower level interface packet receipt) in terms of state transitions (changes to state variables) and output events (packets or signals generated and timers set). The programming language notation used is Pascal [22] with an exponentiation operator (**). Pascal was chosen because it is widely read and has most of the notation needed.

For the purposes of this specification we view the Delta-t environment as logically consisting of three asynchronously running processes which we call:

(1) User,
(2) IPC (embodying Delta-t) and
(3) Link.

The Network (DeltaGram) level is embodied as procedures in both the IPC and Link processes. We use the term process simply to indicate a locus of concurrent activity. The three processes could be quite different kinds of entities in a given environment. In many implementations these entities might just be sets of co-routines within the same module. In other environments there might be many User processes, several Link processes, and a single IPC process multiplexing many associations for one or more of these Users. These are implementation details outside this specification. Unambiguous behavior can be specified in terms of single User, IPC, and Link Protocol processes.

The three processes communicate via shared data structures and wakeup signals, the latter undefined here. The data structure shared between the User and IPC processes is the Interface-State-Record (ISR) defined in Appendix B. The ISR consists of logical Send and Receive queues containing data to be sent or empty buffers for receiving data and other state. The buffers could be in User space or in system space. The data structure shared between the IPC and Link protocol processes consists of packet queues, and a Routing Table. The organization of the processes is shown in Figure 4.2.

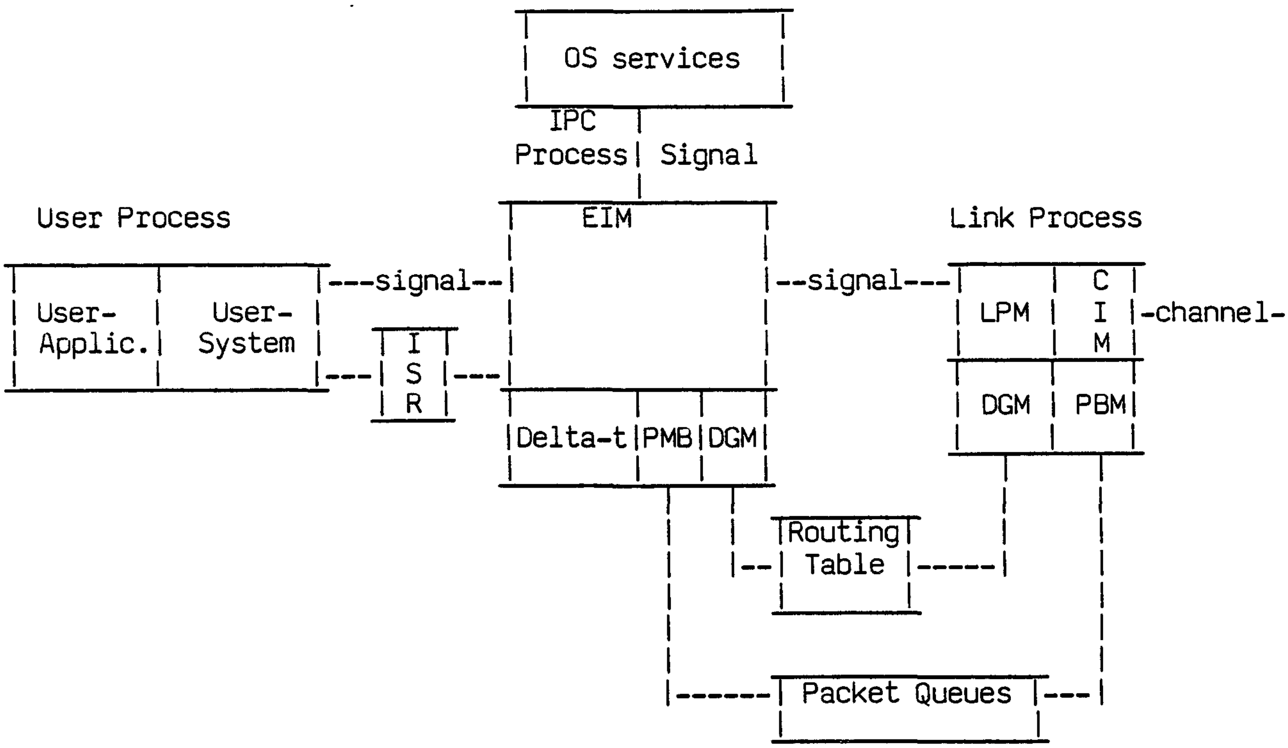


Figure 4.2 Model of a Delta-t Environment

Logically the User process consists of two sets of procedures, the User-application procedures, and the User-system procedures. The User-system procedures implement the LINCS-IPC interface primitives (see Appendix B and reference [21]). These procedures could be library routines or usually, because of protection, efficiency, and system integrity reasons are service procedures accessed via operating system service calls or the equivalent. The User-system procedures update the ISR and signal the IPC process.

Logically, the Link process consists of four sets of procedures and three levels of protocol:

- o The channel interface module (CIM) that interfaces to a lower level channel protocol,
- o The Link Protocol Module (LPM) that implements the Link protocol proper,
- o The DeltaGram Module (DGM) that implements the DeltaGram (Network level) service, and
- o The Packet Buffer Management module (PBM) that manages a pool of packet buffers and the Packet Queues.

The latter two sets of procedures are shared with the IPC process, which contains two levels of protocol, DeltaGram and Delta-t. DeltaGram service has not been separated as a separate process because packet format knowledge is needed within the Link process so that packet lifetimes can be updated to reflect time spent on packet queues (see reference 19 for the DgAdjustLifetime procedure). The Link process signals the IPC process when it places a packet on its packet Queue.

The IPC process consists of two other sets of procedures, besides the DGM and PBM:

- o The environment interface module (EIM) that isolates Delta-t from the details of a specific LINCS-IPC user interface, buffer management, synchronization mechanism, operating system, and lower level protocol environment.
- o The Delta-t module providing end-to-end services between remote IPC users.

The IPC process determines if the transfer is local or remote and utilizes the appropriate data movement mechanisms in each case. The discussion in this specification assumes network communication, but the LINCS-IPC service is supported between processes on the same (local) system as well as between processes on different (remote) systems. Local and remote communication probably use separate data transfer mechanisms for efficiency. The IPC process signals the Link process when packets need sending, and it signals the User process when Sends or Receives complete.

4.2 Event Handling

There are three sets of asynchronous events that affect Delta-t operation (1) IPC user interface (Sends, Receives, Aborts), (2) Timer, and (3) Receipt of packets. Choice of mechanism for synchronizing or a strategy for scheduling these events is very much dependent on the operating system design. Therefore, we assume that the EIM receives event signals, determines their type, schedules their handling and calls Delta-t with the appropriate primitive as needed. The EIM to Delta-t interface consists of five procedures defined in Section 6. Calls to these procedures represent events, their execution performs appropriate state transitions and output functions, and their returns represent output. Their correspondence with events is as follows:

User Interface Events:

Procedures DtStartData and DtFinishData report IPC user interface data sending events or the requirement to Ack a reliable-Ack.

Procedure DtAck reports IPC user interface buffer allocation events, or the need to Ack a received Data or Rendezvous packet.

Timer:

Procedure DtTimeout reports the expiration of a Delta-t timer.

Packet Receipt:

Procedure DtPktRcvd reports the receipt of a packet.

The reporting of expiration of Delta-t's Rtimer and packet receipt have time dependency. If there is too long a delay between the occurrence of Rtimer expiration or packet receipt and notification of Delta-t, some data sent by the other end may be unnecessarily rejected.

The parameters of these procedures define the association, offsets for controlling the logical queue pointers of the ISR (see Appendix B), receiving and sending control flags, and pointers to packet buffers (passed in both directions). The only assumption made here on logical packet buffer structure is that the first buffer in any structure is large enought to contain a DeltaGram header. Any remaining structure (e.g., creation of packet buffers from chained buffers) is known only to the Packet Buffer Management and EIM procedures.

There are two main buffer strategy issues: (1) whether the EIM should buffer data in its own buffers or maintain a pointer structure to buffers directly within user-application space and (2) what structure of buffers are logically supported: circular or block (square), fixed or variable length, etc. The EIM isolates the Delta-t primitives from which choices are made. We also want to isolate the details of how Receive-anys and Receive-specifics interrelate (see Appendix B), and the details of EIM implementation generally.

Besides a procedure to obtain a packet buffer (defined in Section 6), Delta-t needs two timer procedures supplied by the EIM, one to obtain the current dateTime and the other to set or cancel an alarm.

```
    function EIMtime (
        {Arguments - none}
        {Results}
            datetime: DateTime);
        begin
            {returns dateTime as an integer in appropriate units relative
            to some start point}
        end {EIMtime}.

procedure EIMalarm (
    {Arguments}
        assoc:AR; {association for which the timer is being set.}
        cdt: DateTime; {dateTime when a DtTimeout call should be
                issued}
        rcFlg, {request (true)/cancel (false) flag indicating whether an
                alarm should be set or canceled}
        presenceFlg:Boolean; {This flag is valid only if rcFlg is true and
                is returned to the EIM by the alarm server and indicates that
                the ISR should be in memory before calling DtTimeout as the
                ISR may need updating or be used as indicted in return
                parameters.}
        {Results: none;});
    begin
        {update the alarm server's database}.
    end {EIMalarm}.
```

5. Delta-t Use of DeltaGram Packet Header Fields

The Delta-t protocol, as specified here, is assumed to use the DeltaGram protocol [19]. Delta-t does not need explicit packet header space of its own because it can utilize services provided by DeltaGram.

Graphically a DeltaGram packet has the following format when laid out in 32 bit blocks.

0-1	2-3	4-6	7	8-15	16-19	20-23	24-31	
Pver	Ptype	Presl	Pdn	PhdrChksum	PprtctLev	PΔtexp	Plifetime	0
Pid								1
PdestAddr								2
PdestAddr - continued								3
PoriginAddr								4
PoriginAddr - continued								5
Ptdf - (packet type dependent field)								6
Ptdf - continued								7
Ptdf - continued Data packets only - contains user data								variable

The meaning of the fields is the following.

Pver: 2 bit DeltaGram version number (see DeltaGram specification for usage).

Ptype: 2 bit packet type.
- 0 Data packet.
- 1 Reverse Control (Delta-t Ack).
- 2 Direct Control (Delta-t Rendezvous).
- 3 Nak.

Presl: 3 bits reserved.

Pdn: 1 bit, do not Nak if undeliverable flag.

PhdrChksum: 8 bit header checksum - see DeltaGram specification for algorithm.

PprtctLev: 4 bit protection level.

PΔtexp: 4 bits for determining tick size used to decrement Plifetime and to determine initial Plifetime. tick = $\frac{2^{**}P\Delta texp}{256}$ seconds.

Plifetime: 8 bits remaining packet lifetime, in tick units.

Pid: 32 bit packet identifier (Delta-t SN).

PdestAddr: 64 bit destination port identifer.

PoriginAddr: 64 bit origin port identifer.

Ptdf: 64 bit packet-type-dependent-field defined below (may exceed 64 bits for Data packets).

A packet interface between Delta-t and DeltaGram is assumed here; that is, Delta-t makes up a complete DeltaGram packet header and receives a complete packet. The EIM places or removes data from a packet. How Delta-t utilizes or sets each header field for the four DeltaGram packet types is now defined. For all the packet types the following field settings apply.

Pver: Set to appropriate DeltaGram version.

Presl: Set 0.

PhdrChksum: Calculated and set as appropriate.

PΔtexp: Set from global association or connection record state, as required by packet type.

Plifetime: Set as appropriate to Delta-t operation.

PdestAddr: Set to the appropriate destination address.

PoriginAddr: Set to the appropriate origin address.

The remaining header fields are set dependent on packet type.

5.2 Data Packets

Ptype: Set to 0, Data.

Pdn: Set 0, Nak if undeliverable.

PprtctLev: Set to the protection level of the data contained.

Pid: Set to the SN of the first bit in the packet or that of the next bit to be sent if no data is contained.

Ptdf: The Ptdf field for a DeltaGram Data packet has the following format. The formats of the DeltaGram Pfbl, Plbl, and Pabl fields can be defined by Delta-t for its bit labeling use.

```
 0                                    15 16            23 24          31
 _____________________________________________________________________
|                                    |      Pfbl       |     Plbl     |
|               PdataChksum          |      |          |              |   6
|           2     4           11  12 |Pres2 |Pb| Pdrf  |Pres3    | Pe |
|____________________________________|_________________|______________|
|    |   |     |   |              |                                   |
|Pds |Pt |     |   |  Pabl        |          Pdl                      |   7
|    |   |_____|___|______________|                                   |
|    |   |        Pres4 |         |                                   |
|    |   |              |PΔtver   |                                   |
|_____________________________________________________________________|
|                                                                     |
|                            PuserData                                |variable
|_____________________________________________________________________|
```

PdataChksum: Set to checksum of PuserData (see DeltaGram specification for algorithm).

Pfbl:

Pres2: 6 bits reserved, set 0.

Pb: The B mark, set as appropriate for labeling the first data bit in the packet.

Pdrf: The data-run-flag, labels first bit.

1 All previously sent bits have been Acked,

0 There are outstanding unAcked bits.

Plbl

Pres3: 7 bits reserved, set 0.

Pe: The E mark, set as appropriate for labeling the last data bit in the packet.

Pds: Set 0, can segment if necessary.

Pt: Set 0, no trace or timestamp diagnostics.

Pabl

Pres4: 8 bits (6 bits in Pabl and 2 additional) reserved, set 0.

PΔtver: 2 bit Delta-t version number. The four versions have similar meaning as for DeltaGram, although the version numbers may be different.

Pdl: Set to the number of bits in the PuserData field.

PuserData: Variable, 0 or more data bits.

5.3 Ack Packets (Reverse Control)

DeltaGram Reverse Control packets are used for Delta-t Acknowledgment.

Ptype: Set to 1, Reverse-Control.

Pdn: Set 1, Do not Nak if undeliverable.

PprtctLev: Dependent on protection policy enforced.

Pid: Set to the Ack sequence number, the SN of the next expected bit (the receiver's left-window-edge).

Ptdf

```
 0                                          28 29          31
 _____________________________________________________________
|                                           |                 |
|        Pres5                              | Ack Flags       |
|                                           | P    P    P     |
|                                           | p  | w  | r     |
|                                           | u  | o  | a     |
|        10       11 12                     | f  | f  | f     |
|_____________________________________________________________|
|         |          |                                        |
| Pres6   |PΔtver    |      Pwindow                           |
|_________|__________|________________________________________|
```

Pres5 29 bits reserved, set 0.

Ppuf, Pid undefined flag.
- 1 if Pid undefined. Pid is only defined when Rtimer >0. This bit is set 1 when only a relative window is being reported. A Delta-t Ack packet can be used to just report an input window and not Ack any data.
- 0 if Pid defined, possibly Acking an SN.

Pwof: Window overflow flag.
- 1 if overflow,
- 0 if no overflow.

Praf: Reliable Ack flag.
- 1 if this Ack will be retransmitted until its sender receives an acceptable Data packet.
- 0 if normal Ack (sent one time only).

Pres6: 10 bits reserved, set 0.

PΔtver: 2 bit Delta-t version number.

Pwindow: 20 bit flow control window.

5.4 Direct Control Packets

The direct control packet is used by Delta-t to convey various control information. So far only one control subtype has been defined. It is for use in window management (see Section 2.7.3). The general format of the Ptdf field of this type packet is the following.

```
0            7                     31
|  Psubtype  | Subtype dependent (Std)|
|---------------------------------------|
|       Std       | PΔtver  |   Std     |
```

Psubtype: defines the control subtype.

PΔtver: 2 bit Delta-t version number.

Subtype dependent: defined for each subtype.

Psubtype = 1: Rendezvous packet. Packet header fields for a Rendezvous packet are the following.

Ptype: Set 2, Direct-control.

Pdn: Set 0, do not Nak.

PprtctLev: Depends on protection policy enforced.

Pid: Sequence number of next data bit receiver is currently known to expect.
Ptdf

```
0        7 8                       23  24         31
|    1    |        Pres7           | P |   Pres8    |
|         |                        | d |            |
|         |                        | r |            |
|         |       10       11 12   | f |            |
|---------------------------------------------------|
| Pres8 continued|PΔtver  |          Psno           |
```

Pres7: 15 bits reserved, set 0.

Pdrf: (see Pdrf for Data packets).

Pres8: 18 bits reserved, set 0.

PΔtver: 2 bit Delta-t version number.

Psno: SN offset used by receiver to readjust its next expected SN.

5.5 Nak Packets

0	31
PnakReason	PnakRes 11 12
PnakRes	Pdl

No special Delta-t format.

Ptype: Set 3, Nak.

Pdn: Set 1, do not Nak if undeliverable.

PprtctLev: Depends on protection policy enforced.

Pid: That of packet being Nacked.

Ptdf:

The "reason for Nak" code space from 128 to 255 is reserved for the next higher level. For Delta-t:

128 = arbitrary refusal.

129 = out-of-sequence.

6. Delta-t Operation

6.1 Introduction

This section specifies Delta-t service and operation in terms of a Pascal based procedure model for an association specified as an argument to the Delta-t interface procedures. Sections 6.1 and 6.2 in conjunction with preceding sections should be sufficient to give the reader an overview of Delta-t operation. Sections 6.3 to 6.7 present the model in detail. The model is not intended to imply a required implementation. It is intended to unambiguously specify functionality. Any algorithm with equivalent functionality can be used.

The model presented here assumes operation within the environment (EIM) defined in Section 4. Until two or more communicating implementations exist, this specification should be assumed to contain bugs. Please contact the author if questions arise.

The Delta-t model is a finite state machine. A Delta-t input event is represented by a procedure call. Input events are scheduled within the EIM according to implementation dependent resource management priorities. State is represented in a Connection Record (CR) defined in Section 6.2. Variables beginning with capital R or S are CR receive and send variables respectively. The procedures of the model embody the correct state transition rules. Output events are represented either as parameters returned in the Delta-t interface procedures (packets to be sent and updates to EIM state) or procedure calls issued by Delta-t to set or cancel timers. Before calling Delta-t the EIM obtains a buffer large enough to hold a packet header.

The three classes of input events and their effect are now outlined.

Timer Events: Timers are set by Delta-t calling the procedure EIMalarm. When a timer expires, the EIM issues a DtTimeout call. DtTimeout determines which,if any, of the following three events has occurred, performs state update, and generates required output.

Rtimer → 0.

Rtimer is the only timer that could cause potential problems if it ran longer than it was set for. In this case packets might be rejected that could be accepted, leading to possible unnecessary retransmissions and ambiguity if acceptance did not occur. Therefore, this event should have high priority for input to Delta-t.

- o All CR receive state variables are reset to or become default values.
- o There is no output function.

Stimer → 0.

This event is handled in the procedure StimerExpired.

- o All CR state send variables are reset to or become default values.
- o If all packets sent have not been Acked, a giveup timeout has occurred. Data in doubt is identified and an error code is output.

Retrytimer → 0.

A packet's retry timer has expired (checked in function shouldRetry).

- o Packet retransmission is handled in the procedure sendRetry. If the lifetime of the packet to be retransmitted has not expired, parameters are returned as output. Data retransmission takes place by Delta-t indicating to the EIM the data to be retransmitted in the return from DtTimeout. The EIM then recalls the Delta-t procedures DtStartData and DtFinishData to prepare the Data packet for retransmission. Delta-t prepares Ack or Rendezvous packets to be retransmitted and returns a pointer to the packet buffer.

- o A Retrytimer may be set as an output function.
- o The state of the retry data structure is updated.

After having checked for the above events, if any, and having performed the appropriate state transitions an EIM data sending condition (see function tryData) is checked and a return variable is set. (If an initialization wait interval has expired, packet sending can proceed. If a Data packet has exceeded its maximum retransmission interval new Data packet sending is blocked.)

The CR is then checked to see if it's in its default state (Rtimer and Stimer both expired). The CR can be deallocated if it is in its default state.

Data Receiving and Sending: Packet formation and state update takes place in procedures with names of the form sendX ,where X is Data, Ack, Rendezvous, or Nak.

Receive or Receive Abort

When a Receive or Receive-Abort call is issued by the IPC user, or Delta-t has indicated in a return from DtPktRcvd that an Ack is required, the EIM updates its state and issues a DtAck call to Delta-t.

- o The receive window (Riwre) is updated.
- o The Stimer and retry data structure are updated if a reliable-Ack is generated because a zero receive window is opening.
- o An Ack packet is output and send state is updated.
- o The timestamp (Stimestamp) used to initialize the Ack's lifetime is reset.

Send

When a Send call is issued by the IPC user, the EIM updates its state and issues a DtStartData call when its state indicates Delta-t may be able to send a Data packet. If data can be sent this call is followed by a DtFinishData call (to compute header and data checksums). The EIM will also issue a DtStartData call even if no data is available if Delta-t has indicated in a return from a DtPktRcvd call that a Data packet is required to Ack a reliable-Ack.

- o Delta-t checks a Data packet sending condition (see function shouldData) to see if a Data packet can or should be sent. If a Data packet cannot be sent a Rendezvous packet sending condition is checked (see function shouldRendezvous) to see if a Rendezvous packet should be sent. If a Data packet is sent, the packet header is prepared by Delta-t and a pointer to a packet buffer and a count of the amount of data to be sent are returned in DtStartData. The data is then placed in the packet buffer by the EIM and DtFinishData is called. If a Rendezvous packet is to be sent, Delta-t prepares it and returns a pointer to it.
- o The Stimer will be set when a new data or Rendezvous packet is sent.
- o Send state variables reflecting the number of SNs consumed by data or Rendezvous packet are updated. The output window is reset to zero if an E-bit is sent.

o An EIM data sending condition (see tryData) is checked to determine if data sending can continue and a return parameter is set.

An abort of a Send by the IPC user only affects state in the EIM and is not an event of interest to Delta-t.

Packet Received from the Next Lower Level:

When a packet is received the EIM issues a DtPktRcvd call passing Delta-t a pointer to the packet.

o Delta-t tests each packet received for acceptability. The rules for packet acceptance are contained in procedures or functions with names of the form acceptX, where X is as defined above. Packets are discarded if unacceptable. A Nak packet is returned for two cases of unacceptable Data packets (Lifetime expired, or optionally if out-of-sequence packets are rejected). If the received packet is a Rendezvous (packet accepted or not) or Data (when accepted or rejected and when a Nak is not sent) packet an Ack flag is set in the return from DtPktRcvd. The EIM will schedule a DtAck call which will generate an Ack with the latest receive window. The lifetime of the Ack packet begins at the point a packet requiring an Ack is tested for acceptance. Accepted packets are processed in procedures with names of the form processX, where X is as defined above. The handling of accepted packets is now outlined.

Data Packets:

- If data is accepted or overflows, Rtimer is set. In the latter case the variable RovflwInd is also set. The input-window-left-edge (Riwle) is adjusted for data accepted. The retry data structure is updated if a reliable-Ack is Acked by this Data packet.
- Delta-t returns an offset within the received packet and count of the amount of data to accept, its protection level, and whether or not the first and last bits accepted are respectively labeled with a B or E mark.
- An Ack flag is returned. An Ack packet will be generated when the EIM schedules and issues a DtAck call.
- The IPC user is signaled by the EIM if a Receive that it issued completes.

Rendezvous Packets:

- Rtimer is set.
- The acceptance window (Riwle, Riwre) is adjusted.
- If the receive window is zero a return parameter is output to the EIM indicating that it should remember that the correspondent end wants to be reliably informed when the receive window opens.
- An Ack flag is returned. An Ack packet will be generated when the EIM schedules and issues a DtAck call.

Ack Packets:

- Delta-t send state is updated (what data has been Acked, the output window, whether or not waiting for the output window to open, and Nak state).

- o State parameters required by the EIM are returned (what data have been Acked, the new output window, and if overflow has occurred and, what data has overflowed and needs resending).
- o A code is returned to the EIM indicating that a DtStartData call is required to cause a Data or Rendezvous packet to be generated to Ack a reliable-Ack even if there is no data available for sending, that EIM sending can proceed when data is available or that sending is blocked.
- o A data sending condition is checked to determine if EIM sending can proceed (see function tryData) and the EIM is informed of the result.
- o The IPC user is signaled by the EIM if a Send has completed.

Nak Packets:

- o State is updated possibly resulting in optional suspension of new data sending, or in immediate retransmission.
- o The Nak is optionally recorded in a history file.

For reference in the sections below the meaning of the first letter of variable names is the following:

A - association variable.
P - field of a packet Record, defined in Section 5.
R - field of a Connection Record, primarily affecting receiving, defined in the next section.
S - field of a Connection Record, primarily affecting sending, defined in the next section.
s - local send variable.
r - local receive variable.
any other letter - local variable used only in the procedure it is declared or argument or return.

6.2 Connection Record Definition and Management

6.2.1 Introduction

Delta-t operates on control information carried in Delta-t packet headers and state information maintained by each end. Delta-t packet header information was defined in Section 5. We now define the state information maintained at each end. Logically, state information is always being maintained by each end for all possible associations with which Delta-t might be involved (permanent connections). In fact, however, state information must only be explicitly maintained for a subset of associations. For all other associations, the state information values are standard defaults. State records containing default values can be reclaimed.

The nondefault state information required by Delta-t is maintained under timer control. While nondefault state information (either Rtimer or Stimer nonzero) is being maintained for an association an active connection is said to exist. This state information is maintained in a connection-record (CR).

The variables collected together in the CR exist on a per association basis. They are the variables that must be maintained across calls to the Delta-t. Some of these state variables are required in any model or implementation, others are dependent on the details of the model or implementation. Other send and receive variables are local to Delta-t procedures.

a) Receive State

b) Send State

Figure 6.1 Receive and Send CR State Information per Association (not the comp lete CR)

The CR Receive and Send state information is shown in Figure 6.1 In addition other CR parameters for an association are required. These are prefixed with the letter A, and are defined in Section 6.2.5.

An important aspect in the design of any assurance and flow control protocol is the synchronization and evolution of the state information in connection records in the face of arbitrary transmission delays, errors, and end-node crashes and deadstarts. This process is called connection management. Connection records in Delta-t are managed, invisible to the next higher level, based on two timers at each end of an association, the Stimer for sending and the Rtimer for receiving. CR's exist when either Rtimer or Stimer is nonzero. Each timer interval provides assurance and smooth data flow services (see Appendix A). The rules for timer managment were outlined in Section 2.6.

During normal but bursty data flow, with bits being Acked in a timely manner, active CRs come into play and may later be reclaimed with no interactions required with the next higher level. When Stimer = 0 while unAcked SNs are outstanding, the EIM, and in turn the IPC user, is informed that an error exists and what data if any have been sent but not yet Acked. The IPC user must then decide how to continue. This situation can only happen if the network is partitioned or the receiver crashes.

Half open connections are handled by a wait interval after initialization, discussed below.

6.2.2 Sender Initialization

Deadstart or crash recovery requires that all state records (or just those for damaged associations) be reset to their default values. An interval 3Δt must expire on a damaged association (crash with loss of memory) before sending any type of packet (see Appendix A). No Ack or Nak packet should be accepted before data has been sent. This assures that the destination's Rtimer will time out (removing half open connections) and that all data packets sent before the crash and their Acks or Naks have been destroyed. This condition is enforced in the model by checking this interval during the function tryData and the procedures processAck and processNak.

(There is the implied requirement that senders must know what value of PΔtexp they were using before a crash, modeled here as an association constant AΔtexp (see Section 6.2.5).)

6.2.3 Receiver Initialization

Receivers must wait at least Δt after an initialization before accepting any Rendezvous or Data packets to protect against duplicates (see Appendix A). The Δt used is the sender's and, with loss of memory, the receiver will not know it until a packet arrives. Therefore, the receive wait interval is computed from PΔtexp in the packet header relative to the Aidt field in the CR (see Section 6.2.5). This condition is checked in the procedure acceptData and the function acceptRendezvous and guarantees that all packets sent before the crash will have been destroyed, before receiving begins again.

6.2.4 Connection Record Definition

The Connection Record defined below is not meant to imply that a given implementation would require exactly the same variables. More or less variables may be needed depending on its algorithms. All variables are initialized to default values when the CR is created.

```
CR = {Connection Record} record
   Aassoc:AR; {association record defined in Section 6.3}

   AmaxPktSize, {max packet size for this
      association, set from global state when the CR is created.}

   AΔtexp, {parameter set from global state to be used to compute the
   initial value of the packet Plifetime field, placed in the packet
   PΔtexp fields, and used to derive the value for Stimer.  A given
   implementation chooses AΔtexp to create an appropriate Δt.  Δt
   is the sum, Δt = R + MPL + A, where
      R= time sender normally expects to keep retransmiting (this time
         would usually be n average-round trip times).
```

MPL = an estimate of worst case acceptable network-travel-time. It should be a value assuming queuing and processing in the longest expected chain of intermediate store and forward nodes.

A= Maximum expected time until the receiver will Ack an SN. The value is a function of receiver's implementation or some reasonable worst case estimate such as a few seconds. A standard upper bound on A will be established.}

Aretrytime:integer; {time between retransmissions when "Acks" are not received; a number related to average round trip time set from global state.}

Aidt:DateTime; {The dateTime of the last initialization of the environment for this association.}

{Send variables set to default values when the CR is initialized}

Stimer,

{Purpose: Stimer serves two functions, assurance and smooth data flow. The assurance function of the Stimer is also twofold: (1) to assure that the CR is maintained until all Acks will be received if they are ever going to arrive (graceful close, only a remote end crash or network partition would prevent their timely arrival), (2) to assure that no SN is reused with new data until all packets containing it have died.

The smooth data flow function guarantees that the sender's CR is active longer than the peer's CR so that acceptable SNs are generated. No harm results if Stimer is allowed to run beyond its expiration time. Its purpose could be compromised if it is allowed to expire early.

When Stimer expires and Sou≠Sowle an error condition exists (see below).

Default: = 0.

When changed: Stimer is set when a new sequence number (SN) is sent in Data (see procedure sendData) or Rendezvous packets (see procedure sendRendezvous), or a reliable-Ack packet is sent requiring a Data packet as an "Ack" (see procedure sendAck). It is set to the dateTime it is to expire. The Stimer interval is 3*2**AΔtexp. Stimer is reset to 0 when it expires (see procedure StimerExpired).}

StimeStamp:DateTime;

{Purpose: StimeStamp is the dateTime of receipt of a Data or Rendezvous packet requiring an Ack packet. This is a model dependent variable required here because an Ack is not necessarily generated immediately when Data or Rendezvous packets are tested for acceptance. The EIM must schedule a DtAck call to cause Delta-t to update the receive window (Riwre-Riwle) and generate the Ack. If no delay were assumed between the return from a DtPktRcvd call and the issuing of the DtAck, this variable would not be needed. The requirement that must be met for correct Delta-t operation is that there must be no gap between the timing of the lifetimes of the latest

SN and its Ack. The condition to be met is that the combined lifetime of the latest SN received in a Data or Rendezvous packet and its Ack must not exceed 2*2**PΔtexp (2Δt) (see Appendix A). Exactly where a given implementation chooses to end the timing of the lifetime of a received SN and begin the lifetime timing of its Ack is an implementation choice. In this model StimeStamp is used to compute

the interval between acceptance testing of the most recently arrived SN and its Ack. The Rtimer (see receive state below) is to be refreshed at the point the lifetime timing of each incoming SN stops.
Default: = 0.
When changed: StimeStamp is set to the current dateTime during the procedures processData or processRendezvous and reset when an Ack packet is sent (see procedure sendAck)}

{Now we define a send SN space, a series of SNs that correspond in SN space to the pointers in the ISR logical send queue (see Appendix B).}

Sou,
{Purpose: SN of the oldest unAcked SN. If Sou = Sowle then all Data or Rendezvous packets sent have been Acked.
Default: = arbitrary.
When changed: Sou is updated during the procedure processAck as data or Rendezvous packets sent are Acked.

Sowle,
{Purpose: SN of the next bit or Rendezvous packet to be sent (output-window-left-edge).
Default: = Sou.
When changed: Sowle is changed in the procedures sendData and sendRendezvous when a Data or Rendezvous packet is created.

Sowre:SN;
{Purpose: SN + 1 of "largest SN" the receiver can accept (output-window-right-edge). That is, the receiver has advertised willingness to receive SN's up to but not including Sowre.
Sowre is used to determine if Data packets containing data can be sent (see function shouldData), or a Rendezvous packet should be sent (see function shouldRendezvous).
Default: = Sowle + n, where n is a network or association default.
When changed: Sowre is updated to Sou + Pwindow in the procedure processAck, to Sowle in the procedure sendData when a E-bit is sent (output window goes zero), and to Sowle plus an offset provided by the EIM in procedure DtStartData.

{all arithmetic and inequalities with SNs must be performed correctly modulo 2**32. The relationship Sou ≤ Sowle ≤ Sowre must always hold}

SrendSenderInd,
{Purpose: Indicates that a Rendezvous packet has been sent and the sender is waiting for its output window to open (Sowre > Sowle).

Default: <u>false</u>.
When changed: Set during the procedure sendRendezvous and reset during the procedure processAck, when the window opens.}

SovflwInd,
{Purpose: A model dependent flag recording that data sent have overflowed. This will result in a Rendezvous packet being sent when DtStartData is called.
Default: <u>false</u>.
When changed: SovflwInd is set in the procedure processAck when overflow occurs and is reset in sendRendezvous when the SN's of the overflow data have been skipped.}

SeSentInd,
{Purpose: A model dependent flag indicating that an E-bit has been sent, but has not yet been Acked. While SeSentInd is <u>true</u> the output window (Sowre-Sowle should remain zero.
Default: <u>false</u>.
When changed: SeSentInd is set in the procedure sendData when an E-bit is sent and is reset in the procedure processAck.}

SretryInd,
{Purpose: A model dependent variable that records that the next DtStartData is for retry data.
Default: <u>false</u>.
When changed: It is set during the procedure sendRetry. It is reset during the procedure sendData.

SseriousNakInd:Boolean;
{Purpose: Records that a Nak has been received indicating there is some problem serious enough to suspend sending new data packets (not required for correct operation, only for efficiency). Retrys should be continued for the normal cycle just in case the Nak was caused by a transient malfunction or ambiguous Nak exists (see Section 6.6.1).
Default: <u>false</u>.
When changed: During the procedure processNak, and reset during the procedures StimerExpired and processAck}

SnakReason: integer;
{Purpose: Location for keeping the latest PnakReason. This code is reported as a problem hint to the EIM if a giveup timeout error occurs. It is advisory information only.
Default: 0, means have not received any Nak reason.
When changed: Set during processNak and reset in procedures processAck and, StimerExpired (when the CR is reset to default values) (when all data or. packets sent have been Acked).}

SinPtr, SoutPtr, SendPtr = ↑RetryRecord (see below);

{Purpose:

These pointers point to RetryRecords in a Retry Queue. (How retry is handled is model or implementation dependent. A particular retry algorithm is included here for completeness of the model.) SinPtr is nil or points to the first Retry Record in the queue. SoutPtr is nil or points to the oldest RetryRecord in the queue with an active retry timer. SendPtr is nil or points to the end (last) record in the queue. The entries in the closed interval between SinPtr and SoutPtr will be retransmitted when their retry timers expire, if packet lifetime has not expired. The entries in the interval between SoutPtr but not including the entry at SoutPtr, and SendPtr including the entry at SendPtr have had their maximum number of retries and are waiting for acknowledgement.

The oldest entry that can be retried is at SoutPtr and the youngest will be added in front of the entry at SinPtr. The entries are thus ordered by age.

The condition SoutPtr ≠ SendPtr is important as it indicates SNs exist that have had their maximum retrys and no new data should be sent (see Appendix A).

```
RetryRecord = record
   rrtype:  (Data, Ack, Rendezvous); {type of
         packet}
   rrEntrytime, {time placed in queue}
   rrRetryTimer, {time when next retry can take
         place}
   rrLifetime: DateTime; {time when packet lifetime
         expires}
   rrPID: SN; {SN in packet Pid field}
   rrSNO: integer; {for Data packets this is Pdl, for
         Ack packets its Pwindow, for Rendezvous
         packets its Psno}
   rrBlink, {back link to previous entry}
   rrFlink = ↑ RetryRecord; {forward link to next
         entry}
end {RetryRecord}.
```

Default: SinPtr = SoutPtr = SendPtr = nil.

When changed: These pointers are manipulated during the various retry procedures (see Section 6.4.2), and are reset in the procedure StimerExpired when the CR is returned to its default state.}

{Receive related variables}

Rtimer: Datetime;
{Purpose: Rtimer provides assurance and smooth data flow services (see Appendix A). The assurance service of the Rtimer is to provide protection from duplicate packets. The smooth data flow service of the Rtimer is to guarantee that any packet sent with Pdrf = <u>false</u> that arrives at the receiver after a predecessor packet sent with Pdrf = <u>true</u>, will be acceptable. Pdrf is used for detecting missequenced packets [6].
Default: = 0.
When changed: Rtimer is set when a new SN is accepted (new data or Rendezvous packet), or there is a receive window overflow even if no data is accepted. When Rtimer = 0, then Data or Rendezvous packets will only be accepted that have Pdrf = <u>true</u>, any other packet is considered out-of-sequence. Such a packet may be held at the implementer's option but its lifetime must continue to count down until it is in sequence. While Rtimer > 0 packets are accepted when insequence with no regard to the value of Pdrf.

RΔtexp: integer;
{Purpose: This quantity is used to compute the value of the Rtimer interval, to compute the Plifetime field in Ack packets, is used as the PΔtexp field in Ack packets, and to determine if the receive initialization wait interval has expired.
Default: = undefined.
When changed: It is set during the procedures processRendezvous and processData when the first packet is accepted for a given CR. The value is initialized from the PΔtexp field in the received packet that caused Rtimer to be first set.}

{Receive SN space variables, logical receive queue SNs}
Riwle,
{Purpose: Next expected and acceptable SN (<u>i</u>nput-<u>w</u>indow-<u>l</u>eft-<u>e</u>dge). Used to protect against lost, duplicate, and missequenced packets. The procedures, as written in this specification, assume that packets are processed in sequence. Logically, we assume that out-of-sequence packets, if not discarded, are recognized and buffered until they can be processed in sequence. Their Plifetime fields must continue to count down.
Default: Undefined for assurance purposes, however, the interval Riwre-Riwle may be meaningful for flow control.
When changed: Riwle is adjusted during the procedures processData and processRendezvous, as SNs are accepted.}
Riwre:SN;
{Purpose: SN of the next bit beyond where there is currently available buffer space (<u>i</u>nput-<u>w</u>indow-<u>r</u>ight-<u>e</u>dge). That is, the receiver can accept SN's up to but not including Riwre. The interval between Riwle and Riwre defines the number of SNs that can be accepted and the value of Pwindow sent in Ack packets. This window is advisory only.

```
      Default:   undefined.
      When changed: Riwre is adjusted in procedures DtAck, processData,
                    processRendezvous.  It represents user interface
                    Receive events.}
   RovflwInd:Boolean;
     {Purpose:   A flag indicating that the receiver's buffers were overrun
                 and that Data packets should not be accepted until a
                 Rendezvous packet is accepted and Riwle has been adjusted
                 to protect against duplicates of the overflow bits.
      Default:   false.
      When changed: It is set during the procedure processData when
                    overflow occurs, and is reset during the procedure
                    processRendezvous and DtTimeout.}
   end {CR}.
```

6.2.5 Allocation and Deallocation of State

The CR is created and destroyed by the following procedures.

The procedure getCR returns the CR for a given association and, if necessary, creates one.

```
   procedure   getCR (assoc:AR; var crPtr:CRpointer).  {AR and CR are
               association and connection records}.
     begin
       {CRs are kept in an implementation dependent data structure where
       they can be retrieved efficiently by association.  If no CR exists
       for the association, one is created in the default state and is
       placed in the CR structure.  If there is no CR space available then
       crPtr returns nil and the Delta-t procedure will fail.  More
       sophistication is certainly possible but not modeled here.}
       if (EIMtime-Aidt) <3*2**AΔtexp then
         with crPtr↑ do EIMalarm (assoc, Aidt + 3*2**AΔtexp, true,
           true) {This will generate a DtTimeout call later and allow
           sending to proceed after an initialization wait interval}
     end {getCR}.
```

The procedure defaultCR checks whether or not the CR is in its default state. If it is, the CR is reclaimed. In the model, the procedure defaultCR cannot be reached while Ack, Nak, or Rendezvous packets should be sent or Data packets should be resent. Thus implicit in the CR default condition is the requirement that all packets needing sending for control purposes have been sent. New data never having been sent, but not sent because of a zero output window may, however, exist.

```
   procedure defaultCR (crPtr:CRpointer);
     begin
       with crPtr↑ do
         if (Stimer = 0) and (Rtimer = 0) then dispose (crPtr)
   end {defaultCR}.
```

6.3 The Delta-t Module Global Environment

The Delta-t procedures reside in the following declaration environment.

```
const
   Data = 0;
   Ack = 1;
   Dcntr: = 2 {Rendezvous};
   Nak: = 3;
type
   SN              =  0..2**32-1;
   PKT             =  record {Pascal record of the packet structure
                       defined in Section 5};
   Address         =  array [0..63] of Bit;
   CR             = {Connection Record} record {defined above};
   AR             = {association} record destAddr, originAddr:Address
                    end;
   dateTime    = integer;
   CRpointer = ↑CR;
   PKTpointer = ↑PKT;
      RetryPointer = ↑RetryRecord;

procedure getCr (Assoc:AR;var crPtr:CRpointer); {defined in Section
                 6.2.4}

procedure   setTimer (crPtr:CRpointer; timer,interval:DateTime;
                 presenceFlg:Boolean);
   {This procedure sets the timer in the CR pointed to by crPtr and
   calls the EIM alarm service to generate a signal when the timer
   expires.  The presenceFlg is an efficiency hint for the EIM; when true
   it indicates that on a timer expiration the ISR (see Appendix B)
   should be in memory before calling DtTimeout as the ISR may need
   updating.}
   begin
      EIMalarm (assoc, timer, false, presenceFlg); {cancels alarm for
               previous expiration of timer.}
      timer := EIMtime + interval; {time when timer is to expire.}
      EIMalarm (assoc, timer, true, presenceFlg) {sets alarm}
   end {setTimer};

procedure  DGadjustLifetime (timestamp:Datetime; offset:integer;
      ptr:PKTpointer↑PKT; var remainingLifetime:integer); {defined in
      DeltaGram specification [19].
   begin
      This primitive adjusts the lifetime of the packet pointed to by ptr
      and remainingLifetime returns a value ≤ 0 if the lifetime has
      expired else returns a value > 0.}
   end {DGadjustLifetime},

procedure  EIMtime {defined in Section 4});
procedure  EIMalarm ({defined in Section 4});
procedure  DtTimeout ({defined in Section 6.4});
procedure  DtAck ({defined in Section 6.5.1});
procedure  DtStartData ({defined in Section 6.5.2});
procedure  DtFinishData ({defined in Section 6.5.2});
procedure  DtPktRcvd ({defined in Section 6.6});
procedure  dataChecksum ({defined in Section 6.5.2});
```

```
procedure  headerChecksum ({defined in Section 6.5.2});
procedure  addRetryEntry ({defined in Section 6.4.2});
procedure  deleteAckedEntries ({defined in Section 6.4.2});
procedure  deleteRetryEntry ({defined in Section 6.4.2});
procedure sendAck ({defined in Section 6.5.1});
procedure  sendRendezvous ({defined in Section 6.5.2.1});
function  min (a1,a2,a3:integer):integer {returns minimum of 3
   arguments};
function  tryData ({defined in Section 6.5.2});
```

6.4 Timer Event Handling and Retransmission Procedures

6.4.1 DtTimeout

Timer events are reported to Delta-t by calling the procedure DtTimeout. This procedure represents Delta-t's rules for handling timer expiration. It checks whether or not Rtimer, Stimer, a retrytimer, and send initialization wait intervals have expired. It performs the appropriate state update and output actions. It checks to see if the CR is in a default state. It also determines whether or not EIM sending can proceed.

```
DtTimeout (
   {args}
   assoc:AR;       {association record for association with timer
                   expiration.}
      sPkt:PKTpointer {Packet header for possible Ack or Rendezvous
         packet needing retransmission.}
   {returns}
   var retryFlg,  {if true then the next DtStartData call should be for
                  count retry data bits starting at offset relative to
                  ouPtr (reason and sPkt are meaningless)}
   sPktFlg, {if true an Ack or Rendezvous packet needing retransmission
      was formed.}
   giveupFlg:Boolean; {if giveupFlg is true then a packet(s) (Data or
      Rendezvous) with offset bits relative to ouPtr have been sent and
      not Acked and reason indicates hint at reason for failure.}
   var sendCode, {0-means EIM data sending is blocked, do not issue
      DtStartData calls.
         1-means even if output window is smaller
         than desired, e.g. zero, issue a DtStartData call at least when
         an E-bit needs sending to enter Rendezvous-at-sender procedure.
         Other codes not relevant for this return.}

      offset, {defined above}
      count,  {defined above}
      reason:integer;  {defined above});

var crPtr:CRpointer; {pointer to the CR for assoc.}
procedure   sendRetry (crPtr:CRpointer; var retryFlg:Boolean; var offset,
            count:integer; sPkt:PKTpointer); {defined in Section
            6.4.2}
procedure   StimerExpired (crPtr:CRpointer); {defined in Section
            6.4.3}
procedure   defaultCR (crPtr:CRpointer); {defined in Section 6.2.4}
function    shouldRetry (crPtr:CRpointer):Boolean; {defined in Section
            6.4.2}
```

```
begin
   getCR (assoc, crPtr);
   if crPtr ≠ nil then {DtTimeout should never have been called when
         there was not a CR}
      begin
         with crPtr↑ do
            begin
               {initialize returns}
               sendCode:= 1;
               retryFlg:= false;
               sPktFlg:= false;
               giveupFlg:= false;
               offset:= 0;
               count:= 0;
               reason:= 0;

               {test for Rtimer → 0 event}
               if (Rtimer > 0) and ((EIMtime - Rtimer) ≥ 0) then
                  begin {Rtimer has expired}
                  {all CR receive variables are reset to or become
                     default values.}
                     Rtimer:= 0;
                     RovflwInd:= false
                  end;

               {test for retrytimer → 0 event}
               if shouldRetry (crPtr) then sendRetry (crPtr, sPkt,
                  sPktFlg, retryFlg, offset, count);

               {test for Stimer → 0 event}
               if (Stimer > 0) and ((EIMtime - Stimer) ≥ 0) then
                  StimerExpired (crPtr, offset, reason, giveupFlg);

               {check to see if send initialization wait interval has
                  expired or some packet has had its maximum
                  retransmission time.}
               if tryData(crPtr) then sendCode:= 1 else sendCode:=0;

               {check to see if CR is in default state and can be
                     deallocated}
               defaultCR (crPtr)
            end
         end
      end {DtTimeout}.
```

6.4.2 Handling Retransmission

The details of how retransmission is handled is an implementation issue outside the protocol. There are two requirements that must be met however. One requirement is that the retransmission interval R (see Appendix A) for each bit or packet be bounded. The number of retransmissions during this interval is an implementation decision. The upper bound is the lifetime interval for a bit or packet, but in practice it will be less than this to assure that the last retransmission can reach the receiver with Plifetime > 0 and thus be accepted.

A second requirement is that when data or a Rendezvous packet exists that has had its maximum number of retransmissions, new transmissions must be stopped as required by the derivation of timer intervals in Appendix A (represented here by SoutPtr ≠ SendPtr).

Because we assume retransmission is unlikely, with properly adjusted retry timers, a simple retransmission model is presented that seems adequate. An entire packet (all data in a Data packet) must be Acked before a packet is removed from the Retry Queue. A packet is the unit of retransmission.

On the assumption that retry is caused by congestion it may be reasonable to stop new transmissions until everything sent has been Acked. This is not done here however.

Within this section we define all procedures involving retransmission even though only some of them are used when DtTimeout is called.

The retry data structure (a queue of RetryRecords) was defined in Section 6.2.5 during the CR definition. Here we give the procedures and functions that operate on this structure. The initial condition of SinPtr = SoutPtr = SendPtr = nil is assumed.

To add a description of a packets Ptr to the Retry Queue in the CR pointed to by crPtr the following procedure is called.

```
procedure addRetryEntry ({args} crPtr:CRpointer; sPtr:PKTpointer{no
   returns});
   var retryPtr = ↑RetryRecord;
   begin
   with sPtr↑, retryPtr↑ do
      begin
         new (retryPtr);
         {fill in RetryRecord}
         rrType := Ptype;
         rrEntryTime := EIMtime;
         rrLifetime := Plifetime;
         rrPID := Pid;
         rrBlink := nil;
         case Ptype of
            Data:rrSNO := Pdl;
            Ack:rrSNO := Pwindow;
            Dcntrl:rrSNO := Psno
            end;
         setTimer (crPtr, rrRetryTimer, AretryTime,true);
         rrFlink := SinPtr;
         if SinPtr = nil then
            begin
               SoutPtr := retryPtr;
               SendPtr := retryPtr
            end
         else rrFlink↑.rrBlink := retryPtr;
         SinPtr := retryPtr
      end
   end {addRetryEntry}.
```

The next procedure deletes the retry entry pointed at by retryPtr.

```
     procedure deleteRetryEntry ({args}crPtr:CRpointer;
retryPtr:RetryPointer {no returns});
        begin
           with crPtr↑, retryPtr↑ do
              begin
                 if rrBlink = nil {head of queue} then SinPtr = rrFlink
                 else rrBlink↑.rrFlink := rrFlink;
                 if rrFlink = nil {tail of queue} then SendPtr = rrBlink
                 else rrFlink↑.rrBlink := rrBlink;
                 if retryPtr = SoutPtr then SoutPtr := rrBlink;
                 dispose (retryPtr)
              end
        end {deleteRetryEntry}.
```

The next procedure searches the Retry Queue and deletes all the Acked entries. If typeFlg = true all Ack packet entries are to be deleted else delete all Data and Rendezvous packets with Pid + rrSNO ≤ sn}

```
     procedure deleteAckedEntries ({args} crPtr:CRpointer;
     sn:SN;typeFlg:Boolean;  {no returns});
        var tempPtr, retryPtr:RetryPointer;
            b:Boolean;
        begin
        retryPtr := crPtr.SinPtr;
        while retryPtr ≠ nil do
           begin
              with crPtr↑, retryPtr↑ do
                 begin
                    b := (typeFlg and (rrType = Ack)) or (not typeFlg and
                       ((rrType ≠ Ack) and ((rrPID + rrSNO) ≤ sn)));
                    tempPtr:= rrFlink
                    if b then
                       begin
                          EIMalarm (assoc,rrRetryTimer, false, false);
                             {cancel alarm}
                          deleteRetryEntry (retryPtr)
                       end;
                    retryPtr:= tempPtr
                 end
           end
        end {deleteAckedEntries}.
```

The following function checks the retry timer of the entry at SoutPtr to see if its retry timer has expired.

```
     function shouldRetry (crPtr:CRpointer):Boolean;
        begin
           with crPtr↑, SoutPtr↑ do
              shouldRetry := (EIMtime - rrRetryTimer) ≥ 0
        end {shouldRetry}.
```

The next procedure generates the DtTimeout returns required when a packet retry is required.

```
    procedure sendRetry ({args} crPtr:CRpointer; sPkt:PKTpointer;
       {returns} var sPktFlg, retryFlg:Boolean; var offset,count:integer);
       begin
          retryFlg:=false;
          offset:=0;
          count:=0;
          sPkt:=nil;
          with crPtr↑, SoutPtr↑ do
             begin
                rrLifetime := rrLifetime - (EIMtime - rrEntryTime); {update
                   retry packet's lifetime}
                if rrLifetime > 0 then
                   begin {send retry}
                      case rrType of
                         Data: begin
                                  SretryInd:= true; {sets retry flag in
                                  CR indicating next DtStartData call is
                                     for retry data}
                                  {set return parameters}
                                  retryFlg:=true;
                                  offset:= rrPid - Sou;
                                  count:= rrSNO
                                  {Retry entry left at SoutPtr.}
                               end;
                         Ack:  {generate an Ack retry packet}
                            begin
                               sendAck (crPtr, true, false, sPkt);
                               sPktFlg:= true;
                               deleteRetryEntry (SoutPtr)
                            end;
                         Rendezvous:  {generate a Rendezvous retry packet}
                             begin
                               sendRendezvous (crPtr, true, sPkt);
                               sPktFlg:= true;
                               delete RetryEntry (SoutPtr)
                            end
                      end {case}
                   end
                else {entry has had max retries}
                   case rrType of
                      Ack: deleteRetryEntry (SoutPtr);

                      Rendezvous, Data:
                         begin {leave on Retry Queue in case never Acked so
                            error can be reported}
                               rrLifetime := 0;
                               rrRetryTimer := 0;
                               SoutPtr := SoutPtr↑.Blink
                         end
                   end
             end
       end {sendRetry}.
```

6.4.3 Send Timer Expiration

When Stimer expires either of the following two cases could exist:

(1) all bits and packets sent have been Acked.

(2) there are outstanding unAcked bits or an unAcked Rendezvous packet. UnAcked reliable-Acks are removed from the retry structure when they have had their maximum retransmissions. The rules for handling CR state in these cases are imbedded in the following procedure which prepares returns for DtTimeout.

```
     procedure StimerExpired ({args} crPtr:CRpointer; {returns} var
offset, reason:integer, giveupFlg:Boolean); {parameters defined earlier for
DtTimeout.}
     var tempPtr,retryPtr:RetryPointer;
        begin
           with crPtr↑ do
              begin
                 if (Sou = Sowle) then {case 1, no-op}
                 else if (SinPtr ≠ nil) then {case 2, there is unAcked
                 data or an unAcked Rendezvous packet}
                    begin
                       {Output Function}
                       giveupFlg:=true;
                       if SrendSenderInd then offset:= 0 {unAcked Rendezvous
                          packet} else offset:= Sowle-Sou; {reports offset
                          bits ambiguous}
                       reason:= SnakReason {possible reason for problem};
                       {reinitialize CR send variables to default values or
                          they are default already}
                       SrendSenderInd:= false;
                       SseriousNakInd := false;
                       retryPtr:= SinPtr
                       while retryPtr ≠ nil do
                          begin
                             tempPtr:= rrFlink;
                             dispose (retryPtr);
                             retryPtr:= tempPtr
                          end;
                       SinPtr := nil;
                       SoutPtr := nil;
                       SendPtr := nil;
                    end;
                 Stimer:=0;
                 SeSentInd:= false;
                 SnakReason := 0;
                 SovflwInd:= false
                 {other send variables are in default state.}
              end
        end {StimerExpired}.
```

6.5 User Interface Events

6.5.1 Receive or Ack Generation Events

The procedure DtAck and included procedures represent Delta-t's rules for Ack formation and state update. DtAck is called whenever an Ack is required. An Ack is required when (1) an event occurs within the EIM (due to IPC-user Receive or Receive-Aborts or implementation dependent events) affecting the receive window that should be advertised to the other end or (2) when Delta-t indicates with the AckFlg in the return from DtPktRcvd that DtAck should be called in order to provide Delta-t with the current window state so it can generate an Ack packet (caused by receipt of a Data or Rendezvous packet). The receive window to be reported to Delta-t is the amount of Receive-specific buffer available for the association when an ISR has been allocated, otherwise a default window is reported (see Appendix B).

The EIM indicates a reliable Ack is required whenever the input window goes from zero to nonzero and the ISR variable RSind is true (see Appendix B).

The EIM can schedule the issuing of the DtAck call as appropriate (and thus one Ack can acknowledge one or more received packets) subject to the constraint that it is understood that when Delta-t indicates an Ack should be issued its lifetime is counting down.

If a CR does not exist and space for a CR cannot be obtained the procedure fails.

```
procedure DtAck (
   {args}
   assoc:AR; {association record}
   rWindow:integer; {number of bits of receive buffer space
      available for the association}
   rsFlg:Boolean; {if true the other end needs to be reliably notified
      in a reliable-Ack packet that a zero window is opening.}
   sPkt:PKTpointer;{pointer to a packet buffer for an Ack packet.}
   {returns}
   var errorFlg:Boolean;{true if no CR space is available});

   var crPtr:CRpointer;

   begin
      getCR (assoc, crPtr);
      if CR = nil then errorFlg:= true
      else
         begin
            with crPtr↑ do
               begin
                  Riwre:= Riwle + rWindow;
                  errorFlg:= false;
                  sendAck (crPtr, sPkt, false, rsFlg)
               end
         end
   end {DtAck}
```

The following procedure represents correct CR send state update for Ack packet sending and calls procedure createAck to generate an Ack packet.

```
    procedure sendAck ({args}crPtr:CRpointer; sPkt:PKTpointer;
                retryFlg,rsFlg:Boolean); {retryFlg indicates Ack is a
                retry, rsFlg  indicates reliable-Ack should be sent for
                rendezvous-at-sender, sPtr is a pointer to a packet buffer to
                contain the Ack.}

        procedure createAck ({args} crPtr:CRpointer;sPkt:PKTpointer
{also                   return}; retryFlg, rsFlg:Boolean); {defined below}
        begin
        with crPtr↑ do
          begin
            createAck (crPtr, sPkt, retryFlg, rsFlg);
            StimeStamp:=0;
            if Praf then
              begin
                addRetryEntry (crPtr, sPkt);
                if not retryFlg then setTimer (crPtr, Stimer,
                  3*2**AΔtexp, true) {resetting Stimer because a
                  packet needing an Ack is being sent}
              end;
          end
        end {sendAck}.
```

The following procedure specifies correct formation of an Ack packet. This procedure will fill the packet buffer pointed to by sPkt as an Ack packet. retryFlg indicates whether or not this is a new (false) or retry (true) packet. rsFlg indicates whether (true) or not (false) a reliable-Ack should be generated.

```
        procedure createAck ({args} crPtr:CRpointer, sPkt:PKTpointer;
{also                 return}, retryFlg, rsFlg:Boolean);
          begin
            with crPtr↑, sPkt↑do
              begin
                Pver := {DeltaGram version number as appropriate};
                Ptype := Ack;
                Presl:= 0;
                Pdn:= true;
                PprtctLev := {as appropriate for protection policy.};
                if Rtimer > 0 then PΔtexp := RΔtexp else
                  PΔtexp:=AΔtexp;
                Pid := Riwle;
                Pdestaddr := Aassoc.destAddr;
                Poriginaddr := Aassoc.originAddr;
                Pwof:= RovflwInd;
                Ppuf:= (Rtimer=0);
                Pres5:=0;
                Pres6:=0;
                PΔtver:= {Delta-t version number as appropriate}
```

```
            if retryFlg then
               begin
                  Praf := true {wouldn't be on retry list if
                     reliable delivery not desired}
                  Pwindow:=SoutPtr↑.rrSNO;
                  Plifetime := SoutPtr↑.rrlifetime
               end
            else
               begin
                  Pwindow:=Riwre-Riwle;
                  Praf := ((Pwindow > 0) and rsFlg);
                  if Rtimer = 0 then Plifetime := 255
                  else
                     if StimeStamp > 0 then
                        Plifetime := 2**RΔtexp-(EIMtime-StimeStamp)
                     else Plifetime := 2**RΔtexp
               end
            headerChecksum (sPkt)
         end
   end {createAck}.
```

6.5.2 Data or Rendezvous Packet Sending Event

6.5.2.1 DtStartData and DtFinishData

The procedures DtStartData and DtFinishData are called consecutively to send data for the first time, to send retry data, or to cause a header only data packet to be sent to Ack a reliable-Ack. DtStartData may also result in a Rendezvous packet being generated, in which case DtFinishData does not need to be called.

DtStartData is called by the EIM either when (1) there is data to send and the sendCode in the ISR is 1 (e.g. should try to send even if the output window is zero so that a Rendezvous packet will be sent) or (2) when the sendCode returned from the DtPktRcvd procedure is 2 indicating that a Data packet (even if header-only) is required to Ack a reliable-Ack. Data is sent in the sequence issued by the IPC user. DtFinishData should be called to complete a data packet header and after the EIM has placed count2 bits of data in the packet.

```
procedure DtStartData (
         {args}
          assoc:AR; {association record}
          Bflg,
          Eflg:Boolean; {Bflg indicates that the first data bit is to
                         be labeled by a B mark and Eflg indicates that
                         the last data bit as specified by count1 is to be
                         labeled by an E mark.}
          prtctLev,  {protection level of the data}
          owreOffset, {EIM's view of the output window.  Same value as
                             returned to EIM in DtPktRcvd as owreOffset or
                             standard default}.
          count1:integer;  {number of bits of data potentially
                            available for a packet.}
         sPkt:PKTpointer; {pointer to packet header buffer.}
```

```
{returns}
var count2, {count of the number of bits of data that are to
            be placed in this packet.  For a retry, as modeled
            here, count2 must be the number returned in
            DtTimeout.}
   sendCode:integer; {0 - means EIM data sending is blocked,
      do not issue DtStartData calls even if nonzero output
      window.
      1 - means even if output window is smaller than desired
      issue a DtStartData call with nonzero data count when new
      data needs sending (e.g. to enter Rendezvous-at-sender
      procedure.
      Other codes not relevant for this return.}
var typeFlg, {true if Data packet being formed, false if
   Rendezvous packet.}
errorFlg:Boolean  {error flag set true if no CR space
                  available.});

var     crPtr:CRpointer;
procedure  sendData ({defined below});
function  shouldData ({defined below});
function  shouldRendezvous ({defined below});

begin
   count2:= 0;
   errorFlg:= false;
   sendCode:= 1;
   getCR (assoc, crPtr);
   if crPtr = nil then errorFlg:= true
   else
      with crPtr↑, SoutPtr↑ do
            begin
               Sowre:= Sou+owreOffset;
               if (SretryInd or shouldData(crPtr,count1) then
                  {Data packet should be sent}
                  begin
                     typeFlg:= true;
                     sendData
                        (crPtr,sPkt,count1,prtctLev,Bflg,Eflg,count2)
                  end
               else if shouldRendezvous (crPtr, count1) then
                  begin
                     typeFlg:= false;
                     sendRendezvous (crPtr, sPkt, false)
                  end;
               if tryData (crPtr) then sendCode: = 1 else
                  sendCode:= 0
            end
         end {DtStartData}.
```

The following procedure is called after DtStartData, if typeFlg = true (Data packet) is returned by DtStartData.

```
DtFinishData (sPkt:PKTpointer {full packet buffer});
     procedure dataChecksum (sPtr:PKTpointer);
        begin
           {data checksum algorithm as defined in DelatGram Specification
           [19]}
        end {dataChecksum};
     begin
        dataChecksum (sPkt);
        headerChecksum (sPkt)
     end {DtFinishData}.

     procedure headerChecksum; (sPkt:PKTpointer);
        begin
           {header checksum algorithm as defined in DeltaGram Specification
           [19]}
        end {headerChecksum}.
```

6.5.2.2 Sending a Data Packet

There are a set of conditions (1) that indicate a Data packet should not be sent for correct protocol operation and (2) a set that indicate that for efficiency one should not be sent (possibly dependent on the implementation). We only indicate one type 2 condition here.

The Boolean function tryData is a function of the subset of these conditions that determines if the EIM should issue DtStartData calls to try and send data. The tryData conditions and others affecting the decision to actually send a Data packet are incorporated in the function shouldData. The function tryData is required because of the EIM to Delta-t interface presented here. Note that it is possible for tryData to be true and the output window to be zero. This results because, as modeled here, Delta-t does not automatically enter the rendezvous-at-sender procedure when it receives a zero input window in an Ack but instead waits until an attempt is made to send Data (by a DtStartData call being issued with a nonzero data count) causing enter to the rendezvous-at-sender mechanism. Therefore, if sendCode in the EIM's ISR is 1 it should issue a DtStartData call when it has data to send (an E-bit) even if the output window is zero (see Appendix B). The value of sendCode returned by DtStartData will then indicate no further DtStartData calls should be made until the window opens.

A different model of the EIM to Delta-t interface could, for example, allow the EIM to indicate to Delta-t that it should automatically enter the rendezvous-at-sender procedure when a zero output window exists.

```
     function tryData (crPtr:CRpointer):Boolean;
        begin
           tryData:=    ((EIMtime-Aidt) > 3*2**AΔtexp){1 the
                           initialization wait interval is expired})
                        and
                           (SoutPtr = SendPtr) {1, no packets
                           exist that have had their maximum number of
                           retries}
```

```
          and
             not SrendSenderInd {1, not in rendezvous-at-sender
             state}
          and
             not SseriousNakInd{2}
     end {tryData}.
```

The following function determines whether or not Delta-t should send a Data packet. Only send a Data packet if the tryData conditions are satisfied and overflow has not occurred (a Rendezvous packet must be sent) and either the output window is nonzero or a header-only data packet needs sending.

```
function  shouldData (crPtr:CRpointer; count:integer):Boolean;
  begin
    shouldData:= tryData(crPtr) and not SovflwInd {1} and ((Sowre
       > Sowle {2}) or (count = 0){1})
  end {shouldData}.
```

The following procedure determines, based on the maximum packet size for the association and output window size, how much data should be sent (count2), whether or not Pe should be set in the Data packet header and shows correct CR state update when a Data packet is sent.

```
procedure sendData ({args} crPtr:CRpointer;
  sPkt:PKTpointer {pointer to a Data packet header buffer to be filled
     in};count1 {number of bits of data available for sending},
     prtctLev:integer;Bflg,Eflg:Boolean{data labels};{returns
     var} count2:integer {number of bits of data that EIM is to
     place in packet.});

  var eFlg:Boolean;
  procedure startDataHeader ({defined below});

     begin
        eFlg:= false;
        with crPtr↑ do
           begin
              {set up parameters required for procedure
                 startDataHeader and update send state.}
              if SretryInd then
                 begin {retry}
                    count2:= SoutPtr.rrSNO;
                    eFlg:= Eflg
                 end
              else {not a retry}
                 begin
                    count2:= min (AmaxPktSize, count1, Sowre - Sowle);
                       {number of bits that can be placed in a packet
                       is min of max packet size for assoc, bits
                       available, and output window}
                    eflg:= (count2 = count1) and Eflg;
```

```
                Sowle:= Sowle + count2; {update by number of bits
                   being sent.}
                if eFlg then
                   begin
                      Sowre:= Sowle; {close window, no data is
                         sent after an E-bit until it is Acked.}
                      SeSentInd:= true
                   end;
                if (count2 > 0) then SetTimer (crPtr,
                         Stimer, 3*2**AΔtexp, true)
             end;
          startDataHeader (crPtr, sPkt, count2, prtctLev,
                   Bflg, eflg);
          if (count2 > 0) then addRetryEntry(crPtr, sPkt) {only
             need to retry if data sent.}
       end
    end {DtStartData}
```

The following procedure specifies the rules for correct Data packet header formation.

```
procedure startDataHeader ({args} crPtr:CRpointer; sPkt:PKTpointer;
count:integer; prtctLev:integer; b,e:Boolean);
   begin
      with crPtr↑, sPkt↑ do
         begin
            Pver := {DeltaGram version number as appropriate};
            Ptype := Data;
            Presl:= 0;
            Pdn:= false;
            PprtctLev :=prtctLev;
            PΔtexp := AΔtexp;
            Pdestaddr := Aassoc.destAddr;
            Poriginaddr := Aassoc.originAddr;
            Pt := false;
            Pds:= false;
            Pb := b;
            Pe:= e;
            Pres2:= 0;
            Pdl:= count;
            Pres3:= 0;
            Pres4:= 0;
            PΔtver:= {Delta-t version number as appropriate};
            if SretryInd then
               begin
                  Plifetime := SoutPtr↑.rrLifetime;
                  Pid := SoutPtr↑.rrpid;
                  Pdrf := (Pid ≤ Sou); {everything sent previously
                     has been Acked}
                  SretryInd:= false;
                  deleteRetryEntry(SouPtr)
               end
```

```
            else
               begin
                  Plifetime := 2**AΔtexp;
                  Pid := Sowle;
                  Pdrf := (Sou = Sowle)
               end
            {The PhdrChksum and PdataChksum fields are set in the
                  procedure DtFinishData.}
         end
      end {startDataHeader}.
```

6.5.2.3 Sending a Rendezvous Packet

The following function specifies the rule for sending a Rendezvous packet. A Rendezvous packet should be sent if not already in the rendezvous-at-sender state and (overflow has occurred or (there are bits to send and no output window and all data previously sent has been Acked).

```
   function shouldRendezvous (crPtr:CRpointer;countl:integer):Boolean;
      begin
         with crPtr↑ do
            begin
               shouldRendezvous := (not SrendSenderInd)
               {not in rendezvous-at-Sender state}
                  and
                  (SovflwInd or ((countl > 0) and (Sowle = Sowre) and
                     (Sou = Sowle)))
            end
      end {shouldRendezvous}.
```

The following procedure calls procedure createRendezvous and performs correct state update when a Rendezvous packet is to be sent.

```
   procedure sendRendezvous ({args} crPtr:CRpointer; sPkt:PKTpointer;
retryFlg:Boolean);

      procedure createRendezvous ({args} crPtr:CRpointer;
         sPkt:PKTpointer;
         retryFlg:Boolean); {defined below}
      begin
         with crPtr↑, do
            begin
               createRendezvous (crPtr, sPkt, retryFlg);
               addRetryEntry (crPtr, sPkt);
               if not retryFlg then
                  begin
                     setTimer (crPtr, Stimer, 3*2**AΔtexp);
                     SovflwInd:= false;
                     SrendSenderInd:= true
                  end;
         end
      end {sendRendezvous}
```

The following procedure specifies the rules for Rendezvous packet formation.

```
procedure createRendezvous ({args} crPtr:CRpointer; sPkt:PKTpointer
{and return}; retryFlg:Boolean);
   const n = {> 0, implementation convenient value used in Psno};
   begin
      with crPtr↑, sPkt↑ do
         begin
            Pver := {DeltaGram version as appropriate};
            Ptype := Dcntrl;
            Presl:= 0;
            Pon := true;
            PrtctLev :=  {as required by protection policy{;
            PΔtexp := AΔtexp;
            Pdestaddr := AdestAddr;
            Poriginaddr := AoriginAddr;
            Psubtype := 0;
            Pdrf := true;
            Pres7 := 0;
            Pres8 := 0;
            PΔtver:= {Delta-t version number as appropriate}
            if retryFlg then
                  begin
                     Psno := SoutPtr↑.rrSNO;
                     Plifetime := SoutPtr↑.rrLifetime;
                     Pid := SoutPtr↑.rrPID
                  end
               else
                  begin
                     if Sowle ≠ Sou then Psno := Sowle - Sou;
                        {Rendezvous sent due to overflow}
                     else
                        begin {Rendezvous sent due to just zero
                           window}
                           Psno := n;{consume SN space for assurance}
                           Sowle := Sowle + n;
                           Sowre := Sowre + n
                        end;
                     Plifetime := 2**AΔtexp;
                     Pid := Sou
                  end;
               headerChksum (sPkt)
         end
   end {createRendezvous}.
```

6.6 Packet Received Event

6.6.1 DtPktRcvd

DtPktRcvd and included procedures specify Delta-t's rules for packet acceptance testing and processing. DtPktRcvd is called when the EIM receives a packet from the next lower level. The return parameters are dependent on the type of packet received. If the packet received is a Data (and a Nak is not generated) or Rendezvous packet the call to this procedure is followed eventually by a DtAck call to update the input window and generate an Ack. If the packet received is a reliable-Ack the call to this procedure is followed eventually by DtStartData and DtFinishData calls to cause a Data packet, possibly header-only, to be sent. DtPktRcvd should have high enough priority so that packet lifetimes are unlikely to expire due to long packet queuing delays.

```
procedure   DtPktRcvd (
    {args}
    assoc:AR; {association record}
    rPkt, =  {pointer to header buffer for the received packet.
       The size of the packet can be determined from the packet type and,
       if a Data packet, the Pdl field.}
    sPkt:PKT; {packet header buffer for possible Nak packet}
    timeStamp:dateTime; {time packet was received}
    rWindow:integer; {number of bits of potential buffer space
       available for association.}

    {returns}
    var type:integer; {packet type or value indicating ignore other
       returns.}
    var ackFlg:Boolean; {If true the EIM should issue a DtAck call at a
    convenient point to cause an Ack packet to be sent with latest receive
    window.}

    {Data packet}
    var offset,        {offset relative to start of packet at which to
                       obtain first data bit}
       count,          {number of bits to accept}
       prtctLev:integer; {protection level of the data}
    var Bflg,
       Eflg      {flags indicating respectively whether first accepted
                 bit is labeled by a B mark and the last acepted bit is
                 labeled by an E mark.}
       nakFlg, {true if Nak formed}

    {Ack packet}
    var ovflwFlg:Boolean;    {flag if true all data bits at queue
                             position ouPtr + ackOffset and beyond have
                             overflowed and should be reset as if never
                             sent and be sent again.}
    var ackOffset,        {SN offset relative to ouPtr in ISR for the
                          number of data bits Acked}
       sendCode,    {(Also returned for Nak packets) 0 - means data
                    sending is blocked, do not issue DtStartData calls.
                       1 - means even if owreOffset is smaller than desired
                          (including zero), issue a DtStartData call when
                          data needs sending to enter rendezvous-at-sender
                          procedure.
                       2 - means issue a DtStartData call, even if there is
                          no data needing sending to cause a Data packet to
                          be sent to Ack a reliable-Ack.}
```

```
      owreOffset:integer;    {The output window.  This information is
                              passed to the EIM for possible saving in its
                              ISR and subsequent return to Delta-t as an
                              efficiency aid and when the CR is reclaimed.
                              How owreOffset and sendCode can be used by the
                              EIM in its policy for issuing DtStartData
                              calls is discussed in Appendix B.}

   {Rendezvous packet}
   var rsFlg:Boolean;        {This returned flag indicates that the other
                              end wants to be reliably informed when the
                              input window opens.})

   {Nak packet}
      {The parameter sendCode above is also returned for received Nak
         packets};

   const n = {value indicating ignore other returns};
   var   crPtr:CRpointer;
         remainingLifetime:integer;
   procedure  processData ({defined in Section 6.6.2});
   procedure  processAck ({defined in Section 6.6.3});
   procedure  processRendezvous ({defined in Section 6.6.4});
   procedure  processNak ({defined in Section 6.6.5});
   procedure  sendNak ({defined in Section 6.6.6});

begin
   type:= n;
   getCR (assoc, crPtr);
   if crPtr ≠ nil then {if crPtr = nil packet will be discarded and
      become "lost"}
      begin
         ackFlg:= false;
         nakFlg:= false;
         offset:= 0;
         count:= 0;
         prtctLev:= 0; {or should it be highest level?}
         Bflg:= false;
         Eflg:= false;
         ackOffset:= 0;
         owreOffset:= 0;
         ovflwFlg:= false;
         sendCode:= 1;
         rsFlg:= false;
         DGadjustLifetime (EIMtime-timeStamp, 0, rPkt,remainingLifetime)
            {adjusts lifetime for time spent on Delta-t queue and
            checks to see if lifetime has expired.  If lifetime has
            expired remainingLifetime returns ≤ 0.};
         if remainingLifetime ≤ 0 then with rPkt↑do
            if (Ptype = Data) then
               begin
                  nakFlg:= true;
                  type:= Data;
```

```
                  sendNak (crPtr, rPtr, sPkt, 3,{lifetime
                     expired}remainingLifetime)
               end
                  {Sending the Nak is optional.}
            else
               begin
                  type:= Ptype;
                  {code for switch to appropriate version routines would
                     go here}
                  case type of
                     Data:  processData (crPtr, rPkt, sPkt, rWindow,
                        offset, count, prtctLev, ackFlg, Bflg, Eflg,
                        nakFlg);
                     Ack:  processAck (crPtr, rPkt, ackOffset, ovflwFlg,
                        owreOffset, sendCode);
                     Nak:  processNak (crPtr, rPkt, sendCode);
                     Dcntrl:  if rPkt.Psubtype = 1 then processRendezvous
                        (crPtr, rPkt, rWindow, ackFlg, rsFlg);
                  end
               end
      end
   end {DtPktRcvd}.
```

6.6.2 Receipt of a Data Packet

Data packets serve two functions in this protocol, the main one is to carry data, the secondary one, as part of window management, is to "Ack" a reliable-Ack that is reporting the opening of a zero window, completing the rendezvous-at-the-sender procedure. In order for a Data packet to be accepted there must have been sufficient time since the Delta-t environment was initialized and the SN of at least one bit in the packet, or the Pid (in the case of dataless Data packets) must equal Riwle. If a bit is accepted or overflow occurs Rtimer is updated.

The procedure processData checks the Data packet for acceptance by calling acceptData, specifies correct update of the CR, determines what data to accept, and returns parameters to the EIM which then copies the accepted data to buffers it manages. The EIM will signal the user if a Receive completes. When an Ack is required, the EIM will call DtAck to report its current window and an Ack will be generated.

```
   procedure   processData ({args} crPtr:CRpointer; rPkt,
               sPkt:PKTpointer; rWindow, {returns} var offset, count,
               prtctLev:integer;var ackFlg,Bflg, Eflg,nakFlg:Boolean);

      const n = {large number};
      var temp:integer; b:Boolean;
      procedure   acceptData (crPtr:CRpointer; rPkt sPkt:PKTpointer; var
                  ackFlg, nakFlg, b:Boolean); {defined below};
```

```
begin
  with crPtr↑ do
    begin
      acceptData (crPtr, rPkt, sPkt, ackFlg, nakFlg, b,);

      if b then

      begin {packet accepted}
        deleteAckedEntries (crPtr, true, 0); {see discussion
          of retry in Section 6.4.2. This procedure deletes any
          Acks from the retry structure}
        if Rtimer = 0 then
          begin {update CR receive variables}
            Riwle := Pid;
            RΔtexp := PΔtexp
          end;
        prtctLev:= PprtctLev;
        temp:= Pdl-(Riwle-Pid); {number of data bits at and to
          right of Riwle}
        offset:= 256 + (Riwle-Pid); {offset in packet to begin
          accepting data, assumes bits in header run 0-255}
        Bflg:= (Pb and (offset = 256));
        Riwre:= Riwle + rWindow;
        count:= min (temp, Riwre-Riwle,n); {number of bits that
          can be accepted}
        Eflg:= (Pe and (count = temp));{last accepted bit is
          labeled E}
      if (count > 0) then setTimer (crPtr, Rtimer,
        2*2**RΔtexp, false);
      RovflwInd:= (count ≠ temp);
      Riwle:= Riwle + count;
      end
    end
  end {processData}.
```

The following procedure and associated functions specify the rules for Data packet acceptance. To be accepted there must have been enough time since the environment was initialized, the receiver is not in the overflow state, the packet must contain data insequence, and there must be at least one SN on the input-window-left-edge. Note that if Rtimer > 0 then Pdrf is ignored in the function SNonWindowEdge. The procedure acceptData also determines if an Ack or Nak packet should be generated and starts the lifetime of the Ack counting down.

```
procedure acceptData ({args} crPtr:CRpointer, rPkt, sPkt:PKTpointer
  {returns} var ackFlg, nakFlg, b:Boolean);

  var temp:dateTime;
  function outOfSequence (crPtr:CRpointer; rPkt:PKTpointer); {see
    below};
  function SNonWindowEdge (crPtr:CRpointer; rPkt:PKTpointer); {see
    below};
```

```
procedure handleOutOfSequence (rPkt, sPkt:PKTpointer); {see below};

begin
   with crPtr↑, rPkt↑ do
      begin
         ackFlg:= true; {initialize to generate Ack whether accepted
            or not}
         b := false; {initialized to reject packet}
         temp:= StimeStamp; {save in case a Nak generated so can be
            restored}
         StimeStamp:= EIMtime; {The lifetime of the Ack begins
            here.}
         if ((EIMtime -Aidt) ≥ 2**PΔtexp) and not RovflwInd) then
            begin {interval since initialization long enough and not
                  in overflow state}
               if SNonWindowEdge (crPtr, rPkt) then b:= true
                  else if outOfSequence (crPtr, rPkt) then
                     begin
                        handleOutOfSequence (crPtr,rPkt,sPkt,nakFlg);
                        ackFlg:= false;
                        StimeStamp:= temp {Nak is generated
                           immediately so uses PΔtexp in packet
                           being Naked.  An Ack may be generated from
                           an earlier packet receipt and need to keep
                           its lifetime aging.}
                     end
            end
      end
end {acceptData}.
```

The following function tests for duplicate data. Duplicate zero length data packets might be accepted, but cause no harm. Acceptance is independent of whether or not a window exists large enough to hold any data.}

```
function SNonWindowEdge (crPtr:CRpointer; rPkt:PKTpointer):Boolean;
   begin
      with crPtr↑, rPkt↑ do
         SNonWindowEdge := ((Rtimer = 0) and Pdrf) or ((Rtimer > 0) and
         ((Pdl > 0) and (Pid ≤ Riwle) and (Riwle ≤ Pid + Pdl-1)) or
         ((Pdl = 0) and (Pid = Riwle))); {When Rtimer = 0 and Pdrf any
         SN is acceptable otherwise at least one bit is at Riwle or Pid =
         Riwle when Pdl = 0}
   end {SNonWindowEdge}.
```

The following function tests for out of sequence data.

```
function outOfSequence (crPtr:CRpointer; rPtr = ↑PKT):Boolean;
   begin
      with crPtr↑, rPkt↑ do
         outOfSequence := ((Rtimer = 0) and
         (not Pdrf)) or ((Rtimer > 0) and (Riwle < Pid))
   end {outOfSequence}.
```

The following procedure handles out of sequence data. Whether or not out-of-sequence Data packets are accepted is an implementation option. If they are accepted they would be queued until they are insequence. This queue would be examined periodically; for example, after each data packet with data was processed. The queue would be cleared when overflow occurred. Plifetime must continue counting down. Here we just generate a Nak

```
procedure handleOutOfSequence (crPtr:CRpointer; rPkt, sPkt:PKTpointer; var
          nakFlg:Boolean);
        begin {}
          sendNak (rPkt, sPkt,129 {out of sequence},0);
          nakFlg: = true
        end {handleOutOfSequence}.
```

6.6.3 Receipt of an Ack Packet

Missequenced, lost, or duplicate Ack packets can cause no assurance harm, although such occurrences may lead to the exchange of extra packets, as discussed under window management in Section 2.7.3.

The procedure processAck specifies the rule for Ack packet acceptance and correct state update when an Ack packet is accepted. It calls acceptAck to test an Ack packet for acceptance. Some duplicate or missequenced Acks are rejected, but not all. Duplicate or missequenced Acks with Pid=Sowle or that have Ppuf set true are not detectable.

The procedure must handle two main cases (1) data or a Rendezvous packet may be Acked or data overflow has occurred, or (2) only a relative flow control window is being reported. It must also recognize when a reliable-Ack is received.

```
     procedure processAck ({args} crPtr:CRpointer; rPkt:PKTpointer;
     {returns} var ackOffset, owreOffset,sendCode:integer; var
     ovflwFlg:Boolean);

        function acceptAck (crPtr:CRpointer; rPkt:PKTpointer):Boolean;
        {defined
          below}

        begin
        with crPtr↑, rPkt↑ do
          begin
            if acceptAck (crPtr, rPkt) then
              begin
                if (not Ppuf and (Stimer > 0)) then
                  begin {State update and output functions when Data
                        bits or Rendezvous packet may be Acked or data
                        overflow has occurred}
                    if SrendSenderInd then ackOffset:= 0
                      {Ack is for Rendezvous packet; No data is sent
                      while SrendSenderInd = true}
                    else ackOffset:= Pid - Sou; {Ack is for data and
                      this is the number of bits Acked}
                    Sou:= Pid; {update Sou for SNs Acked}
```

```
                    if Sou = Sowle then
                       begin {Everything sent has been Acked}
                          SeSentInd:= false;
                          SseriousNakInd:= false;
                          SnakReason:= 0
                       end;
                    owreOffset:= Pwindow;
                    if Pwof then
                       begin {window overflow has occurred; state in
                             EIM and Retry Queue must be reset as if
                             overflow bits were never sent.  Rendezvous
                             packet must be sent eventually.}
                          ovflwFlg:= true;
                          SovflwInd:= true;
                          deleteAckedEntries (crPtr, false, Sowle)
                             {delete Acked and overflow data from
                             Retry Queue.}
                       end
                          else deleteAckedEntries (crPtr, false, Sou)
                             {delete only Acked Data or Rendezvous
                             packets from Retry Queue}
                 end
              {State update and output function for all accepted Ack
                 packets}
              if not SeSentInd then Sowre:= Sowle + Pwindow; {update
                 window whether Ack acks anything or not}
              if Sowre > Sowle) then SrendSenderInd := false;
                 {an out of sequence or old duplicate Ack could cause
                 SrendSenderInd to be reset and data to be sent which
                 would overflow and entry to the Rendezvous-at-sender
                 cycle to be repeated.  No harm results}
              if Praf then sendCode:= 2 else if tryData(crPtr)
                 then sendCode:= 1 else sendCode:= 0
           end
       end {with}
    end {processAck}
```

The following function specifies the rule for Ack packet acceptance. Stimer > 0 implicitly indicates adequate time since environment initialization has occurred. This is also true for Nak packets. Duplicate or missequenced Acks just reporting a window change cause no harm, other than causing possible extra packets being sent. Accept Ack is written to reject duplicate Acks when unAcked SNs exist.

```
    function acceptAck (crPtr:CRpointer; rPkt:PKTpointer):Boolean;
       begin
          with crPtr↑, rPkt↑ do

          acceptAck := ((Praf and (Pwindow > 0)) or not Praf) {Praf is
                          only used to reliably report a window opening.}
```

```
                 and
                   (Ppuf
                   or (Stimer = 0) {relative input window being reported
                   when Riwle, or Sou and Sowle undefined}
                   or
                   ((Stimer > 0) and
                   (((Sou < Pid) and (Pid ≤ Sowle)) {Acks data or
                   Rendezvous packet}
                     or
                     ((Sou=Sowle) and (Pid=Sowle)){just reports input
                       window change})))
       end {acceptAck}.
```

6.6.4 Receipt of a Rendezvous Packet

The purpose currently envisioned for the control called the Rendezvous packet is to indicate to the receiver that it should translate its SN space and turn off RovflwInd and begin accepting Data packets again and that the sender state shows a zero window (with or without overflow), there is more data to send, and that the sender will wait for a reliable-Ack to arrive indicating the window has opened (rendezvous-at-the-sender). The Rendezvous packet with Pdrf = true performs the above.

As currently used, Pdrf is always set true.

Rendezvous-at-the-sender has to be done in a reliable way. The Rendezvous packet needs acknowledgment and the window opening control needs acknowledgment. Rendezvous packets consume SN space and are therefore protected against loss, duplication, or missequencing. Reliable-Ack packets (packets with Praf = true) indicating a nonzero window are "acked" by an acceptable Data packet. Reliable-Ack packets are retransmitted until Acked. This protects against lost packets. Duplication or missequencing of Acks are not a problem as the mechanism will at most cause extra packets to be sent as a result of these hazards, but no improper acceptance of data or sender being blocked permanently can take place.

Rendezvous packets might also be used, in general, to force the receiver to return its state to the sender or adjust its expected SN, but no need for such purposes currently exists.

The following procedure specifies the rule for Rendezvous acceptance testing in function acceptRendezvous and correct state update. An Ack packet will be generated.

```
    procedure processRendezvous ({args} crPtr:CRpointer; rPkt:PKTpointer;
       rWindow:integer; {returns} var rsFlg,ackFlg:Boolean);

       function acceptRendezvous (crPtr:CRpointer; rPkt:PKTpointer):Boolean;
            {defined below}

       begin
         with crPtr↑ do
           begin
```

```
            if acceptRendezvous (crPtr, rPkt) then
               begin
                  deleteAckedEntries (crPtr, true, 0) {deletes
                     any Acks from Retry Queue}
                  if Rtimer = 0 then
                     begin
                        Riwle:= Pid;
                        RΔtexp:= PΔtexp
                     end;
                  setTimer (crPtr, Rtimer, 2*2**RΔtexp, false){a new
                     SN was accepted}
                  RovflwInd := false;
                  Riwle := Riwle + Psno;
                  Riwre:= Riwle + rWindow;
                  rsFlg:= Riwre = Riwle
               end
            ackFlg:= true; {an Ack is to be sent whether or not packet
               accepted.}
         end
      end {processRendezvous}.
```

The following function specifies the Rendezvous packet acceptance condition: enough time has elapsed since the environment was initialized and SN space is consumed and Pdrf is true if Rtimer = 0 or the Pid is on the left window edge if Rtime > 0.

```
      function acceptRendezvous (crPtr:CRpointer; rPkt:PKTpointer): Boolean;
         begin
            with crPtr↑, rPtr↑ do
                  acceptRendezvous := ((EIMtime - Aidt) ≥ 2**PΔtexp)
               and (Psno > 0)
               and ((Pdrf and (Rtimer = 0))
                  or ((Pid = Riwle) and (Rtimer > 0)))
         end {acceptRendezvous}.
```

6.6.5 Receipt of a Nak Packet

Nak packets are not essential to the correct operation of Delta-t. They have been included to allow for possible earlier retransmission of Naked data and to provide diagnostic information. It is important that an error not be reported to the IPC user until Stimer has expired as there may have been a duplicate of the Naked packet that succeeded and an Ack may yet be received. This situation is likely to be rare in the class called possiblyFatal below. This situation could result from a failure of the header checksum to detect an error or in a network partition or crash. We believe an implementation should generate Naks, in case partners are using them for diagnostic or earlier retransmittsion purposes but it could choose to ignore them on receiving.

The following procedure represents an example handling of a Nak packet.

```
   procedure processNak ({args} crPtr:CRpointer; rPkt:PKTpointer
      {returns} var sendCode:integer);
```

```
type possiblyFatal = (1,2,5,128); {cannot reach destination, no
        such destAddr, improper protection level, refuse to accept}
   canImmediatelyRetry = (129,4,3); {out of sequence, data checksum
        error, lifetime expired}

function acceptNak (crPtr:CRpointer; rPkt:PKTpointer):Boolean;
{defined below}

begin
   with crPtr↑, rPkt↑ do
      begin
         if acceptNak (crPtr, rPtr) then
            begin
               SnakReason := PnakReason;
               case PnakReason of
                  possiblyFatal:SseriousNakInd := true; {This will
                           prevent new data being sent on the
                           assumption that the problem will presist,
                           although allowed to continue on the chance
                           that the problem is temporary or the Nak is
                           ambiguous.}
                  canImmediatelyRetry:  {all bits reported by this
                           Nak could be immediately retransmitted at
                           implementation option and the retry timers
                           updated};
                     end;
                        {setup error record and call procedure
                        errorStatistic here.}
                        if tryData then sendCode:= 1 else sendCode:= 0
                  end
      end
end {processNak}.
```

The following function represents the Nak packet acceptance condition. A Nak is only for outstanding data.

```
function acceptNak (crPtr:CRpointer; rPkt:PKTpointer):Boolean;
   begin
      with crPtr↑, rPkt↑ do
         acceptNak := (Stimer > 0) and ((Sou ≤ Pid + Pdl) and
            (Pid + Pdl ≤ Sowle))
   end {acceptNak}.

procedure errorStatistic (error RecPtr = ↑Error Record);
   begin
      {Updates counters, event records keeping diagnostic statistics}
   end {errorStatistic}.
```

6.6.6 Sending a Nak

Nak packets may be sent as a result of a packet's lifetime having expired (see Section 6.6.1) or as the result of a Data packet having been rejected as out-of-sequence (see Section 6.6.2).

The Nak packet header fields are generated from the header of the packet being Naked and the argument nakReason.

```
procedure sendNak ({args}crPtr:CRpointer; rPkt, sPkt:PKTpointer;
    nakReason,remainingLifetime:integer);
     {rPkt is pointer to packet being Naked and sPkt is
       buffer for Nak packet being formed}
    begin {Several of the fields in the packet header are left alone}
       sPkt.Pver:= rPkt.ver;
       sPkt.Presl:= 0;
       sPkt.Ptype:= Nak;
       sPkt.Pdn:= true;
       sPkt.PprtctLev:= rPkt.PprtctLev;
       sPkt.PΔtexp:= rPkt.PΔtexp;
       sPkt.Plifetime:= 2**rPkt.PΔtexp + remainingLifetime {assume
          remainingLifetime ≤ 0};
       sPkt.Pid:= rPkt.Pid;
       sPkt.Pdestaddr:= rPkt.Poriginaddr;
       sPkt.Poriginaddr := rPkt.Pdestaddr;
       sPkt.PnakRes := 0;
       sPkt.PnakReason := nakReason;
       sPkt.Pdl:= rPkt.Pdl;
       Pheaderchecksum (sPkt);
    end {sendNak}.
```

Acknowledgment

Delta-t was designed with John Fletcher. Jed Donnelley played an important role in motivating the need for a reliable transaction oriented protocol, the sender and receiver not having to agree on the values for the components of a common Δt, and for rendezvous-at-the-sender. Dan Nessett, Bob Judd, and Lansing Sloan made many helpful suggestions. The design benefited significantly from interactions with members of the ARPA TCP protocol design community, particularly Jon Postel. Valuable discussions with Ann Duenki and Peter Schicker of the former EIN community influenced several decisions.

References

1. D. Belsnes, "Single-Message Communication," IEEE Transaction on Communications COM-24, No. 2 (1976).

2. S. R. Bunch, J. D. Day, "Control Structure Overhead in TCP," Proc. Trends and Applications: 1980 Computer Network Protocols, IEEE Cal. No. 80 CH 1529-7C, May 1980, pp. 121-128.

3. J. Burruss, et al, "Specification of the Transport Protocol," NBS Report ICST/HLNP-81-1, Feb. 1981.

4. J. G. Fletcher, R. W. LWatson, "An Overview of the LINCS Architecture," LLNL Working Document, July 9, 1981.

5. J. G. Fletcher, "LINCS Security Policy," LLNL Woirking Document, July 15, 1981.

6. J. G. Fletcher, R. W. Watson, "Mechanisms for a Reliable Timer-Based Protocol," Computer Networks, No. 4/5, September/October, pp. 271-290. Also in Proceedings Computer Network Protocols Symposium, Liege, Belgium, February 1978, p. C5-1/C5-17.

7. L. Garlick, R. Rom, and J. Postel, "Reliable Host to Host Protocols: Problems and Techniques," Proceedings 5th Data Communications Symposium, IEEE/ACM, September 1977.

8. J. L. Grange, M. Gien, eds., Flow Control in Computer Networks, IFIP, North-Holland Publishing Co., 1979.

9. J. F. Haverty, R. D. Rettberg, "Inter-process Communication for a Server in UNIX," Proceedings Compcon 78, September 1978, pp. 312-315.

10. ISO, "Draft Connection-oriented Basic Transport Protocol Specification," June 1981.

11. J. M. McQuillan, V. G. Cerf, Tutorial: A Practical View of Computer Communications Protocols, IEEE Catalog No. EHO 137-0, 1978.

12. J. B. Postel, "Specification of Internetwork Transmission Control Protocol," TCP Version 4, Jan. 1980, available through Defense Advanced Research Projects Agency, IPTO.

13. L. Pouzin and H. Zimmermann, "A Tutorial on Protocols," Proceedings IEEE, Vol. 66, No. 11, November 1978, pp. 1346-1370.

14. C. A. Sunshine and Y. K. Dalal, "Connection Management in Transport Protocols," Computer Networks, Vol. 2, No. 6, 1978, pp. 454-473.

15. R. W. Watson and J. G. Fletcher, "An Architecture for Support of Network Operating System Services," Proceedings 4th Berkeley Conference on Distributed Data Management and Computer Networks, August 1979. (Also in Computer Networks, Vol. 4, No. 1, pp. 33-49, February 1980.

16. R. W. Watson, "Timer-based Mechanisms in Reliable Transport Protocol Connection Management" Proc. Computer Network Protocols, sponsored by NBS and IEEE, Gaithersburg, Maryland, 29 May 1980. (Also in Computer Networks 5 (1981) 47-56.

17. R. W. Watson, "Interprocess Communication: Interface and End-to-End (Transport) Protocol Design Issues," Distributed Systems Architecture and Implementation: An Advanced Course, Springer-Verlag, Berlin, NY, 1981.

18. R. W. Watson, "Distributed System Architecture Model," Distributed Systems, Architecture and Implementation: An Advanced Course, Springer-Verlag, Berlin, NY, 1981.

19. R. W. Watson, "DeltaGram Protocol Specification," LLNL Working Document, Aug. 15, 1981.

20. R. W. Watson, "Service Support Level Protocol Specification," LLNL Working Document, Aug. 15, 1981.

21. R. W. Watson, "LINCS Interprocess Communication (Transport) Service Specification," LLNL Working Document, Aug. 15, 1981.

22. N. Wirth, K. Jensen, PASCAL User Manual and Report; Springer-lVerlag, N.Y., Berlin, 1978.

APPENDIX A

Notes on Timer Values and Rules

The purpose of this appendix is to outline the arguments leading to the requirements that the Stimer run 3Δt and that the Rtimer run 2Δt. The value of Δt used can be different for each direction of data movement on an association. The conditions that the timer intervals must satisfy are the following.

A. Rtimer Conditions

1) (Assurance) No duplicates can be accepted.
2) (Smooth flow) Guarantee that any packet sent with Pdrf = false that arrives at the receiver after a predecessor packet sent with Pdrf = true will be acceptable (will arrive before the receiver's Rtimer has run out).

B. Stimer Conditions

1) (Assurance)
   a) Allow a graceful close (do not close until all data or packets sent needing Acks can be acknowledged).
   b) Assure that no SN will be reused until all previous packets or their Acks or Naks using the SN have died. This condition is not necessary for Naks (see Section 6.6.1)

2. (Smooth flow) Run equal to or longer than Rtimer to guarantee acceptable SN's are generated.

The timer rules given in Section 2.6 satisfy the above conditions on assumption that the sender initializes the Lifetime for an element (bit) to Δt. The term Δt has a different meaning in this specification than it did in the original paper [6]. The term Δt, as used here, is the sum of three estimates on the sender's part, no one of which needs bounding individually if their sum is bounded:

R = time the sender would normally expect to retransmit.
MPL = maximum packet lifetime or a worst case estimate of network travel time.
A = time for receiver to generate an acknowledgment.
Δt = R + MPL + A.

Timer Rules

Condition A-1 needs Rtime > Δt

Condition A-1 is satisfied by the interval Δt because the receiver sets its timer whenever it accepts an SN. No bit can live longer than Δt by R.5 (see Section 2.6). (Note: the receiver cannot just set its Rtimer from the value in the Plifetime field of the accepted packet because the rule for counting it down requires at least one tick for each link and node a packet traverses even if the time spent on that link or in that node is infinitesimal. Therefore, two identical packets going by different routes could live different times relative to R timer and cause a duplicate to be accepted.)

Condition A-2 needs Rtime > 2Δt

The timer rules R.1 through R.6 assure that Condition A-2 is satisfied by the interval 2Δt. The following worst case scenario requires this interval.

1) A packet P_i with Pdrf = true is emitted by the sender and arrives instantly at the receiver. The receiver sets Rtimer.
2) Because of lost Acks requiring packet retransmission or delayed Acks, no Ack to P_i has arrived at the sender at Δt-x (where x is a very small number).
3) The instant Δt-x is the last moment when a packet containing new elements can be emitted by the sender because of rule R.2. This packet will have Pdrf = false because P_i was unAcked at the time it was sent.
4) In the worst case it could arrive at the receiver at 2Δt-x since the Rtimer was set in step 1. For a packet with Pdrf = false to be accepted, Rtimer > 0, therefore yielding the need for Rtimer to run 2Δt.

Condition B-1 needs Stime > 2Δt

1) A Packet can live at most Δt by rule R.5. A Data packet and its Ack packet can live at most 2Δt, with no gap in the timing of their lifetimes.
2) For the same reason above, if an Ack is ever going to be received, it will be received within 2Δt.

Condition B-2 needs Stime > 3Δt

A Data packet can take a maximum of Δt to reach the receiver which will set its Rtimer to 2Δt at that instant. Therefore, in the worst case the Rtimer can run at most 3Δt relative to the time of setting of Stimer.

Crash with Loss of Memory

Sender Sender must wait 3Δt. The sender wants to be able to choose any initial sequence number and be assured that:

1) it will be accepted, implying that Rtimer must have gone to zero (the Δt being discussed is the Δt used by the sender before the crash).
2) any Data packets sent prior to the crash or their Acks that might have the same SN have died.

Condition 1 above requires 3Δt because a packet emitted just before the crash could take Δt to reach the receiver. The Rtimer would then run 2Δt from that point.

Condition 2 above would be satisfied by 2Δt as discussed earlier.

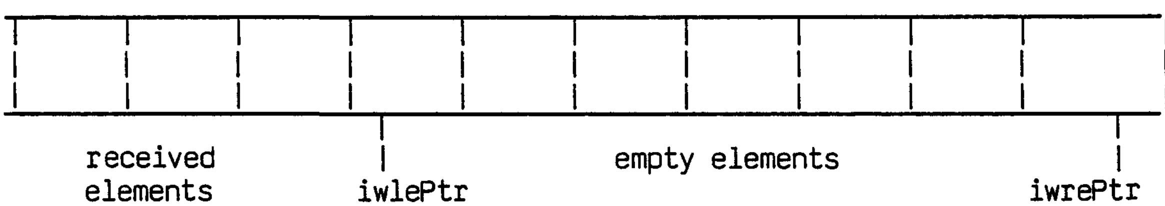
Receiver Receiver must wait Δt.

The receiver wants to be assured that it does not accept any duplicate of any SN accepted before the crash. Waiting Δt before accepting Data or Rendezvous packets is sufficient for this need. Ack or Nak packets must only provide information on data sent after deadstart and waiting the 3Δt interval above assumes no old Acks will still exist. Given the way the algorithm of Section 6 is written Naks might exist longer but no harm results. (Note: If we changed the rule that says "senders keep retransmitting when an element has had its maximum retransmission interval" to "not retransmitting and freezing the Lifetime," the wait after a crash would increase to $2\Delta t$. All other timer values would be the same, but the derivation would be as per reference [7].)

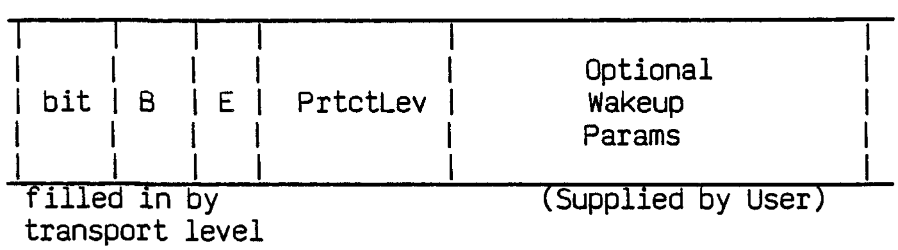
APPENDIX B
EIM Interface State Record Definition and IPC User Interface Operations

Interface State Record

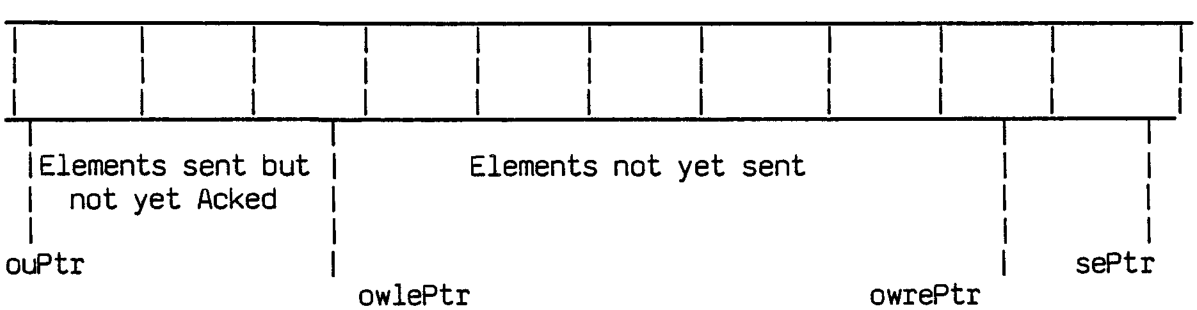
This appendix contains a logical view of the data structure maintained for each association by the EIM. We call this an Interface State Record (ISR). The full duplex bit streams for an association and the interface to the next higher level can be abstractly represented by four queues and a set of associated state variables. There is one queue at each end of the association for each direction of data flow. A queue element consists of a bit labeled with appropriate attributes (B, E, protection level). The association queues are shown in Figure B.1. Figure B.2 illustrates the ISR necessary to represent the queues, as well as the associated state variables. Additional state information may be necessary depending on the implementation and the internal interface to Delta-t (e.g., a flag may be necessary to remember that a DtAck call should be issued).



(a) Receive Queue



(b) Receive Queue Element



(c) Send Queue

bit	B	E	PrtctLev	Optional Wakeup Params

(d) Send Queue Element

Figure B.1. Association Queues

iwlePtr	Receive State
iwreOffset	
RSind	
ouPtr	Send State
owleOffset	
owreOffset	
seOffset	
sendCode	
giveupError	

Figure 2. Interface State Record

ISR Definition

Receive State

iwlePtr

Purpose: The input-window-left-edge pointer. Pointer to the next empty element in which to place a labeled bit of data. It is valid only if iwreOffset>0.

Default: 0 (head of queue)

When changed: Incremented by the EIM as a bit is taken from an accepted Data packet.

iwreOffset

Purpose: An offset relative to iwlePtr defining the number of available empty elements, the input window. iwrePtr = iwlePtr + iwreOffset.

Default: 0.

When changed: Updated by the User Receive or Receive-Abort procedures and decremented by the EIM when a bit is received.

RSind

Purpose: A flag set true when rsFlg returned from DtPktRcvd is true indicating that the correspondent sending port desires to send on this association and to be reliably informed when empty receive queue elements are available.

Default: false.

When changed: Set true by the EIM when rsFlg returned true from DtPktRcvd. Reset when DtAck called with rsFlg indicating nonzero window exists (empty elements added to receive queue).

Send State

ouPtr

Purpose: Pointer to the oldest unacked element. A pointer to the oldest (lowest numbered) element sent but not yet acknowledged. There is an element to be Acked only if owleOffset>0.

Default: 0 (head of queue)

When changed: Incremented by EIM as each bit sent is Acked (ackOffset returned in DtPktRcvd).

owleOffset

Purpose: An offset (output-window-left-edge) relative to ouPtr defining owlePtr, a pointer to the next element to be sent for the first time. There is an element to be acked only if>0.

Default: 0.

When changed: Incremented by the EIM whenever a bit is sent for the first time. Decremented by EIM as each bit sent is Acked.

owreOffset

Purpose: An optional variable in the interface. An offset (output-window-right-edge) relative to ouPtr defining owrePtr, pointing one queue position beyond the highest numbered element that the receiver could accept. This variable should not be used to determine when to issue DtStartSend calls. It is only of value to reinitialize the Sowre variable in the CR if the ISR persists longer than the CR.

Default: n (some default, receivers are initially willing to accept.)

When changed: Updated by EIM from owreOffset returned by the DtPktRcvd.

seOffset

Purpose: An offset relative to ouPtr defining sePtr (send-end), the next queue position to add an element for sending.

Default: 0 (head of queue)

When changed: Incremented or decremented by the User Send and Abort procedures and by the EIM as bits are Acked.

sendCode

Purpose: Code indicating whether or not a DtStartData call can or should be issued.

0 = do not issue DtStartData call even if nonzero output window exists as some protocol condition is blocking data sending.

1 = issue DtStartData call when there is data available for sending even if a zero output window exists. The EIM's sending (calling DtStartData) strategy when sendCode = 1 must recognize the following possibilities. When the output window, represented by (owreOffset-owleOffset), is less than desired, including zero, an Ack reporting a larger window could have gotten lost. Therefore, as a minimum, when there is an E-bit in the send queue (an implied wakeup for higher level action) or after some EIM time interval a DtStartData call should be issued. If the output window is zero Delta-t will enter a reliable rendezvous-at-sender procedure and sendCode will return 0 and polling will not be required. If the output window is positive Delta-t will send as much data as will fill the output window (and keep retransmitting until an Ack is received) and the resulting Ack will report the latest window.

giveupError

Purpose: An error code set when LINCS-IPC gives up trying to send data on an association. The value of the code indicates the giveup reason (to the best of LINCS-IPC ability).

Default: 0.

When changed: When giveup occurs or the ISR is reset.

SQE = send-queue-element
(all fields in SQE set by User)
bit - 0 or 1;
B - B mark as defined earlier.
E - E mark as defined earlier
prtctLev - protection level
wakeup - optional implementation dependent parameters to be used by the interface wakeup algorithm.

RQE = receive-queue-element
(fields set by LINCS-IPC when a bit is received)
bit - 0 or 1
B - B mark as defined earlier
E - E mark as defined earlier
prtctLev - protection level
(field set by User)
wakeup - as defined for SQE.

User Operations on an Association

The abstract User primitives below are viewed as being implemented within the LINCS-IPC layer. These are separate from the Delta-t primitives of Section 6, although they are reflected to the Delta-t primitives. The IPC User manipulates the ISR by calls to these primitives. The primitives below only define needed functionality. Reference [21] discusses issues associated with creating a practical IPC-User interface. We assume that the interface module implementing the primitives handles synchronization between the two asynchronously running User and LINCS-IPC layers by some appropriate mechanism, such as a monitor.

User Primitives

All the procedures except Wait use the appropriate ISR for the indicated association.

```
Assoc = record {unordered pair of LINCS addresses}
     address1,
     address2:  {LINCS address}
     end.

procedure Receive (a:Assoc; e:RQE);{Places an empty element on the Receive
                                    Queue}
     begin
         {Places e on the Receive Queue};
         iwreOffset:= iwreOffset + 1
     end {Receive}

procedure Send (a:Assoc; e:SQE); {Places an element to be sent on the Send
                                  Queue}
     begin
         {Places e on the Send Queue};
         seOffset:= seOffset + 1
     end {Send}

procedure SendAbort (a:Assoc); {removes an element from the indicated send
         queue}
     begin
         {remove a SQE and decrement seOffset.  Only elements
                                              not yet sent can be removed}
           end {Send Abort}

procedure ReceiveAbort (a:Assoc); {removes an element from the indicated
receive queue}
     begin
         {remove a RQE and decrement iwreOffset.  Only elements not
                                    yet filled by LINCS-IPC can be removed}
         end {ReceiveAbort}.

procedure Wait
     begin
         {The caller is blocked until a wakeup condition for any of its
         associations becomes true.  The wakeup conditions are implementation
         dependent, but are assumed to follow the guidelines of Section 2.6.
         The wait/wakeup mechanism is assumed to handle correctly any close
         call conditions resulting from asynchronously running User and
         LINCS-IPC modules.  When a Wait is issued, wakeup may be immediate as
         a result of the current state of the ISR and queue elements.}
     end {Wait}
```

```
procedure Status (a:Assoc; var SR:ISR);
    begin
        {The fields of ISR are copied into SR.  If the giveupError field is
        nonzero the ISR send state is reset to default values.}
    end {Status}
```

Managing the Interface-State-Record

The abstract IPC service specified above is defined in terms of permanent associations. That is, it is assumed that each node supported a permanent ISR for all possible associations with which it could be involved. This is clearly not practical. One would like to only maintain ISR's for active associations, i.e., those involved in a "conversation". Further, it is often the case that the identifiers of one or both ends of an association are not known at the point when an ISR must be allocated. This is common in the case of Server processes, since the address of a Customer port that may request service cannot be known ahead of time, yet state and buffer resources must be allocated to receive the requests.

To deal with these issues the notion is introduced of allowing the User to specify that an ISR can only be used for receiving with either a specific association (specific-ISR) or can be used with any of an indicated set of associations (any-ISR). A specific-ISR has both ports of the association with which it can be used completely defined as two full 64 bit LINCS addresses. The any-ISR has one or both ports incompletely defined. We define a new primitive for this purpose.

```
procedure Allocate (a0, a1:Address; var flg:Boolean);
    begin
        {a0 and a1 define the ends of the association(s) that can
        utilize the allocated ISR.  If a0 and a1 are fully specified
        then we call it an Allocate-specific.  If either a0 or a1 are
        incompletely specified then we call it an Allocate-any, where "any"
        refers to any association that matches the specified parts of a0
        and a1.  flg returns false if an ISR could not be allocated.  If
        an ISR already exists for the (specific) association, Allocate does
        nothing.}
    end {Allocate}
```

When any of the primitives of the previous section are issued, a check is first made of all specific-ISRs for the local port. If one is found, its state controls the transfer, otherwise an error exists. When data is received at a port a check is made of the specific-ISRs. If one is found, it controls the receipt of the data. If one is not found, then the any-ISRs are examined and the first one that can be used with (matches) the desired association is made specific. If none are found, the sender is flow control blocked.

NOTE - In an actual implementation the function of the Allocate primitive could be combined with the Send or Receive primitives as is done in Appendix A of reference [21].

Allocate, as defined here, has no end-to-end significance. It only allocates a local ISR. The "Open", "Call Establishment" or other such primitives defined in many other transport interfaces often do have end-to-end significance as well as cause an ISR to be allocated [3,10,12]. They are used for the User level synchronization function that the B mark provides in the LINCS service, supported by Delta-t. They are also used to indicate when to establish transport protocol connection records and connection management packet exchanges and may have other purposes not needed in Delta-t.

Having allocated an ISR one then needs to define when and how it is deallocated.

```
procedure Deallocate (a0, a1:Address);
    begin
        {The system searches for a specific-ISR or any-ISR that matches
        the a0, a1 pair and deallocates it.}
    end
```

The Deallocate primitive has no end-to-end significance. The end-to-end synchronization User level significance of "Close", "Disconnect" or other such primitives found in some transport services is provided in LINCS with higher level conventions in the data. Delta-t does not require hints from the User interface or end-to-end control communication in regard to when to discard its state.

The Deallocate primitive's functionality can be combined with other primitives such as Abort. The Send and Receive Interface state can also be separately allocated and deallocated. For example, the receive state could be deallocated if all available buffer space were Aborted or if an E-bit arrived.

Queue Structure

The IPC service specified above was defined in terms of queues of individually labeled bits. It is unlikely in practice to be implemented in such an abstract form. A more likely implementation will create the logical queues by use of chained block buffers (first bit address and count), where all the bits in a buffer are labeled with the same security level and only the first bit in the buffer may be labeled with a B mark and only the last bit may be labeled with an E mark. Wakeup conditions are also likely to involve buffer boundaries or completions. An example block buffer based interface is given in Appendix A of reference [21].

Window Advertisement

The window advertised in Delta-t Ack packets is logically the number of available elements in the receive queue of the specific-ISR for an association. If no specific-ISR is available for an association a default window should be advertised. When a specific-ISR is deallocated an Ack packet advertising the default window should be sent.